Korrekte Steuerungssoftware

Dissertation

zur Erlangung des Grades eines

`Doktors der Naturwissenschaften`

der Technischen Universität Dortmund
an der Fakultät für Informatik

von
Günter Graw

Dortmund
2010

Tag der mündlichen Prüfung: 29.04.2010

Dekan: Prof. Dr. Peter Buchholz

Gutachter: Prof. Dr. Heiko Krumm, Prof. Dr. Peter Herrmann

Inhaltsverzeichnis

Zusammenfassung

Muster sind ein bedeutender Bestandteil der objektorientierten Softwareentwicklung. Sie haben Einfluss auf alle Phasen und viele Domänen der industriellen Softwareentwicklung, unter anderem auch auf das Gebiet der Steuerungssoftware für industrielle Anlagen. Sehr aktuelle Standards – wie die OPC UA und die IEC 61131-3 – zeigen, dass objektorientierte Modellierungstechniken auch für die Erstellung von Steuerungssoftware bedeutsam sind. Anderseits muss Steuerungssoftware realzeitfähig, fehlertolerant und verteilbar sein und muss zusätzlich hohen Korrektheitsansprüchen genügen, wie sie nur durch Verwendung formaler Methoden gewährleistet werden können, da fehlerhaftes Verhalten zu große Risiken birgt. In dieser Arbeit wird der Ansatz verfolgt, objektorientierte Analyse und Entwurfsmodelle, die musterbasiert mittels UML erstellt worden sind, auf der Basis des Spezifikationsstils cTLA [HK00], der auf Lamports TLA basiert [Lam94], zu formalisieren. Dabei sind Analyse- und Entwurfsmuster von zentraler Bedeutung, die zur Entwicklung sog. Verfeinerungsmuster verwendet werden. Durch ein Verfeinerungsmuster wird ein Analysemuster an das Anforderungen bzgl. Verteilung, Fehlertoleranz und Realzeitverhalten bestehen, durch mehrere Entwurfsmuster verfeinert. Für ein Verfeinerungsmuster wird die korrekte Verfeinerung unter Verwendung von cTLA und TLA nachgewiesen. Dadurch lassen sich Sammlungen von Verfeinerungsmustern erstellen, die die Basis für die Entwicklung korrekter Steuerungssoftware bereitstellen. Weiterhin werden geeignete Techniken und Werkzeuge zur Übersetzung nach cTLA und die Verifikation von Verfeinerungsmustern behandelt und ihre Anwendung im Rahmen der Erstellung der Steuerungssoftware für eine Beispielanlage demonstriert.

Danksagung

Mein Dissertationsprojekt erforderte die Unterstützung mehrerer Personen, denen ich viel zu verdanken habe.

Meinem Betreuer, Herrn Prof. Dr. Heiko Krumm, möchte ich für die kontinuierliche Unterstützung danken. Dabei möchte ich besonders seine geduldige Bereitschaft zur Anleitung meiner wissenschaftlichen Arbeit hervorheben. Seine konstruktive Kritik war mir stets eine große Hilfe. Letztlich wäre diese Arbeit ohne sein Engagement im „Graduiertenkolleg zur Modellierung und modellbasierten Entwicklung komplexer, technischer Anwendungssysteme“ nicht möglich gewesen.

Herrn Prof. Dr. Peter Herrmann vom Institut für Telematik der NTNU in Trondheim gebührt mein besonderer Dank für die Möglichkeit, meine Ideen in seiner Arbeitsgruppe vorstellen zu dürfen. Dabei habe ich wertvolle Impulse für die Weiterentwicklung meiner Arbeit erhalten. Außerdem bedanke ich mich für die zügige Erstellung des Zweitgutachtens.

Bei Herrn Prof. Dr. Peter Marwedel bedanke ich mich für seine Bereitschaft, den Vorsitz über die Prüfungskommission zu übernehmen.

Meiner Frau Tanja und meinem Sohn Lukas danke ich für ihre Geduld.

Kapitel 1

Einführung

In diesem Kapitel soll die Verifikation objektorientierter Steuerungssoftware motiviert werden. Dazu wird ein Überblick sowohl über den technischen Entwicklungsstand als auch über den Beitrag dieser Arbeit gegeben werden.

1.1 Gegenwärtiger technischer Entwicklungsstand in der Sicherheitstechnik

Bei der Kontrolle komplexer, technischer Prozesse, wie sie z. B. in der chemischen Industrie auftreten, kommt der Informationsverarbeitung in Gestalt der Leittechnik eine entscheidende Bedeutung zu. Deshalb ist es nur naheliegend Informatikmethoden für den Entwurf, die Implementierung und den Betrieb leittechnischer Einrichtungen anzuwenden.

Bei diesen leittechnischen Einrichtungen spielen Realzeitsysteme, bei denen es sich um hochverlässliche korrektheitsgesicherte Softwaresysteme handelt, eine große Rolle. Das Fachgebiet der korrektheitsgesicherten Echtzeitsysteme steht allerdings gegenwärtig erst am Anfang der Erforschung. Die Ursachen für die Bedeutung dieses Gebietes liegen zum einen im steigenden Sicherheitsbewusstsein unserer Gesellschaft, welches auf dem Wunsch beruht, menschliches Leben und Umwelt zu schonen. Zum anderen liegen sie in der Tendenz zu flexiblen, sich ändernden Anforderungen adaptierbaren und realzeitfähigen, auf Software basierenden Steuer- und Regelgeräten und Systemen. Diese werden für die Überwachung eingesetzt, um die Produktionsmenge und Qualität von Gütern zu steigern. Dabei soll der Verlässlichkeit informationsverarbeitender Systeme in hohem Maße vertraut werden können. Dies soll durch eine formelle Zulassung durch die zuständigen Aufsichtsbehörden ermöglicht werden.

Gegenwärtig wurden noch keine Lizenzen für korrektheitsgesicherte Software vergeben, weil Software wesentlich unzuverlässiger als Hardware ist. In einer Software können im Gegensatz zur Hardware, in der auf Verschleiß beruhende Fehler auftreten können, nur Entwurfsfehler auftreten, d. h. sie sind in der Systematik des Entwurfs begründet. Diese Entwurfsfehler können nur unter Anwendung mathematischer Korrektheitsbeweise beseitigt werden.

Bisher existieren eine Reihe praxiserprobter Methoden und Richtlinien, die ihren Nutzen für den Entwurf, die Verifikation und die Validation unter Beweis gestellt haben. Diese versagen jedoch, wenn die Korrektheit größerer Systeme bzw. Programme mit mathematischen Methoden nachgewiesen werden soll. Das ist der Grund dafür, dass Abnahmebehörden große Zurückhaltung bei der Erteilung von Lizenzen für softwaregesteuerte Systeme der Leittechnik üben. Das kann zu hohen Kosten für die beteiligten Unternehmen führen, da Anlagen aus rechtlichen Gründen nicht in Betrieb genommen werden dürfen.

Die Sicherheitstechniken aus den Ingenieurdisziplinen basieren auf den zufallsverteilten Ausfällen technischer Systeme, wodurch im Entwurf entstandene Fehler unberücksichtigt bleiben, weil angenommen wird, dass sie durch Validierungsmethoden vor Auslieferung eines Produktes beseitigt werden können.

Noch existiert keine ausreichende Unterstützung für den Entwurf und die Implementierung korrektheitsgesicherter Softwaresysteme sowie für die Durchführung korrektheitssichernder Beweise an sich. Deswegen sind Lösungen eher individuell. Es hat sich noch keine allgemein anerkannte Sicherheitstechnik etabliert, die die Entwicklung solcher Systeme ermöglicht. Deswegen sind Korrektheitsbeweise von Software extrem aufwändig oder gar unmöglich.

1.2 Sicherheit in der Prozessautomatisierung

Der Begriff der Sicherheit ist in dieser Arbeit von zentraler Bedeutung, wobei die englische Übersetzung safety ist. Bei Wikipedia [1] findet man folgende Definition:

> „Sicherheit bezeichnet einen Zustand, der frei von unvertretbaren Risiken der Beeinträchtigung ist oder als gefahrenfrei angesehen wird. ... Anders als im angloamerikanischen Sprachraum wird im Deutschen normalerweise nicht zwischen den beiden Themen "Security" und "Safety" unterschieden. Beide Begriffe werden stattdessen allgemein unter "Sicherheit" zusammengefasst. Während "Safety" den Schutz der Umgebung vor einem Objekt, also eine Art Isolation beschreibt, handelt es sich bei "Security" um den Schutz des Objektes vor der Umgebung. "

Der Zusammenhang zwischen der Zuverlässigkeit und der Sicherheit von technischen Systemen wurde schon relativ früh in der Raumfahrt- und der Luftfahrttechnik erkannt. Die Aussage, die man erhält ist: Je zuverlässiger ein System arbeitet, desto sicherer ist es. Um die Zuverlässigkeit von Systemen zu beschreiben, hat es sich als zweckmäßig erwiesen, Betrachtungen auf der Basis von mathematischen bzw. statistischen Modellen vorzunehmen. Für viele Methoden zur Behandlung der Zuverlässigkeit gibt es mittlerweile Standards.

Der Zusammenhang, der sich aus Zuverlässigkeit, Wirtschaftlichkeit und Sicherheit ergibt, ist inzwischen stärker in das Bewusstsein getreten und hat sich besonders stark auf die Genehmigung großtechnischer Anlagen, wie Kernkraftwerke, Chemieanlagen und moderne Nahverkehrssysteme, ausgewirkt.

Im Bereich der Sicherheit existieren keine allgemein anerkannten Methoden. Stattdessen wird das Problem der Sicherheit anwendungsorientiert behandelt. Damit wird die Behandlung von Sicherheitsaspekten zwangsweise sehr stark durch das jeweilige Anwendungsgebiet geprägt, wie z. B. in der Chemie- oder Kerntechnik. Erst die Arbeit der letzten Jahre hat verdeutlicht, dass sich die Ergebnisse aus den unterschiedlichen Anwendungsgebieten sehr stark ähneln bzw. sogar identische Sicherheitsprobleme auftreten.

Zentraler Gegenstand der Betrachtungen dieser Arbeit ist der Sicherheitsaspekt in der Prozessdatenverarbeitung. Deswegen sei zunächst die Definition des Begriffs Sicherheit, wie sie in DIN/VDE 31 000, Teil 2, festgelegt wurde, genannt:

> „Sicherheit ist eine Sachlage, bei der das Risiko nicht größer als das Grenzrisiko ist.
> Als Grenzrisiko ist dabei das größte, noch zu vertretende Risiko zu verstehen. "

Bei dieser Definition wird das Risiko zur Betrachtung herangezogen. Nach einer anderen Definition wird unter Sicherheit die Fähigkeit einer Einrichtung verstanden, innerhalb einer vorgegebenen Zeiteinheit zu reagieren. Im Gerätesicherungsgesetz (GSG) werden technische Vorschriften angegeben, die auf einer dreistufigen Gliederung der DIN/VDE-Normen beruhen. Diese Gliederung besteht aus Grund-, Gruppen- und Anwendungsnormen. Die DIN V 19 250 legt die grundlegenden Sicherheitsbetrachtungen für automatisierungstechnische Schutzeinrichtungen auf Basis der DIN/VDE 31 000 Teil 2 fest. Die IEC 61 508-1 definiert vier Sicherheitsklassen, die mit SIL 1-4 (Safety Integrity Level) bezeichnet werden. In [HK99] wird eine Empfehlung gegeben, die unterschiedlichen Sicherheitsklassen eine möglichst kleine Menge von Sprachkonstrukten zur Realisierung von Steuerungen, eine entsprechende typische Programmiersprache bzw. -methode und weiterhin eine Verifikationsmethode zuordnet. Als typische Programmiermethoden können Ursache-Wirkungstabellen, Funktionspläne mit verifizierten Teilsprachen, formal verifizierte Teilsprachen

[1] www.wikipedia.de

bzw. statische Sprachen mit sicheren Konstrukten verwendet werden. Das Spektrum der Verifikationsmethoden reicht über sozialen Konsens, die diversitäre Rückwärtsanalyse bis zu formalen Korrektheitsbeweisen. Das höchste Sicherheitsniveau wird SIL 1 genannt. Bei diesem Sicherheitsniveau werden alle aufgezählten Verifikationsmethoden verwendet. Von besonderem Interesse für diese Arbeit ist das SIL 2, das die formale Verifikation ermöglicht. Eine formal verifizierte Teilsprache entspricht weitestgehend Modula-2, ohne dass dynamische Konstrukte verwendet werden. Die formale Verifikation erstreckt sich allerdings nur auf Teilkonstrukte der Programmiersprache.

1.2.1 Anforderungen des Echtzeitbetriebes

Bei sicherheitsgerichteten, rechnergestützten Automatisierungssystemen erfolgt die Verarbeitung im Echtzeitbetrieb, welcher in DIN 44 300 wie folgt definiert wird:

> „Ein Betrieb eines Rechensystems, bei dem Programme zur Verarbeitung anfallender Daten ständig betriebsbereit sind, derart dass die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind. Die Daten können je nach Anwendungsfall nach einer zeitlich zufälligen Verteilung oder zu vorbestimmten Zeitpunkten anfallen."

Die in dieser Betriebsart arbeitenden Digitalrechner haben demzufolge die Aufgabe, Programme zu verarbeiten, die mit externen technischen Prozessen in Verbindung stehen. Die Ausführung der Programme muss zeitlich mit den auftretenden Ereignissen synchronisiert werden und schritthaltend mit den zugehörigen Prozessen verlaufen. Deshalb sind Echtzeitsysteme immer als in eine größere Umgebung eingebettet anzusehen und werden auch als "eingebettete Systeme" bezeichnet. Der Echtzeitbetrieb weicht von der allgemeinen Datenverarbeitung durch das explizite Hinzutreten der Dimension Zeit ab. Dieses drückt sich in der grundlegenden Anforderung nach Rechtzeitigkeit aus, welche auch unter extremen Lastbedingungen erfüllt sein muss.

1.2.2 Softwaretechnik für technische Systeme

Gegenwärtig ist die Softwaretechnik durch die Trends der Objekt- und Komponentenorientierung geprägt. Diese Trends beruhen auf der Hoffnung, das softwaretechnische Prinzip der Wiederverwendung von Softwarebausteinen zu verwirklichen [Szy00], was auch Auswirkungen auf die Adaptabilität bzw. Flexibilität von objektorientierten Systemen hat. Allerdings sind die Techniken zum Einsatz der Objekt- und Komponentenorientierung in verschiedenen Domänen unterschiedlich weit entwickelt. So hat der Einsatz der Objekttechnologie im Bereich von Geschäftssoftware in den letzten Jahren einen hohen Standard erreicht. Hier sind beispielsweise die Anwendung von Programmiersprachen, wie C++ oder Java, oder auch Middlewaretechnologien, wie CORBA [OPR96] aber auch die UML (Unified Modeling Language), zur Modellierung komplexer Systeme zu nennen. In der Domäne der Anlagensteuerungstechnik haben objektorientierte Konzepte bisher nicht die Bedeutung gefunden, wie im Bereich von Geschäftssoftware. Dies ist einerseits auf die Schwächen dieser Technologie im Bereich Realzeitverarbeitung zurückzuführen andererseits aber auch auf die Existenz etablierter Techniken, die Programmiersprachen auf niedrigem Abstraktionsniveau (etwa Assembler oder imperative Sprachen oder neuentstandene Normen wie die IEC 61131-3[2]) verwenden. Mit dem Vordringen der Objekttechnologie in den Bereich der Realzeitverarbeitung – auch für verteilte Anwendungen – gewinnen sie auch hier stärker an Bedeutung. Ein Maßstab hierfür sind die jüngst erfolgten neueren Entwicklungen:

- Die Definition und Entwicklung von RT-CORBA (Real-Time Common Object Request Broker Architecture) im Bereich realzeitfähiger Middleware durch die OMG (Object Management Group) und ihr angehörende Unternehmen [KSK03].
- Die Standardisierung der UML, die eine universelle Modellierungssprache ist, durch die OMG. Die UML kann zur Visualisierung, Spezifikation und Dokumentation von Softwaresystemen

[2]Bei der IEC 61131-3 handelt es sich um die einzig weltweit gültige Norm für Programmiersprachen im Bereich der speicherprogrammierbaren Steuerungen.

eingesetzt werden. Im April 2006 legte eine Task Force der OMG die UML 2.1 zur Standardisierung vor. Die UML liegt aktuell in der Version 2.2 als Standard vor. Gegenwärtig wird von der OMG an der Version 2.3 gearbeitet. Die MDA (Model Driven Architecture) [KWB03] Initiative der OMG versucht, UML-basierte Modelle unterschiedlicher Abstraktionsniveaus ineinander generativ oder elaborativ zu transformieren, wobei Korrektheitsaspekte bisher häufig vernachlässigt wurden, da es sich um ein junges Gebiet handelt. Forschungsergebnisse, die auch formale Aspekte berücksichtigen, sind in [GH02] vorgestellt worden.

- Die Definition eines Standards für die UML-RT (Unified Modeling Language for Real-Time) durch die OMG oder die Real-Time UML durch die Firma I-Logix. Durch die OMG wird ein Profil zur Spezifikation von Realzeit-Anforderungen bereitgestellt.

- Die OPC Unified Architecture (OPC UA) stellt einen aktuellen Standard dar, der objektorientierte Technologien für den Bereich der Automatisierungstechnik bereitstellt [MLD09]. Die Modellierung erfolgt mittels UML. Für den bedeutenden Standard IEC 61131-3 ist Ende November 2009 ein Vorschlag für eine Anpassung erschienen [Tec09], der auf der OPC UA beruht. Dieser Vorschlag passt gut zu den in dieser Arbeit vorgestellten Konzepten.

- Die Entwicklung von realzeitfähigen Technologien im Bereich der Programmiersprache Java. Eine besondere Herausforderung ist hierbei die realzeitfähige Speicherbereinigung (Real Time Garbage Collection). Hier sind insbesondere Real-Time Java von Newmonics [Nil98] und Embedded Java von Sun zu nennen. Weiterhin ist bei Sun unter Mitwirkung namhafter Experten aus der Industrie für Realzeitsoftware eine realzeitfähige Erweiterung von Java entwickelt worden, die als Real Time Specification for Java (RTSJ) [Wel04, Dib02] bezeichnet wird. Giotto ist eine an Java angelehnte Sprache für eingebettete Realzeitsysteme [HKMM02, HKSP02]. Exotasks [TABP07] sind ein aktuelles Folgeprojekt von Giotto. Es stellt ein taskorientiertes Verarbeitungsmodell auf der Basis von RTJS bereit. Als RTJVM (Real Time Java Virtual Machine) findet Metronrome [BCR03] von IBM Verwendung, die auf der Code-Basis der IBM J9 JVM, entwickelt worden ist. Mit Metronome und einem Realzeit-Betriebssystem – etwa dem mittlerweile als Open Source verfügbaren Real Time Linux[3]– können realzeitfähige Java Programme auf der Basis von RTSJ ausgeführt werden, wobei gegenwärtig Antwortzeiten im Bereich von einer Millisekunde garantiert werden können. Metronome unterstützt insbesondere die realzeitfähige Garbage Collection mit kurzen Zeitintervallen. Mit dieser interessanten Kombination von Projekten ist als eine der ersten Anwendungen ein Helikoptermodell mit vier Rotoren, der so genanannte Javiator [ABI+07], entwickelt worden.

- Das JEOPARD-Projekt[4] (Java Environment for Parallel Realtime Development) stellt Werkzeuge zur plattformunabhängigen Entwicklung von Echtzeit-Systemen für SMP Mehrkern-Plattformen bereit [Sie08]. Diese Werkzeuge werden die erhöhte Software-Produktivität und Wiederverwendbarkeit, die auf Desktop-Systemen mit Mehrkernprozessoren Standard sind, mit den speziellen Bedürfnissen von eingebetteten und Echtzeitsystemen verbinden. Das Projekt wird aktiv zu Standards für portierbare Software in diesem Einsatzgebiet, wie etwa der RTSJ (Real-Time Specification for Java), beitragen.

Auch auf der Grundlage dieser Technologien entstandene Systeme unterliegen selbstverständlich den Anforderungen aus dem Bereich der Sicherheitstechnik, z. B. denen der SIL. Allerdings fehlen aufgrund ihres relativ hohen Aktualitätsgrades hier anerkannte Methoden. Andererseits erscheint gerade wegen der Möglichkeit der Wiederverwendung die Entwicklung einer Methode zur Gewährleistung/Verifikation von Sicherheitseigenschaften und sei es durch Kombination mit anderen Methoden der Sicherheitstechnik sehr verlockend, da eine kostenintensive Sicherheitsüberprüfung nur einmal vorgenommen werden muss und sicherheitsgeprüfte Bausteine später in beliebigen Kontexten wiederbenutzt werden können.

[3] https://wiki.ubuntu.com/RealTime
[4] www.jeopard.org

1.2.3 Verifikationstechniken

Im Juli 2005 ist von Tony Hoare und J. Misra das sog. Grand Challenge Projekt [HM03] vorgeschlagen worden. Es wird angegeben, dass das US-Wirtschaftsministerium einen Schaden von 20 - 60 Milliarden Dollar pro Jahr durch vermeidbare Softwarefehler für die amerikanische Wirtschaft schätzt. Die Autoren argumentieren, dass die Zeit reif für die Konstruktion eines Werkzeuges zur Programmverifikation ist, welches logische Beweise verwendet, um eine automatische Prüfung eines als Eingabe bereitgestellten Programms vorzunehmen. Die Prototypen dieses Werkzeuges sollen auf einer gut fundierten und vollständigen Theorie der Programmierung beruhen. Weiterhin sollen diese durch ein breites Spektrum von Softwarekonstruktions- und Analysewerkzeugen unterstützt werden. Alle diese Werkzeuge sollen durch eine umfangreiche experimentelle Anwendung auf große und in der Anwendung befindliche Computerprogramme evaluiert und weiterentwickelt werden. Das Projekt soll die wissenschaftliche Basis zur Behebung vieler Arten von Programmierfehlern bereitstellen, die Entwicklern und Anwendern von Software nutzen kann. Als Zeitrahmen für das Projekt werden die nächsten beiden Dekaden dieses Jahrhunderts vorgesehen. Der Aufwand soll auf viele Nationen verteilt werden.

Als eine formale Spezifikationstechnik wird TLA (Temporal Logics of Actions) [Lam94, Lam03], eine lineare Temporalzeitlogik erster Ordnung, aufgeführt. Durch Aktionen werden Zustandsübergänge durch Änderungen von Zustandsvariablen spezifiziert. TLA erlaubt die Verifikation von Sicherheits- und Lebendigkeitseigenschaften temporallogischer Spezifikationen. Weiterhin ist es möglich mit TLA Realzeiteigenschaften zu spezifizieren. Zur Verifikation werden durch TLA Beweisregeln zur Verfügung gestellt. Mit der Spezifikationstechnik cTLA [HK95, Her98] wird der kompositionale Aufbau von TLA-Spezifikationen auf der Basis von Aktionenkopplungen mittels Konjunktion unterstützt.

Für TLA+ wird mit TLC (Temporal Logic Checker) ein Model-Checker zur Verfügung gestellt, der es ermöglicht, TLA-Spezifikationen mit eingeschränkter Mächtigkeit maschinell zu verifizieren. TLC ist von Yuan Yu und Leslie Lamport bei DEC und Microsoft entwickelt worden. TLC besitzt die Fähigkeit, im Fehlerfall einer Spezifikation Gegenbeispiele bereitzustellen. TLC ist bisher in größerem Umfang zur Hardwareverifikation [YML99], z. B. für den Alpha Prozessor, verwendet worden. Außerdem wird TLC bei Intel [LB03] für die Verifikation beim Chip-Design eingesetzt.

Neue Trends zur maschinellen Verifikation verwenden Techniken des maschinellen Lernens, um Boolesche Formeln zu verifizieren. Diese sog. SAT (satisfiability) Solver sind in der Lage eine Boolesche Formel auf Erfüllbarkeit zu prüfen. SAT bezeichnet auch das Boolesche Erfüllbarkeitsproblem. In der Regel müssen diese Formeln in CNF (Conjunctive Normal Form) für einen SAT Solver als Eingabe zur Verfügung gestellt werden.

1.2.4 Korrekheitsgesicherte Steuerungssoftware

Da die gegenwärtigen Methoden zur Entwicklung von objektorientierter Steuerungssoftware für keine, bzw. nur unzureichende Unterstützung zur Verifikation bereitstellen, scheint es sinnvoll hier Abhilfe zu schaffen. Vielversprechend erscheint die Kombination von bewährten Konzepten der objektorientierten Software-Entwicklung – wie etwa Mustern, bzw. objektorientierte Modellierung – mit den Methoden der formalen Verifikation. Der Ansatz die objektorientierte Modellierung für die Analyse und den Entwurf von Steuerungssoftware auf der Basis von Mustern, die durch die Spezifikationssprache cTLA formalisiert werden, wird in dieser Arbeit entwickelt. Dabei werden speziell Verfeinerungsbeziehungen zwischen Analyse- und Entwurfsmodellen betrachtet.

1.3 Beiträge der Arbeit

Das Ziel der Arbeit besteht darin eine Methode zur Korrektheitsprüfung von Entwürfen von Steuerungssoftware zu entwickeln, die an sich domänenübergreifend ist, aber im Rahmen dieser Arbeit auf die Leittechnik von Chemieanlagen ausgerichtet ist. Dabei soll die Anwendung objektorientierter Techniken und Methoden im Vordergrund stehen, um den Verifikationsaufwand unter dem Kosten- und Zeitaspekt tragbar zu machen. Für dieses Problemfeld hat es sich angeboten

mehrere Ansätze aus den Bereichen Objektorientierung und Verifikation zu kombinieren. Hier soll das Fazit der Kombination von Methoden aus den Gebieten Verifikation und objektorientierte Modellierung, welches für die Lösung den höchsten Nutzen versprach und in dieser Arbeit Verwendung gefunden hat, kurz umrissen werden:

- Die Beschreibung einer Architektur für die Erstellung von Steuerungssoftware chemietechnischer Anlagen wird vorgestellt.
- Die Definition eines Analysemuster-Systems für Prozesssteuerungssoftware unter Verwendung von UML 2.0 als Spezifikationssprache, dessen Muster verwendet werden, um das abstrakte Softwaremodell zu spezifizieren.
- Die Entwicklung eines Entwurfsmuster-Systems mit UML 2.0, das typische Anforderungen an Prozesssteuerungssoftware aus den Bereichen Fehlertoleranz, Realzeit und Verteilung erfüllt. Die Muster des Entwurfsmuster-Systems werden für die Erstellung des konkreten Softwaremodells verwendet.
- Die Definition des Begriffs des Verfeinerungsmusters in Anlehnung an [Sha03] wird vorgenommen. Dabei erfolgt die korrekte Verfeinerung eines Analysemusters auf der Basis einer Menge von Entwurfsmustern. Weiterhin wird die Spezifikation einer Sammlung von Verfeinerungsmustern für Prozesssteuerungssoftware unter Verwendung von UML 2.0 vorgestellt.
- Die Weiterentwicklung und Erprobung der Realzeit-Erweiterungen für cTLA werden vorgenommen. Diese erlauben es, persistente und volatile Wartezeiten für Aktionen auf der Grundlage von Uhrenvariablen zu spezifizieren. Dafür wird eine speziell an den cTLA-Spezifikationsstil angepasste Aktion *Tick* zur Erhöhung von Uhrenvariablen verwendet.
- Die Entwicklung des Tools UML2cTLA zur Übersetzung von verhaltensbeschreibenden Diagrammen aus UML 2.0 in cTLA. Die Integration von UML2cTLA mit bestehenden Tools zur Verarbeitung von cTLA-Spezifikationen (etwa cTc) und des Model-Checkers TLC wird demonstriert.
- Eine exemplarische Verifikation von ausgewählten Verfeinerungsmustern mit Unterstützung des Model-Checkers TLC (Temporal Logic Checker) – speziell zum Nachweis der Korrektheit von Refinement-Mappings – wird vorgenommen. Außerdem werden Handbeweise für den Nachweis von Realzeit-Eigenschaften von Verfeinerungsmustern unter Verwendung der Realzeit-Erweiterungen von cTLA durchgeführt.

Damit liegt eine praktikable Methode vor, Bestandteile von Software mit vertretbarem Aufwand formal korrekt zu entwickeln.

1.4 Aufbau der Arbeit

Im Kapitel 2 wird ein Überblick der Anforderungen an Steuerungssoftware und deren Architektur gegeben. Der verteilte Regler wird als kapitelübergreifendes Beispiel eingeführt. Das Kapitel 3 dient der Einführung der Syntax, der Semantik und der Konzepte von UML. Im Kapitel 4 wird der Musterbegriff erläutert. Um die Korrektheit von Mustern zu beweisen, werden im Kapitel 5 die Spezifikationssprache TLA und der Model-Checker TLC vorgestellt. Im Kapitel 6 wird der kompositionale Spezifikationsstil cTLA beschrieben. Analysemuster für das abstrakte Softwaremodell werden in Kapitel 8 entworfen. Weiterhin werden im Kapitel 7 Entwurfsmuster für das konkrete Softwaremodell von Steuerungssoftware behandelt. Im Kapitel 9 werden Verfeinerungsmuster für Steuerungssoftware konzipiert. Anschließend werden in Kapitel 10 und Kapitel 11 die Semantik der Bestandteile eines Verfeinerungsmusters und deren Übersetzung in cTLA vorgestellt. Schließlich wird der Beweis der Korrektheit eines Verfeinerungsmuster in Kapitel 12 geführt. Werkzeuge, die den Korrektheitsbeweis eines Verfeinerungsmusters unterstützen, werden in Kapitel 13 vorgestellt. Abschließend wird in Kapitel 14 die Anwendung von Verfeinerungsmustern im Rahmen einer Beispielanlage demonstriert. Im Ausblick werden mögliche Weiterentwicklungen für Software zur korrekten Systemsteuerung behandelt.

Kapitel 2

Architekturmodell für Steuerungssoftware

Die Geschichte der Prozessführung im Bereich der verfahrenstechnischen Produktionsbetriebe ist durch zwei Modellansätze geprägt [BE94]:

- Den prozessorientierten Modellansatz, welcher es gestattet, den technischen Prozess nach einem Top-Down-Verfahren – einem Verfahren, das vom Abstrakten zum Speziellen vorgeht [HF06] – zu zerlegen. Durch entsprechende Verfeinerung gelangt man schließlich zu den sog. Grundoperationen, die man nicht mehr zerlegen kann oder will. Umgekehrt lässt sich in diesem Modell die übergeordnete Prozessführung netzförmig aus Grundbausteinen zusammensetzen.

- Den anlagenorientierten Modellansatz [EKS92, Epp93, BE94], bei dem die Anlage und ihre Fähigkeit zur Prozessführung im Vordergrund stehen. Hier erfolgt der Aufbau im allgemeinen Bottom-Up. Dies ist eine Vorgehensweise, bei der aus dem Speziellen das Abstrakte erarbeitet wird [HF06]. Ausgangspunkte sind die Signale, die mit dem Feld ausgetauscht werden sowie die Motor- und Messstellenlisten. Darauf bauen dann Bausteine wie eine Motor- bzw. Stellungsregelung auf, in denen diese Signale unabhängig vom laufenden Prozess in einer für die sog. PLT-Stelle, die eine Schnittstelle zwischen einer leit- und einer anlagentechnischen Funktionseinheit bildet, typischerweise verknüpft werden. Weiterhin kann es je nach Automatisierungsgrad weitere Steuerobjekte geben. Diese werden nach ihrem Koordinationsgegenstand als Gruppensteuerung, Teilanlagensteuerung oder Anlagensteuerung bezeichnet.

Früher standen diese beiden Alternativen unvereinbar gegenüber, während in den letzten Jahren Ansätze zu einem integrierten Modell erarbeitet worden sind. Gegenwärtige Steuerungssoftware zeichnet sich durch ihre ausgeprägte Herstellerabhängigkeit aus. Deswegen sind in den letzten Jahren verstärkte Anstrengungen zur Normierung unternommen worden. Diese mündeten etwa in der IEC 61131-3 [HK99, NGLS98, Sei03], die eine Sammlung von Funktionen für Steuerungssoftware vordefiniert. Sie können bei Bedarf mittels Funktionsbibliotheken realisiert werden. Die Spezifikation der Funktionsbausteine kann hierbei mit einer Sprache[1] vorgenommen werden, die sich an gefärbten Petri-Netzen orientiert und ursprünglich unter dem Namen Grafcet [DAV95] publiziert wurde.

In diesem Kapitel werden zunächst in der Literatur etablierte Ebenenmodelle für die Produktion vorgestellt. Diese werden anschließend dafür verwendet, ein Architekturmodell für Steuerungssoftware zu entwickeln, das auch die Anwendung objektorientierter Techniken ermöglicht. Hierzu werden sowohl die statischen als auch die dynamischen Bestandteile der Architektur betrachtet. Die dynamischen Architekturbestandteile legen fest, wie Bestandteile der statischen Architektur interagieren. Von zentraler Bedeutung für das Architekturmodell ist der Begriff des

[1] Diese wird auch als Sequential Function Chart(SFC) bezeichnet.

Funktionsbausteins [EKS92, Ens00], der eine objektorientierte Behandlung von statischen Architekturbestandteilen für Anlagensteuerungssoftware ermöglicht und auch bei den dynamischen Bestandteilen wertvolle Unterstützung liefert. Funktionsbausteine konnten bereits mit der IEC 61131-3 behandelt werden. Die Verteilung von Funktionsbausteinen wird mit dem Standard IEC 61499 [Chr00, Pet00, Neu00], der ein allgemeines Modell und eine Methodik für die Beschreibung von Funktionsblöcken in einem implementationsunabhängigen Format bereitstellt, ermöglicht. Diese Methodik, die an den Bedürfnissen von Anlagensteuerungen ausgerichtete Diagramme zur Verfügung stellt, kann von Systementwicklern verwendet werden, um verteilte Steuerungssysteme zu konstruieren. Es ist somit möglich, ein System zu definieren, das durch logisch verbundene Funktionsbausteine aufgebaut ist, die auf unterschiedlichen Verarbeitungsknoten ablaufen. Der Standard definiert hierfür spezielle Funktionsbausteine, die als Client und Server auf unterschiedlichen Verarbeitungsknoten arbeiten können.

2.1 Ebenenmodelle der Produktion

In der Literatur existieren zahlreiche Darstellungen von Ebenenmodellen der Produktion. Polke stellt in [Pol94] ein Modell vor, das aus vier Ebenen besteht. Dieses Modell wird in Abbildung 2.1 mit der Unternehmensleitebene, der Produktionsleitebene, der Prozessleitebene und der Feldebene gezeigt. Die Unternehmens- und die Produktionsleitebene haben einen betriebswirtschaftlichen Schwerpunkt und die Auswirkungen von eingeleiteten Maßnahmen bewegen sich im Rahmen von langfristigen Zeiträumen. Detaillierte Angaben zur Unternehmens- und Produktionsleitebene sind der Literatur zu entnehmen, da in dieser Arbeit der Interessenschwerpunkt auf der Feld- und Prozessleitebene liegt, bei denen die Reaktionszeiten im Bereich von Sekunden bzw. Millisekunden liegen. Hier soll zur weiteren Vertiefung eine an Brack [Bra93], Lauber [Lau89], Polke [Pol94], Balzer und Epple [BE94] angelehnte Gliederung der Prozessleitebene in zwei Unterebenen vorgestellt werden, wodurch sich insgesamt folgendes Ebenenmodell ergibt:

- Unternehmensleitebene
- Produktionsleitebene
- Prozessleitebene
  - Prozessführungsebene
  - Prozesssicherungs-/Prozessstabilisierungsebene
- Feldebene

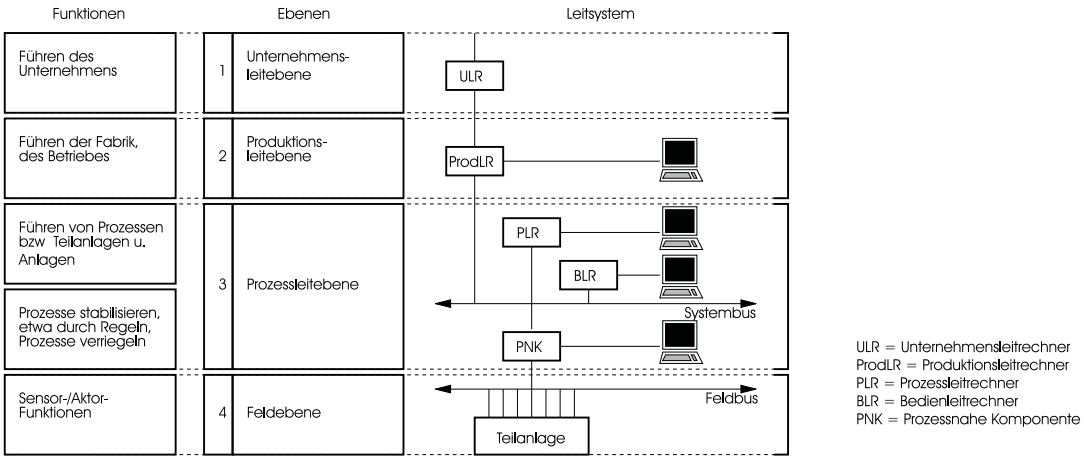


Abbildung 2.1: Die Ebenen der Produktion nach [Pol94]

Die Unterebenen zur Prozessleitung werden zwischen der Produktionsleitebene und der Feldebene aus Polkes Ebenenmodell eingefügt und substituieren somit die Prozessleitebene.

Im technischen Prozess selbst wirken mechanische Elemente, wie z. B. Behälter, Rührer, Mixer, Ventile, Rohre und Pumpen. Die Feldebene hat die Aufgabe – z. B. bei der Verletzung von Grenzwerten – direkt in den technischen Prozess einzugreifen, was durch die Verwendung von Sensor- und Aktorfunktionen, die physikalische Größen messen bzw. Stellgrößen beeinflussen, geschieht.

Auch wenn durch das Modell eine gewisse Übereinstimmung der räumlichen Verteilung und Kommunikationsinfrastruktur von Komponenten einer Anlage nahegelegt wird, ist das Modell im Wesentlichen als logische Gliederung zu verstehen, die Funktionen auf verschiedene Abstraktionsniveaus verteilt. Es ist durchaus möglich, die Funktionen mehrerer Ebenen in einer Komponente, wie z. B. einem intelligenten Sensor, zu vereinen. In Abbildung 2.1 werden die Ebenen mit ihren Funktionen aufgeführt. Die Abbildung zeigt außerdem die in der jeweiligen Ebene verwendeten Rechner. In den folgenden Abschnitten werden die Feldebene sowie die Unterebenen der Prozessleitebene in umgekehrter Reihenfolge, beginnend mit der Feldebene, vertiefend erläutert.

2.1.1 Feldebene

Durch die Feldebene werden Einzelfunktionen bereitgestellt. Einzelfunktionen sind Funktionen, die der Prozessführungsaufgabe eines Aktors oder Sensors gerecht werden, also auf der untersten Ebene der Steuerung des technischen Prozesses zur Anwendung kommen. Beispiele für typische Einzelfunktionen sind:

1. Das Einschalten einer Pumpe
2. Das Öffnen/Schließen eines Ventils
3. Das Aktivschalten eines Reglers

2.1.2 Prozessstabilisierungsebene

In der Prozessstabilisierungsebene werden die grundlegenden Steuerungen (open/feed-back control loops) zur Stabilisierung von Teilprozessen und Steuerungen von sog. Grundfunktionen realisiert. Zum Verständnis der Grundfunktionen ist es wichtig zu wissen, dass häufig die bloße Formulierung einer Herstellvorschrift eines Produktes mit Hilfe der Einzelfunktionen aus der Feldebene nicht genug verfahrensorientiert ist. Deswegen werden von den Anlagenbetreibern stärker am Verfahren orientierte Anweisungen benötigt. Betrachtet man den Aufbau einer Teilanlage, so sieht man, dass in der Regel mehrere technische Einrichtungen, PLT-Stellen und Anlagenteile als Gruppe zusammen ein Ausrüstungsmodul bilden, das eine Grundfunktion bereitstellt. Beispiele hierfür sind:

1. Das Zugeben einer Substanz
2. Das Temperieren einer Flüssigkeit
3. Das Ab- und Umpumpen einer Flüssigkeit
4. Das Inertisieren und Rühren einer Substanz

Zur Bearbeitung solcher Funktionalitäten wird jeweils eine ganze Gruppe von koordininiert arbeitenden Grundfunktionen benötigt. Die Stabilisierungsebene empfängt einzustellende Sollwerte von der Prozesssicherungsebene und versucht durch geeignete Modifikation der Stellgrößen von Aktoren diese Sollwerte als Istwerte der Sensoren zu realisieren. Die Prozessstabilisierungsebene kann Ereignisse generieren, wenn Istwerte zuvor definierte Grenzwerte unter- bzw. überschreiten. Außerdem verfügt sie über Mechanismen zur periodischen Generierung von Ereignissen, um beispielsweise in regelmäßigen Abständen die Istwerte von Sensoren an die Prozesssicherungsebene zu melden.

2.1.3 Prozesssicherungsebene

Die Prozesssicherungsebene überprüft Aktionen und Parameter darauf, ob sie die Sicherheit einer Teilanlage gefährden könnten. Die Prozesssicherungsebene empfängt von der hierarchisch übergeordneten Prozessführungsebene Befehle für Aktionen und verknüpft, bzw. verriegelt Teilaktionen abhängig vom Prozesszustand, um die Sicherheit einer Teilanlage zu gewährleisten. Teilaktionen betreffen die Vorgabe von Soll- und Grenzwerten an die Prozessstabilisierungsebene. Die Komponenten der Prozesssicherungsebene werden konfiguriert, indem sie mit entsprechenden Parametern geladen werden. Die Prozesssicherungsebene liest Istwerte und empfängt Ereignisse von der Prozessstabilisierungsebene. Die von der Prozessstabilisierungsebene empfangenen Ereignisse werden von der Prozesssicherungsebene als gewöhnliche Meldungen, Warnungen oder Alarme klassifiziert und an die Prozessführungsebene weitergereicht. Zusätzlich kann sie selbst Meldungen, Warnungen und Alarme generieren. Es bleibt festzuhalten, dass die Funktionalität der Prozesssicherungsebene mit ihren Verriegelungssteuerungen oft in intelligente Komponenten integriert oder mit der Funktionalität anderer Ebenen zusammengefasst wird. Ein Beispiel für eine Funktion der Prozesssicherungsebene ist eine Verriegelungssteuerung für eine Pumpensteuerung. Diese Verriegelungssteuerung verhindert, dass bei der Pumpe ein Unterdruck, der zu einer Zerstörung der Membran führen kann, entsteht.

2.1.4 Prozessführungsebene

Die wichtigsten Funktionen der Prozessführungsebene sind die Operatorfunktionen *Anzeigen und Bedienen* sowie die Ablaufsteuerung. Bei Störungen und Ausfällen werden Diagnosen und geeignete Ausnahmebehandlungen durchgeführt. Die Prozessführungsebene empfängt Meldungen, Warnungen und Alarme von der Prozesssicherungsebene und fragt den Prozesszustand ab. Sie führt ein (selektives) Zustandsabbild des Prozesses und archiviert Ereignisse und Zustandssequenzen. Von der Prozessführungsebene werden durchzuführende Aktionen und Konfigurationen für Komponenten ausgewählt. Schalttafeln, die den Zustand einer Anlage durch Bilder visualisieren, damit die Anlagenbediener geeignete Maßnahmen einleiten können, sind Beispiele für die Funktionalität dieser Ebene.

2.2 Funktionsbausteine als statische Bestandteile des Architekturmodells

In [EKS92] wird das Konzept des Funktionsbausteins vorgestellt, das die Grundlage für eine softwaretechnische Lösung der Problematik zur Erstellung von Steuerungssoftware bereitstellt.

In [Ens00] wird ein Funktionsbaustein folgendermaßen definiert:

> „Ein Funktionsbaustein ist ein nach Aufgabe und Wirkung abgrenzbares Objekt. Er dient zum einen der Beschreibung leittechnischer Funktionen und zum anderen zu deren Realisierung. Als Beschreibungsmittel dient ein Funktionsbaustein der Beschreibung abgeschlossener Funktionen, die durch den Wirkzusammenhang von Eingangsgrößen, internen Zustandsgrößen, Parametern und Ausgangsgrößen festgelegt wird. Als Realisierungsmittel dient ein Funktionsbaustein als ein Objekt innerhalb eines Leitsystems zur programmiertechnischen Umsetzung einer Funktion.“

Eine Funktionsbaustein-Instanz ist ein konkretes Exemplar eines Funktionsbausteins mit eigener Identität und eigenen Zustandsdaten.

Es ist möglich, sämtliche Prozessführungsfunktionen in Funktionsbausteinen zu kapseln. Die Bausteine besitzen folgende Eigenschaften:

- Der Austausch von Informationen mit anderen Funktionsbausteinen erfolgt nur über definierte Ein- und Ausgangsschnittstellen, die als Konnektoren bezeichnet werden.
- Die Verwaltung von Methoden und Zustandsinformationen wird innerhalb des Bausteines vorgenommen.

- Die Ableitung der Eigenschaften eines Funktionsbausteines ergibt sich aus einem Typkonzept.

Aufgrund dieser Eigenschaften spricht man auch von einem objektorientierten Bausteinkonzept. In dem in [EKS92] vorgestellten Konzept spielt der Typ eine wichtige Rolle, da jeder Funktionsbaustein einen Typ definiert. Damit werden im Typ die Methode, die Ein- und Ausgangskonnektoren und die Datenstruktur eines Funktionsbausteins vollständig festgelegt.

Funktionsbausteine können hierarchisch miteinander verschaltet bzw. verknüpft werden. Für die Funktionseinheiten zur Prozessführung lassen sich – unabhängig von ihren speziellen – eine Reihe gemeinsamer Eigenschaften festlegen. Der Ausgangspunkt zur Auffindung dieser Eigenschaften ist die weitgehend gleiche Arbeitsweise. Sämtliche Prozessführungsfunktionen können nach [BE94] in Bausteinen gekapselt werden.

Jeder Funktionsbaustein besitzt einen bestimmten Betriebszustand. Betriebszustände beschreiben in einer stark verdichteten Form den Zustand, in dem sich eine Funktionseinheit befindet. Weiterhin ist es möglich, für einen Funktionsbaustein geeignete Bearbeitungszustände zu definieren. Funktionsbausteine stellen bestimmte Standardfunktionalitäten bereit:

- Verriegelung
- Einschaltfreigabe
- Freigabe
- Stopp

Für einen allgemeinen Funktionsbaustein lassen sich die folgenden Betriebszustände festlegen:

- Halt: Halt-Schritte sind spezielle Schritte einer Fahrweise, die aufgrund bestimmter Prozessereignisse oder eines Halt-Befehles angefahren werden und nur durch einen Weiter-Befehl wieder verlassen werden können.
- Bereit
- Fertig
- Stopp
- Lauf
- Abgebrochen

Weiterhin existieren Funktionalitäten, die es ermöglichen die folgenden Betriebszustände einzunehmen:

- Starten
- Anhalten
- Abbrechen
- Stoppen
- Fortsetzen

Funktionsbausteine sind zunächst als abstraktes Konzept zu verstehen. In [Ens00] wird auch vom Funktionsbausteinkonzept gesprochen, das in seiner Funktionalität weiter an die Ebene der Produktion, in der es eingesetzt werden soll, angepasst werden muss. In Tabelle 2.1 wird angegeben, durch welche speziellen Funktionsbausteine dies für jede Ebene der Produktion zu geschehen hat.

Im Folgenden soll zunächst geklärt werden, welche speziellen Funktionalitäten Funktionsbausteine zur Prozesssteuerung bzw. zur Prozessstabilisierung besitzen, bevor im Weiteren die Zuordnung von Funktionsbausteinen für die einzelnen Ebenen des oben vorgestellten vier Ebenenmodells aus Abbildung 2.1 unternommen wird.

Ebene	Funktionen	Zuordbarer Funktionsbaustein
Prozessführung	Führen, lenken, leiten, Monitoring von Teilanlagen und Anlagen	Bild eines Anlagenelementes, Teilanlagensteuerung, Anlagensteuerung
Prozesssicherung/ Prozessstabilisierung	Steuern, regeln, stabilisieren, verriegeln	Gruppensteuerung, Verriegelungssteuerung, Regler(Fuzzy, PID)
Feldebene	Aktor- und Sensorfunktionalitäten (Messen)	Einzelgerätesteuerung

Tabelle 2.1: Die Funktionen der einzelnen Ebenen

2.2.1 Spezielle Funktionalitäten von Funktionsbausteinen

Durch Betrachtung von Beispielen kann gezeigt werden, dass jeweils eine verhältnismäßig geringe Anzahl von anwendungsspezifischen Funktionsbausteinen ausreicht, um alle in bestimmten Gebieten der Automatisierungstechnik auftretenden Programme auf abstraktem Niveau zu formulieren, wie sie etwa im Bereich der Prozesssteuerung und Prozessstabilisierung Verwendung finden.

Technische Komitees der Gesellschaft für Mess- und Automatisierungstechnik (VDI/VDE-GMA Fachausschuss 5.3 "Herstellerneutrale Konfigurierung von Prozessleitsystemen") und der chemischen Industrie (NAMUR Arbeitsgruppe 1.2 "Software-Technik und systemunabhängige Konfigurierung") haben einen Satz von 67 anwendungsspezifischen Funktionsbausteinen identifiziert, der geeignet ist, den Großteil der in der chemischen Verfahrenstechnik auftretenden Automatisierungssprachen in der graphischen FUP-Sprache nach IEC 61131-3 zu formulieren [WA97]. Die folgende Liste der in der Richtlinie VDI/VDE 3696 definierten Funktionsmodule vermittelt einen Eindruck typischer Funktionalitäten:

1. *monadische mathematische Funktionen.* Dies sind etwa abs, cos, sin exp, log, nichtlineare statische Interpolationsfunktion
2. *Polyadische mathematische Funktionen, z. B. +, -, *, :*
3. *Vergleiche* ($>$, $<$, $=$, $\leq$, $\geq$ ...)
4. *Boolesche Funktionen* ($\wedge$, $\vee$, $\neg$)
5. *Auswahlfunktionen* (Min, Max, Multiplexer)
6. *Technologische Rechenfunktionen* (Durchflussmengenkorrektur, Wärmeflussberechnung, Flüssigkeitsberechnung, Flüssigkeitstandsberechnung, Berechnung gesättigten Dampfdrucks, Berechnung gesättigter Dampftemperatur)
7. *Zähler, Zeitgeber, mono- und bistabile Elemente* (Auf- und Abwärtszähler, Flusszähler, Ausschaltverzögerung, Einschaltverzögerung, nicht retriggerbare monostabile Kippstufe, bistabile Kippstufe mit Rücksetzdominanz, bistabile Kippstufe mit Setzdominanz, Erkennung fallender Flanken, Erkennung ansteigender Flanken, Impulsdauermodulator)
8. *Prozesseingabe/-ausgabe* (Analogeingabe, Binäreingabe, Impulseingabe, digitale Worteingabe, Analogausgabe, Binärausgabe, digitale Wortausgabe)
9. *Dynamische Elemente und Regler* (Vorsteuerung, Differenzierung mit Verzögerung, Integrator, Totzeit, Universalregler (PID), Standardregler (PID) 1. sowie 2. Ordnung

Weiterhin gibt es sog. *Notabschaltsysteme*, die in [HK99] folgendermaßen definiert sind:

> „Ein System, das einen Prozess überwacht und nur handelt – d. h. den Prozess in einen statisch sicheren Zustand überführt (normalerweise eine Abschaltung) – wenn die Sicherheit von Menschen, Umwelt oder Investitionen gefährdet ist."

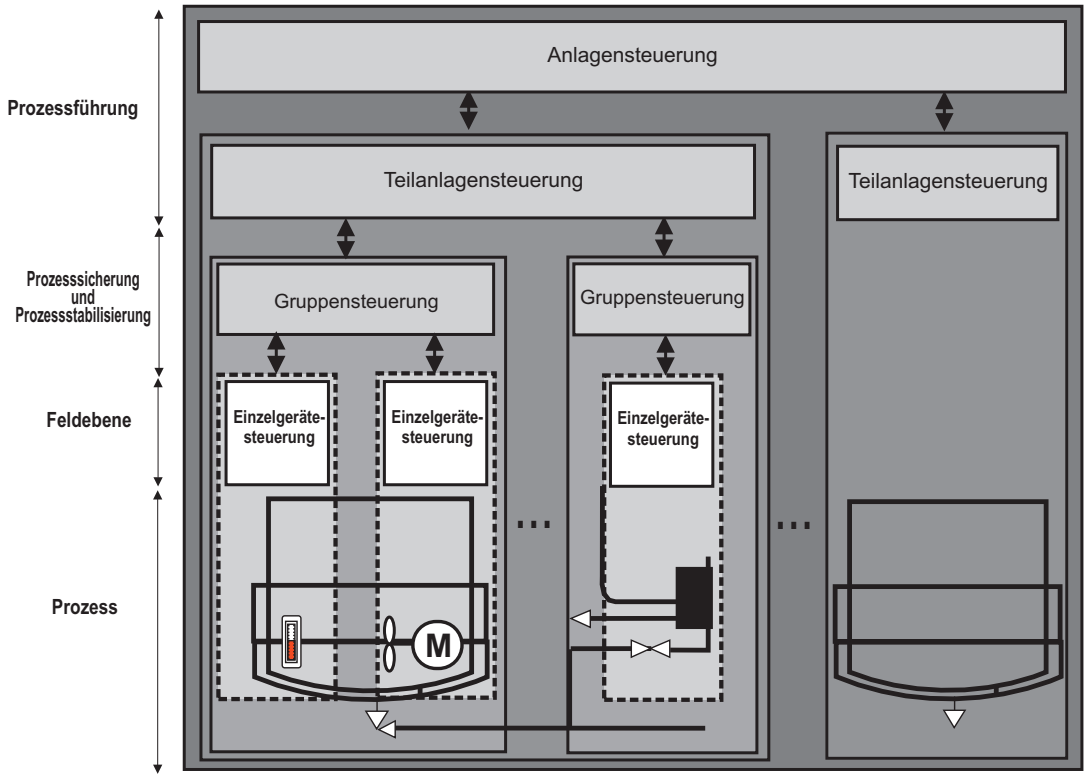


Abbildung 2.2: Die Architekturbestandteile der einzelnen Prozessebenen

Ein solches System beobachtet, ob bestimmte physikalische Größen, wie Temperaturen oder Drücke, innerhalb vorgegebener Grenzen bleiben. Werden diese Grenzen verletzt, dann werden sog. Notabschaltaktionen, wie das Öffnen oder Schließen von Ventilen oder Abschalten von Motoren, vorgenommen.

2.2.2 Steuerungssoftware für die Feldebene

Die Software für Funktionseinheiten, die Einzelfunktionen realisieren, welche einem Aktor bzw. Sensor fest zugeordnet sind, wird durch Funktionsbausteine, die als Einzelfunktionssteuerungen bzw. Einzelgerätesteuerungen (s. Abbildung 2.2) bezeichnet werden, implementiert. Eine spezielle gerätetechnische Realisierung und die anlagentechnischen Verriegelungen müssen gegenüber einer überlagerten Steuerung verborgen bleiben.

2.2.3 Steuerungssoftware für die Prozessstabilisierungs- und Prozesssicherungsebene

Für die Prozessstabilisierungsebene sind zunächst Funktionsbausteine mit Reglerfunktionalitäten von Bedeutung. Diese Funktionsbausteine existieren in den Ausprägungen PID-Regler und Fuzzy-Regler.

Grundfunktionen, die in der Prozessstabilisierungsebene auftreten, werden durch sog. Gruppensteuerungen (s. Abbildung 2.2) realisiert. Gruppensteuerungen können Verriegelungen umfassen, welche der Prozesssicherungsebene zuzuordnen sind. Gruppensteuerungen können auch Verriegelungen besitzen und übernehmen damit auch die Funktionalität der Prozesssicherungsebene. Gruppensteuerungen können hierarchisch angeordnet werden.

2.2.4 Steuerungssoftware für die Prozessführungsebene

Eine verfahrenstechnische Anlage ist nach [Pol94] eine Zusammenfassung mehrerer Teilanlagen zu einer abgegrenzten technischen Einheit. Eine Teilanlage ist in Anlehnung an DIN 28004 Bestandteil einer Anlage, die zumindest teilweise selbstständig betrieben werden kann und somit eine eigenständige Funktionseinheit zur Realisierung einer technischen Anlage ist. Dies bedeutet, dass Teilanlagen einen oder mehrere Verfahrensabschnitte selbstständig durchführen können. Zur Lösung einer verfahrenstechnischen Aufgabe müssen die verschiedenen Grundfunktionen koordiniert und mit entsprechenden Grundfunktionen der Prozessstabilisierungsebene versorgt werden. Wie dies zu geschehen hat, ist Bestandteil einer auf Grundfunktionen aufbauenden Herstellvorschrift. Die Abwicklung der Herstellvorschrift, die Überwachung und die Koordination der Grundfunktionen einer Teilanlage übernimmt die sog. Teilanlagensteuerung (s. Abbildung 2.2). Dafür gibt es zwei Möglichkeiten:

1. Die Teilanlage kann durch ihre anlagentechnische Ausstattung nur eine einzige Herstellvorschrift oder einen bestimmten, vorgegebenen Satz von Herstellvorschriften abwickeln. In diesem Fall können diese Herstellvorschriften ohne Beschränkung der verfahrenstechnischen Flexibilität der Anlage als fest vorgegebene Fähigkeiten zugeordnet werden. Typische Beispiele sind der Stromtrockner oder der Vakuumbandfilter.

2. In einer Teilanlage können eine große Anzahl unterschiedlicher Herstellvorschriften abgewickelt werden. Die Teilanlage kann nicht alle auf ihr abwickelbaren Herstellvorschriften kennen, oder es werden von Zeit zu Zeit neue Herstellvorschriften hinzugefügt. In solchen Fällen muss die Herstellvorschrift für jedes Produkt bereitgestellt werden. Eine typische Teilanlage dieses Typs ist der Rührkesselreaktor, in dem verschiedene Produkte hergestellt werden.

Die Teilanlagensteuerung der einzelnen Anlagen ist im Sinne der Führung der Anlagen insgesamt zu koordinieren. Dazu dient die hierarchisch überlagerte Teilanlagensteuerung. Insgesamt erhält man auf diese Weise, ausgehend von den einzelnen Aktoren einer Teilanlage, eine an der verfahrenstechnischen Aufgabe orientierte hierarchische Strukturierung der Prozessführungsaufgabe sowie der zugehörigen Funktionseinheiten.

Analog zu Teilanlagensteuerungen sind Anlagensteuerungen für die Steuerung von Anlagen auf der obersten Ebene der Prozessführung verantwortlich.

Funktionseinheiten zur Prozessführung werden über in ihrem Aufbau einheitliche Bilder beobachtet und bedient. In den Bildern müssen die unterschiedlichen Informationen über die Funktionsbausteine dargestellt werden. Dazu dienen Standardbilder mit der angewählten Fahrweise, den Führungswerten, dem Betriebszustand, dem Befehlsmodus und dem Bearbeitungszustand. Aus solchen Bildern heraus wird auch die Handbedienung einer Teilanlage oder einer Anlage vorgenommen. In weiteren Bildern werden entsprechende Informationen der zu einer übergeordneten Funktionseinheit gehörenden, untergeordneten Funktionseinheiten listenförmig zusammengefasst. Aus solchen Bildern heraus lassen sich die untergeordneten Einheiten freigeben oder in Automatik übernehmen. Zur Unterstützung der Fehlerdiagnose dienen weitere Bilder, in denen der Zustand von Signalen, Verriegelungen oder Schrittbedingungen angezeigt wird.

2.3 Dynamische Bestandteile der Architektur

In diesem Abschnitt sollen die dynamischen Bestandteile des vorgestellten Architekturmodells beschrieben werden. Die dynamischen Bestandteile zeigen, wie Bestandteile der statischen Architektur miteinander zusammenwirken. Funktionsbausteine erhalten Befehle, die deren Arbeitsweise lenken. Alle Befehle an einen Funktionsbaustein können entweder vom Anlagenfahrer oder von der übergeordneten Funktionseinheit gegeben werden. Zu jedem Typ eines Funktionsbausteins gehört ein spezieller Befehlssatz zur Steuerung des Ablaufs bei der Befehlsverarbeitung. Eine Reihe von Befehlen lassen sich für alle Funktionsbausteine standardisieren. Dies sind insbesondere Befehle zur Schaltung der Betriebszustände. Typische Betriebszustände sind:

- Ein: Das Einschalten der angewählten Fahrweise,

- Aus: Das Abfahren der angewählten Fahrweise und das Einnehmen des Grundzustandes,
- Halt: Das Anfahren des zum aktuellen Schritt gehörenden Halt-Schrittes,
- Weiter: Das Weiterfahren nach einem Halt,
- Außer Betrieb: Das Außer-Betrieb-Setzen der Funktionseinheit,
- In Betrieb: Das In-Betrieb-Setzen der Funktionseinheit.

Weiterhin gibt es Befehle zur Auswahl einer Fahrweise und zum Setzen von Führungswerten. Eine Funktionseinheit zur Prozessführung besitzt außerdem bestimmte Befehlsmodi:

- Hand: Der Anlagenfahrer ist Befehlsgeber, das heißt, die Funktionseinheit nimmt Befehle nur vom Anlagenfahrer an.
- Frei: Es ist kein Befehlsgeber definiert, das heißt, es werden weder vom Anlagenfahrer noch von der übergeordneten Funktionseinheit Befehle angenommen. Bestehende Aufträge werden jedoch weiter bearbeitet. Der Anlagenfahrer darf die Funktionseinheit in den Modus Hand schalten, die übergeordnete Funktionseinheit ist der Befehlsgeber.
- Die übergeordnete Funktionseinheit ist Befehlsgeber, das heißt, Befehle werden nur von der übergeordneten Funktionseinheit angenommen.

Ein Befehl startet, modifiziert, unterbricht oder beendet in einem empfangenden Funktionsbaustein einen Auftrag. Zu jedem Typ einer Funktionseinheit gehört ein spezieller Befehlssatz zur Initiierung und Steuerung des Ablaufes von Aufträgen. Ein Auftrag besteht aus der einzustellenden Fahrweise und den zugehörigen Führungswerten. Fahrweisen sind alternativ einstellbare Funktionsabläufe einer Funktionseinheit. Zu jeder Fahrweise gehört ein fester Satz von Führungswerten, die vorgegebenen Stell- bzw. Sollwerten entsprechen und für die Steuerung von Funktionsbausteinen und Funktionseinheiten benötigt werden. Solche Führungswerte besitzen bestimmte physikalische Einheiten. In der Regel werden Führungswerte durch den Anlagenfahrer oder die übergeordnete Funktionseinheit auftragsbezogen gesetzt. Es ist jedoch auch möglich, einer Funktionseinheit per Befehl mitzuteilen, dass für den auszuführenden Auftrag Führungswerte über gesonderte Eingänge der Funktionseinheit gesetzt werden.

Bei einer genaueren Betrachtung der verfahrensorientierten Abläufe stellt man schnell fest, dass innerhalb eines Ablaufs durchaus unterschiedliche Vorgehensweisen realisiert werden müssen. Innerhalb einer Fahrweise kann jede Grundfunktion durch Bedienung oder als Reaktion auf Ereignisse (z. B. Störungen) verschiedene Betriebszustände einnehmen. Diese Betriebszustände müssen beispielsweise beim Entwurf des zugehörigen SFC berücksichtigt werden. Einzelne alternativ wählbare Funktionen einer Funktionseinheit werden als Fahrweisen einer Grundfunktion bezeichnet. Die gesamte Funktionalität einer Teilanlage ist durch die Fahrweisen aller ihrer Grundfunktionen gegeben.

In einem hierarchischen Prozessleitsystem erteilen übergeordnete Funktionsbaustein-Instanzen untergeordneten Funktionsbaustein-Instanzen Befehle. Die untergeordneten Funktionsbaustein-Instanzen führen diese Befehle in eigener Regie aus und überwachen selbst die ordnungsgemäße Ausführung. Sie zeigen der übergeordneten Funktionsbaustein-Instanz den Status der Ausführung und alle zur Beurteilung ihrer Arbeitsweise wichtigen Zustände an. Eine bemerkenswerte Eigenschaft des Modells ist es, dass die Befehle konsequent nachrichtenorientiert ausgegeben werden. Ein Befehl, der von der übergeordneten Funktionsbaustein-Instanz an die untergeordnete Funktionsbaustein-Instanz geschickt wird, kann von dieser akzeptiert oder abgewiesen werden. Die übergeordnete Funktionsbaustein-Instanz erhält eine Quittung über Akzeptanz oder Nichtakzeptanz des Befehls. Danach wird der Befehl von der untergeordneten Einheit technologisch abgewickelt. In diesem vereinfachten Modell erhält der übergeordnete Baustein keine explizite Nachricht über den Stand der Abwicklung und die Beendigung des Auftrages, der von dem Befehl ausgelöst wurde. Der übergeordnete Baustein muss sich bei Bedarf durch eine Statusabfrage über den Stand der Verarbeitung informieren.

Wie bereits in den einzelnen Unterebenen der Prozessleitung definiert, können Störungen bzw. Warnungen auftreten. Funktionseinheiten zur Prozessführung überprüfen Befehle auf ihre Zuverlässigkeit, Eingangswerte auf einzuhaltende Grenzwerte, das Ansprechen von Verriegelungsbedingungen und den bestimmungsgemäßen Verlauf, der von ihnen einzustellenden Prozesszustände. Treten Abweichungen auf, dann setzen die Funktionseinheiten entsprechende Meldungen bzw. Warnungen ab und informieren die zugehörige übergeordnete Funktionseinheit über die aufgetretene Störung. Eine wichtige Störung liegt vor, wenn die untergeordnete Funktionseinheit ihre Aufträge nicht mehr ordnungsgemäß abwickeln kann. Gründe dafür können sein:

- Die Funktionseinheit ist auf Handbetrieb eingestellt.
- Die Funktionseinheit ist gestört.
- Die Funktionseinheit ist außer Betrieb.

Die von Funktionsbausteinen bzw. Reglern generierten Meldungen bzw. Warnungen werden an Funktionsbausteine oder an sog. Bilder einer Anlage, die sich z. B. im Kontrollraum befinden, weitergeleitet.

2.3.1 Beispiele für die praktische Eignung des Architekturmodells

Bei der Vorstellung einer Architektur bleibt natürlich die Frage offen, ob sie den Bedürfnissen und Anforderungen praktischer Steuerungssysteme gerecht wird. Die Frage der Eignung soll hier kurz erörtert werden.

In [ME99] wird über die Anwendung von standardisierten Funktionsbausteinen und der Bedienoberflächen von Leitsystemen mit einem Toolkit berichtet. Der Artikel beschreibt ein Projekt, in dem die folgenden Funktionalitäten als grundlegend (Grundfunktionselemente) angesehen werden:

- Die Ansteuerung eines einfachen Motors ist möglich.
- Die Ansteuerung muss mit zwei Drehzahlen und in zwei Drehrichtungen erfolgen können.
- Das Ansteuern eines Frequenzumrichters, der eingehende Frequenzen in Drehzahlen eines Motors umrichtet, wird unterstützt.
- Die Funktionalität zur Ansteuerung eines Auf-/Zu-Ventils (Aktoransteuerung, wiederum Einzelgerätesteuerung) wird angeboten.
- Die Ansteuerung eines Regelventils (Aktor einer Einzelgerätesteuerung) muss möglich sein.
- Das sensornahe Einlesen und Verarbeiten (Diskretisieren eines Analogwertes) wird angeboten.
- Die Regler (z. B. PID oder Fuzzy) werden unterstützt.
- Die Dosierungsfunktionalitäten werden durch spezielle Einzelgerätesteuerungen von Aktoren unterstützt.

Alle diese Funktionsbausteine verfügen über folgende Eigenschaften:

- Die Ansteuerung der entsprechenden Aktoren wird vorgenommen.
- Die Auswertung aller Rückmeldungen wird vorgenommen.
- Die Auswertung von Leittechnikstörungen (wie etwa Drahtbruch, Baugruppenfehler etc.) wird durchgeführt.
- Die Verknüpfungseingänge (Verriegelung, Freigabe und weitere Funktionen) werden ausgewertet.

- Eine Schnittstelle zu einer überlagerten Grundfunktion und die dafür notwendige Verwaltung des Zugriffs per Hand oder Automatik werden unterstützt.
- Die Bedienung der Motorbausteine erfolgt vor Ort und die Sicherheitsschalter der Motorbausteine werden ausgewertet.

In [EH99] wird der Einsatz eines Prozessleitsystems mit Grundfunktionsfahrweisen in einer Batch-Anwendung vorgestellt, die die Anwendung einer ähnlichen Steuerung bei der Bayer AG beschreibt. Anzumerken bleibt, dass die einzelnen Übergangsstatuskennungen als parallele Ketten in einem SFC-Plan realisiert werden. Unterschiedliche Fahrweisen einer Grundfunktion, wie z. B. die Dosierung von einem Produkt A mit einem Produkt B, werden in unterschiedlichen SFC-Plänen projektiert. Fahrweisen des gleichen Typs sollen in verschiedene Projekte hineinkopiert werden. In einer Architektur, wie sie hier vorgestellt worden ist, lassen sich bestimmte Mechanismen zur wiederkehrenden Lösung von Problemen, die aus Anforderungen resultieren, identifizieren. Derartige Problemlösungen werden als Muster bezeichnet, die im folgenden Kapitel genauer betrachtet werden sollen. Von speziellem Interesse werden im weiteren Verlauf Muster für Steuerungssoftware sein [Rie00, Mey00].

2.4 OPC Unified Architecture

Die OPC Unified Architecture (OPC UA) ist die neuste aller Spezifikationen der OPC Foundation. Die erste Version der OPC UA wurde 2006 nach einigen Jahren Spezifikationsarbeit verabschiedet. Anfang 2009 wurde eine stark überarbeitete Version, an der ca. 30 Unternehmen mitgewirkt haben [MLD09], veröffentlicht.

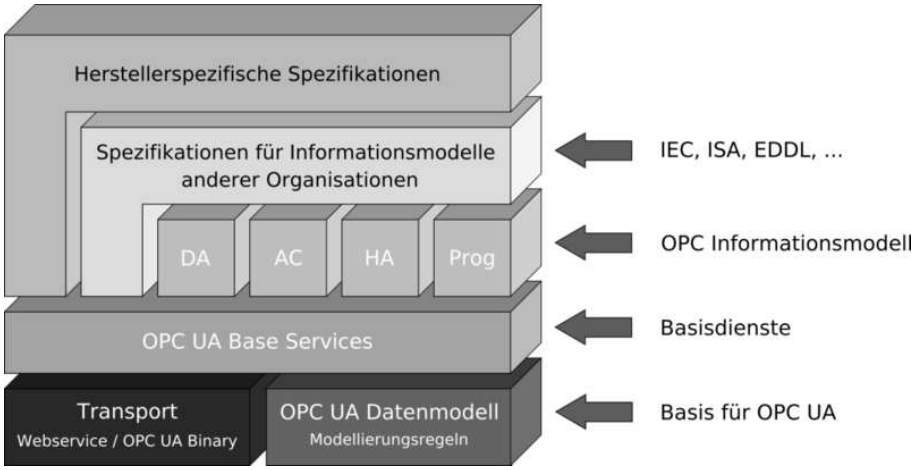


Abbildung 2.3: Die Struktur der OPC UA (entnommen aus Wikipedia)

Im Folgenden werden einige interessante Eigenschaften der OPC UA kurz vorgestellt. Die Skalierbarkeit von Embedded-Controllern bis zu Mainframes wird unterstützt. Der Kommunikationsstack der OPC UA beruht auf einer portablen ANSI-C-Implementierung. Es ist möglich den Stack sowohl für den Multithreaded-Betrieb als auch für den Singlethreaded/Singletask-Betrieb zu kompilieren, was für die Portierung in eingebetteten Geräten von Bedeutung ist. Es wird eine eigene Security-Implementierung, die auf den neuesten Standards beruht, bereitgestellt. Für jeden Service gibt es konfigurierbare Timeouts. Die effiziente Übertragung von großen Datenpaketen wird unterstützt.

Der Kommunikationsstack ist eine von vielen Neuerungen. Die Architektur der OPC UA ist eine Service-orientierte Architektur (SOA) und in mehrere logische Ebenen aufgeteilt (s. Abbildung 2.3). Sämtliche von der OPC UA definierten Base-Services sind abstrakte Methodenbeschreibungen, die unabhängig vom Protokoll sind. Sie bilden die Basis für die gesamte Funktionalität der OPC

UA. Die Transportschicht setzt diese Methoden in ein Protokoll um, d. h. sie serialisiert bzw. deserialisiert die Daten und versendet diese über das Netz. Gegenwärtig sind zwei verschiedene Protokolle dafür vorgesehen. Zunächst gibt es ein auf Webservices basierendes Protokoll. Zusätzlich existiert ein binäres, hoch optimiertes und performantes TCP-Protokoll. Weitere Protokolle können bei Bedarf hinzugefügt werden. Damit wird die Kommunikation zwischen einem OPC-UA-Client mit einem OPC-UA-Server unterstützt.

Die Spezifikation der OPC UA besitzt die folgenden Bestandteile:

- Konzepte (Concepts)
- Sicherheitsmodell (Security Model)
- Adressraum-Modell (Address Space Model)
- Dienste (Service)
- Informationsmodell (Information Model)
- Datenzugriff (Data access) und historisierter Datenzugriff (Historical Data Access)
- Alarme und Bedingungen (Alarms and Conditions)
- Programme (Programs)

Das OPC-Informationsmodell ist ein vollvermaschtes Netzwerk aus Knoten (Nodes), das es ermöglicht, alle Arten von Metainformationen und Diagnoseinformationen zu übertragen. Ein Knoten ist mit einem Objekt in der objektorientierten Programmierung vergleichbar. Ein Knoten besitzt in der Regel Attribute, die gelesen werden können (Data Access (DA)), Operationen, die aufgerufen werden können, und Events, die verschickt werden können. Solche Knoten stellen sowohl die Nutzdaten als auch alle anderen Arten von Metadaten zur Verfügung. Der auf diese Weise modellierte Adressraum beinhaltet somit auch ein Typmodell, das sämtliche Datentypen beschreibt. Ein Programm beschreibt eine komplexe zustandsbehaftete Funktionalität. Das Informationsmodell bildet die Grundlage für die Spezifikation der übrigen Modelle. Um Informationsmodelle sowie Programme zu spezifizieren, wird UML verwendet. Dafür wird auf einige der Diagramme aus Kapitel 3 zurückgegriffen. Selbstverständlich ist auch die OPC UA selbst mittels der UML modelliert. Das Adressraum-Modell beschreibt, welche Objekte im Adressraum eines OPC-UA-Servers existieren.

OPC-UA-Spezifikationen sind keine reinen Anwenderspezifikationen. Vielmehr wird ein generischer Rahmen für Anwenderspezifikationen bereitgestellt. Umfangreiche Bestandteile beschreiben UA-Interna, die vom Kommunikationsstack behandelt werden. Diese sind von besonderem Interesse für die Entwicklung eines eigenen UA-Stacks oder dessen Portierung. Für die Anwendungsentwicklung, die OPC UA verwendet, greift man auf die OPC-UA-API zurück. Interessant für Anwendungsentwickler sind jedoch speziell das Adressraum-Modell, die Dienste und das Informationsmodell. Für eine eigenständige Anwendung ist es erforderlich, ein neues Informationsmodell zu entwerfen.

Andere bedeutende Standards, wie MIMOSA (Machinery Information Management Open Systems Alliance) und IEC 61131, bauen auf der OPC UA auf oder werden gegenwärtig an diese angepasst. Mit dem OPC UA Information Model für IEC 61131-3 [Tec09] der Version 0.09.1 liegt eine frühe Fassung für einen sehr wichtigen Standard im Bereich der Automatisierungstechnik vor, der OPC UA und damit auch UML verwendet.

2.5 Beispielanlage zur Entwicklung von Steuerungssoftware

In diesem Abschnitt wird eine Anlage zur Produktion von Grundstoffen von Gummibärchen – nämlich der Gelatine – als Beispiel vorgestellt. Die Software, die zur Steuerung dieser Anlage verwendet wird, wird den Leser durch die gesamte Arbeit begleiten.

Gelatine, der wichtigste Grundstoff für die Herstellung von Gummibärchen ist ein hochreines Protein, das durch die partielle Hydrolyse von kollagenem Eiweiß hergestellt wird. Ausgangsmaterial zur Produktion von Gelatine ist die dicke Haut des Schweins, die sog. Schwarte, oder Rinderhaut[2]. Aus diesen Tierbestandteilen wird das Eiweiß herausgelöst und zu Gelatine verarbeitet. Der chemische Prozess zur Erzeugung der Gelatine ist mehrstufig.

Zunächst erfahren die oben genannten Rohstoffe eine spezielle Vorbehandlung mit Säuren oder Laugen. Dadurch wird das Eiweiß aufgeschlossen, bevor es aus dem Rohmaterial herausgelöst wird. Die so entstandene Lösung wird nun mit Warmwasser versetzt und durchläuft einen mehrstufigen Extraktionsprozess, in dem Gelatine mit unterschiedlicher Gelierfestigkeit gewonnen wird. Die ersten Abzüge, die bei niedrigsten Temperaturen gewonnen werden, ergeben höchste Gelierfestigkeit der dabei ausgeschmolzenen Gelatine. Diese fällt etwa in einer fünfprozentigen Lösung an. Anschließend an den ersten Extraktionsvorgang wird die verbliebene teilextrahierte Lösung mit frischem Warmwasser höherer Temperatur angesetzt und nochmals extrahiert. Dies wird so oft fortgesetzt, bis der letzte Rest der Gelatine bei Siedetemperaturen in Lösung gegangen ist. Das gesamte Verfahren besteht aus den Verfahrensschritten des Befüllens, des Erhitzens und des Abpumpens, die sequentiell nacheinander abfolgen. Natürlich ist es möglich, dass die drei Verfahrensschritte mehrfach hintereinander stattfinden.

Die letzten Verarbeitungsschritte bei der Gelatineproduktion sind das Reinigen, das Erhitzen und das Trocknen. Hierbei werden Fettspuren, Kollagenfasern und Salzreste aus der Gelatinelösung entfernt. Die gereinigten Lösungen jeder Gelierfestigkeit werden bei Temperaturen bis zu 140 Grad Celsius sterilisiert und zum Erstarren gebracht. Dieses Gel wird getrocknet und grob gemahlen. Damit ist der Gewinnungsprozess abgeschlossen.

In der linken Hälfte der Abbildung 2.4 ist der Aufbau unserer Anlage zur Herstellung von Gelatine auszugsweise angegeben. Eine der Teilanlagen dient der beschriebenen Extraktion der Gelatine aus den Rohstoffen. Dazu verfügt die Teilanlage über mehrere Behälter. Wir betrachten nur einen Behälter *B1*, der für eine Extraktion der Gelatine verwendet wird. Der Bediener kann von seinen Steuerpulten die Produktion beeinflussen und erhält über die Bilder Informationen bzgl. des Verarbeitungszustandes der Anlage, z. B. über die Temperatur, die im Kessel zur Teilextraktion herrscht.

Die Teilanlagensteuerung unterstützt sowohl den Hand- als auch den Automatikbetrieb. Der Extraktionsvorgang kann korrekt fertiggestellt, durch Fehler abgebrochen und angehalten werden. Die Teilanlagensteuerung kooperiert ihrerseits mit mehreren Gruppensteuerungen und einem Regler. Die erste Gruppensteuerung steuert das Befüllen des Behälters, die zweite das Abpumpen des Gelatinegels mit der Membranpumpe *P1*. Im Feld befinden sich zahlreiche Sensoren und Aktoren, deren Software durch die Gruppensteuerungen und den Regler kontrolliert wird. Bei periodischen Regelungen sind der Regler, die Einzelgerätesteuerung des Aktors und die Einzelgerätesteuerung des Sensors räumlich voneinander entfernt. Die Sensoren und Aktoren greifen direkt in den Prozess ein, welcher die Regelstrecke enthält, während der Regler auf einer entfernten Hardware-Komponente installiert ist. Eine ähnliche Anlage, bei der Reaktionszeiten allerdings für den Ausfall der Kühlung der Anlage von Bedeutung sind, ist in [HGK98] beschrieben. Der Behälter *B1* muss permanent erwärmt werden, um die Temperatur stabil zu halten. Dazu wird erhitztes Wasser, das durch das Ventil *V2* geleitet wird, verwendet. Durch einen Motor wird das Ventil geöffnet bzw. geschlossen. Die Kombination aus Ventil *V2* und Motor *M1* ist ein Aktor. Der Sensor *TIS* – im Folgenden „Sensor "genannt – ermittelt permanent die Temperatur der Flüssigkeit im Behälter *B1*. Ein PID-Regler ist vorgesehen, um die Temperatur von *B1* in einem bestimmten Bereich zu halten. Mit der Membranpumpe *P1* wird die extrahierte Lösung aus dem Behälter gepumpt. Diese Membranpumpe darf nur in bestimmten Zuständen angefahren werden, um zu verhindern, dass beim Abkühlen der Membran ein Unterdruck entsteht, der zur Perforation bzw. Überdehnung führen kann. Um dies auszuschließen, wird eine Membransteuerung verwendet, die die Membran positioniert. Sollte dennoch ein Anfahren der Membranpumpe erfolgen, wird eine Verriegelung der Membranpumpe durch eine Verriegelungssteuerung vorgenommen.

An die Steuerungssoftware der Anlage bestehen nicht-funktionale Anforderungen bzgl. der Verteilung, Realzeitfähigkeit und Fehlertoleranz. Zunächst werden die Verteilungsanforderungen be-

[2]Auch die Knochen von Schweinen und Rindern sind geeignet.

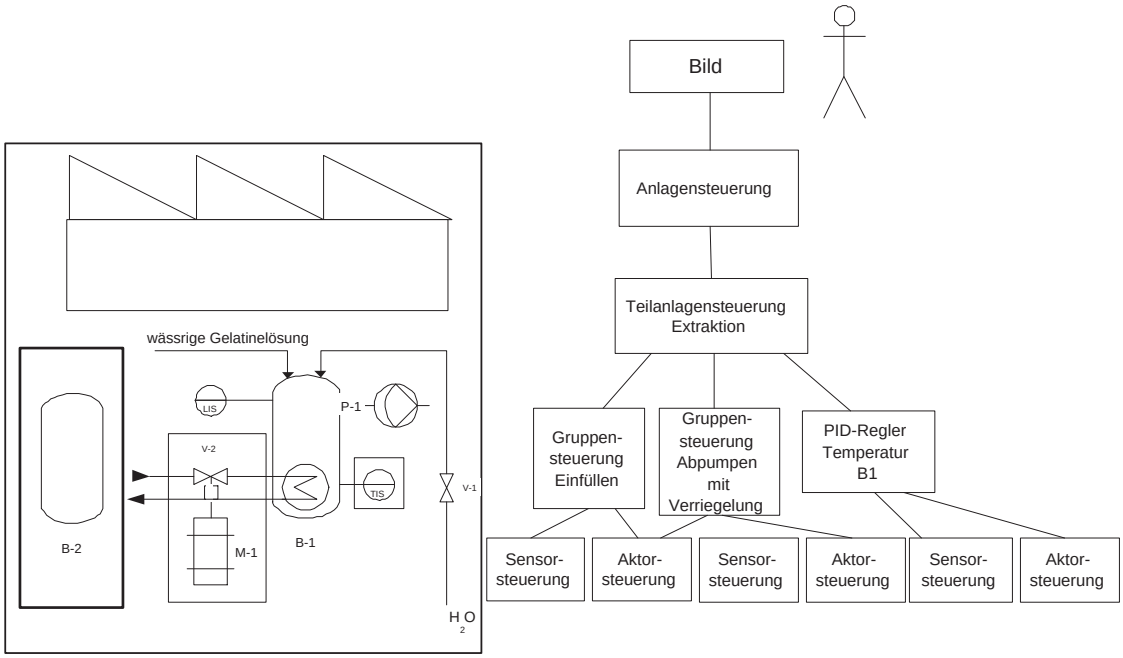


Abbildung 2.4: Die Anlage des Beispiels mit der zugehörigen Steuerungssoftware

handelt. Die Gruppensteuerung zum Einfüllen ist von der Sensorsteuerung S_1 räumlich entfernt. Außerdem befinden sich die Gruppensteueuerung zum Abpumpen und ihre Aktorsteuerung A_1 an verschiedenen Positionen. Auch die Sensorsteuerung S_2 und die Aktorsteuerung A_2 sind nicht direkt bei dem Regler für die Temperatur positioniert, sodass für die Kommunikation eine räumliche Distanz überbrückt werden muss. Der Austausch von Informationen und Anweisungen zwischen dem Bild und den einzelnen Steuerungen muss transaktional erfolgen. Dies ist erforderlich, da Anweisungen durch den Bediener der Anlage entweder ganz oder gar nicht ausgeführt werden müssen, um zu vermeiden, dass unvollständige Steuerungsbefehle zu sinnlosem Verhalten der Anlage führen können. Außerdem ist dies auch für die Protokollierung der Anweisungen in einer Datenbank relevant.

An alle Einzelgeräte-, Gruppen- und die Teilanlagensteuerungen bestehen Realzeitanforderungen, da Ergebnisse in vorgegeben Zeiten verarbeitet und Berechnungen in genau definierten Zeiträumen abgeschlossen werden müssen.

Um sicherzustellen, dass die Membran der Pumpe keinesfalls beschädigt wird, muss die Verriegelungssteuerung, die mit der Gruppensteuerung zum Abpumpen zusammenwirkt, fehlertolerant ausgelegt werden. Das bedeutet, dass ein Ausfall der Gruppensteuerung und der Verriegelungssteuerung mit hoher Wahrscheinlichkeit verhindert werden muss.

Kapitel 3

Unified Modeling Language (UML)

Die UML (Unified Modeling Language) ist eine standardisierte Modellierungssprache, deren Syntax und Semantik durch ein Metamodell definiert ist. Die UML liegt gegenwärtig in der standardisierten Version 2.2 vor. Diese Arbeit bezieht sich auf die Version 2.0 [OMG03]. Als Modellierungssprache stellt UML 2.0 umfangreiche Konstrukte zur Spezifikation von Software bereit. Die Ausführung von UML 2.0 Modellen wird unterstützt, indem *Actions* eingeführt werden. Auch in xUML (executable UML), das ein UML-Profil für ausführbare UML Modelle definiert [MB02, MTAL98, MTAL99], steht die Ausführbarkeit der Modelle im Vordergrund. Ein Profil ist ein Metamodell für eine bestimmte Domäne und gibt an, wie Elemente der UML für einen bestimmten Zweck kombiniert werden. UML 2.0 Modelle sind gut geeignet, um automatisch Quellcode aus ihnen zu generieren. Weiterhin wird die Modellierung von Realzeitsystemen und eingebetteten Systemen besser unterstützt. Zunächst wird kurz die Semantik eines objektorientierten Softwaresystems behandelt. Anschließend werden grundlegende Begriffe vorgestellt und an einem Beispiel verdeutlicht. Zuletzt wird auf die einzelnen Diagramme der UML eingegangen.

3.1 Grundlegende Semantik objektorientierter Software

In einem objektorientierten System interagieren Objekte, um Daten zu verarbeiten. Jedes Objekt hat eine eindeutige Identität. Objekte werden zur Laufzeit erzeugt und zerstört. Ein Objekt besitzt Instanzvariablen. Den Instanzvariablen werden zur Laufzeit Werte zugewiesen. Objektvariablen sind bestimmte Instanzvariablen, deren Werte Referenzen auf andere Objekte repräsentieren. Objekte rufen zur Laufzeit Operationen auf sich selbst oder auf anderen Objekten auf, um bestimmte Dienste in Anspruch zu nehmen. Der Begriff der Operation abstrahiert von einer konkreten Implementierung. Ist die Implementierung einer Operation gemeint, so wird stattdessen von einer Methode gesprochen. In Operationen werden die Werte von Instanz- und Objektvariablen modifiziert bzw. gelesen, Berechnungen durchgeführt, Operationen auf anderen Objekten aufgerufen und Objekte zerstört bzw. erzeugt. Der Aufruf von Operationen geschieht durch den Versand von Nachrichten. Eine Nachricht besteht aus dem Namen der aufzurufenden Operation und Argumentwerten. Nachrichten ermöglichen die Kommunikation von verschiedenen Objekten, transportieren Argumentwerte von einem Objekt zum anderen und liefern bei Beendigung der aufgerufenen Operation Rückgabewerte an das aufrufende Objekt zurück.

Klassen beschreiben schablonenhaft die Struktur und das Verhalten von Objekten, die Instanzen dieser Klasse sind. Eine Klasse besitzt einen eindeutigen Klassennamen. Klassen besitzen Attribute mit einem Namen und einem Typ, die die Instanzvariablen eines Objektes beschreiben. Weiterhin besitzen Klassen Operationen, die die Signatur und das Verhalten der Dienste eines Objektes beschreiben. Die Signatur einer Operation besteht aus den Operationsnamen, den Argumenten und dem Rückgabetyp. Jedes Argument besitzt einen Namen und einen Typ.

3.1.1 Abstrakter Regler als ein objektorientiertes Beispielsystem

In diesem Abschnitt soll am Beispiel eines abstrakten Reglers die grundlegende Arbeitsweise eines objektorientierten Softwaresystems erläutert werden. Ein abstrakter Regler besteht aus einer Klasse *abstractController* für den Regler, einer Klasse *abstractActor* für den Aktor und einer Klasse *abstractSensor* für den Sensor (s. Abbildung 3.1). Die Sensorklasse *abstractSensor* besitzt ein Attribut x, auf das mit der Operation *getValue* zugegriffen wird, um den im Sensor gemessenen Istwert auszulesen. Die Klasse *abstractActor* für den Aktor besitzt ein Attribut y, welches den aktuellen Stellwert repräsentiert und durch die Operation *setValue* gesetzt wird. Die Reglerklasse *abstractController* besitzt die Attribute x und y, um den Stellwert und den Istwert aufzunehmen. Sie verfügt außerdem über die Operationen *loop* und *compute*. Die Operation *loop* wird einmalig aufgerufen, um in die Regelschleife zu gelangen. Die Operation *compute* wird verwendet, um den neuen Stellwert zu berechnen. Zur Laufzeit besitzt ein Reglerobjekt eine Referenz auf ein Aktorobjekt und eine Referenz auf ein Sensorobjekt. Die Operation *loop* wird einmalig angestoßen. In der Operation *loop* wird die Operation *getValue* des Sensorobjektes aufgerufen, die den Istwert zurückliefert, welcher in der Instanzvariable x abgelegt wird. Anschließend wird die Operation *compute* aufgerufen, um den neuen Stellwert zu ermitteln, der in der Instanzvariable y abgelegt wird. Zuletzt wird mit einer *getValue*-Nachricht an das Aktorobjekt der neue Stellwert gesetzt.

3.2 Grundlegende Begriffe der objektorientierten Software-Modellierung

In diesem Abschnitt werden Definitionen für die zentralen Begriffe der UML angegeben und in Tabelle 3.1 in einer Übersicht aufgeführt.

Zentral ist der Begriff des Objektes. Ein Objekt ist eine konkret vorhandene und agierende Einheit mit eigener Identität und definierten Grenzen, die einen eigenen Status und ein Verhalten kapselt [HK05, Stö05]. Ein Objekt besitzt Instanzvariablen, die Werte annehmen und Objektvariablen, die andere Objekte referenzieren. Die Identität eines Objektes ist von seinen Instanzvariablen unabhängig und nicht veränderbar. Die Werte dieser Variablen beziehen sich auf das individuelle Objekt. In einem Objekt sind Dienste in Form von Operationen vorhanden, die von anderen Objekten aufgerufen werden können.

Ein Attribut beschreibt eine Instanzvariable, die in einem Objekt enthalten ist und für die jedes Objekt einen individuellen Wert aufweist. Ein Attribut besteht aus einem Namen, einem Typ und einer Sichtbarkeit. Die Sichtbarkeit beschreibt die Zugreifbarkeit eines Attributes. Eine durch ein Attribut beschriebene Instanzvariable ist vollständig unter der Kontrolle des Objektes, zu dem sie gehört und hat außerhalb dieses Objektes keine eigene Identität.

Operationen sind Dienste, die von einem Objekt angefordert werden können. Sie werden durch ihre Signatur, die sich aus dem Operationsnamen, den Parametern mit ihren Namen und Typen und einem gegebenenfalls vorhandenem Rückgabetyp zusammensetzt, beschrieben.

Durch eine Klasse werden die Attribute und die Operationen zu einer Schablone für eine Menge von Objekten zusammengeführt. Eine Klasse ist die Definition der Attribute, Operationen und der Semantik für eine Menge von Objekten. Alle Objekte einer Klasse entsprechen dieser Definition.

Durch eine Assoziation wird eine Relation zwischen zwei Klassen angegeben.

Um Beziehungen zwischen zwei Objekten zu verstehen, ist der Begriff des *Links* relevant. Ein Link ist eine Instanz einer Assoziation und repräsentiert somit eine Referenz auf ein anderes Objekt. Eine Objektvariable nimmt zur Laufzeit einen oder mehrere Links auf.

Jetzt soll die Verarbeitung in einem Objekt betrachtet werden. Hier ist der Begriff der *Action* für das Verständnis der Verarbeitungsschritte eines Objektes wesentlich. Eine *Action* ist ein atomarer Verarbeitungsschritt, der innerhalb eines Objektes ausgeführt wird. Ein solcher Verarbeitungsschritt kann unterschiedliche Auswirkungen besitzen. Zu den möglichen Auswirkungen gehören die Änderung der Werte von Instanz- bzw. Objektvariablen, der Aufruf bzw. die Entgegennahme einer Operation oder die Antwort auf einen Operationsaufruf.

Actions besitzen *Pins*, um parameterähnlich Werte und Objekte als Eingaben entgegenzunehmen bzw. bei der Verarbeitung entstandene Objekte und Werte als Ausgaben zur weiteren

Verarbeitung über Objektflüsse weiterzureichen.

Ein Objektfluss beschreibt die Weiterleitung eines Objektes zwischen zwei *Actions* über Pins.

Die Weiterleitung von Kontrollinformation in Form von Token zwischen zwei *Actions* zur Gewährleistung einer bestimmten Reihenfolge geschieht durch einen Kontrollfluss.

Durch eine *Aktivität* (engl. activity) wird auf der Basis von Kontroll- und Objektflüssen die Ausführungsreihenfolge von einer Menge von *Actions*, die in der *Aktivität* modelliert sind, festgelegt.

Wesentlich für ein Objekt sind dessen Zustände und wie Übergänge zwischen diesen erfolgen. Ein Zustand definiert eine bestimmte Situation, in der sich ein Objekt befindet. Hierbei wird in Kontrollzustand, Datenzustand, Zustand der Ereigniswarteschlange und Lebenszykluszustand unterschieden. Der Datenzustand gibt an, welche Werte die Instanzvariablen und Objektvariablen eines Objektes besitzen. Der Kontrollzustand beschreibt, welche *Ereignisse* von einem Objekt zum gegenwärtigen Zeitpunkt verarbeitet werden können. Der Zustand der Ereigniswarteschlange gibt an, welche *Ereignisse* von einem Objekt empfangen worden sind. Weiterhin wird der Lebenszykluszustand definiert, der angibt, ob ein Objekt gegenwärtig inaktiv, aktiv oder terminiert ist. Der Zustand eines Objektes wird aus dem Daten-, dem Kontroll-, dem Lebenszykluszustand und dem Zustand der Ereigniswarteschlange gebildet.

Ein *Ereignis* ist der Aufruf einer Operation, das Eintreffen eines Signals oder der Ablauf einer Frist. *Ereignisse* werden in der Ereigniswarteschlange bis zum Zeitpunkt ihrer Verarbeitung zwischengespeichert, damit eingehende *Ereignisse* die augenblicklich stattfindende Verarbeitung nicht unterbrechen. *Ereignisse* enthalten Datenwerte in vordefinierten Parametern. Solche *Ereignisse* können in einem Objekt zu einem beliebigen Zeitpunkt auftreten. *Ereignisse* werden entweder durch bestimmte *Actions* aus einem anderen Objekt ausgelöst, die den Aufruf einer Operation bzw. die Versendung eines Signals bewirken oder durch den Ablauf einer im selben Objekt spezifizierten Frist verursacht.

Eine Transition ist ein Zustandsübergang, der durch ein *Ereignis* verursacht wird, das als Auslöser (engl. Trigger) bezeichnet wird, falls eine bestimmte an der Transition modellierte Bedingung erfüllt ist. Transitionen, die keinen Auslöser besitzen, werden als *triggerless* bezeichnet. Transitionen sind der wichtigste Ausgangspunkt für die Ausführung einer Menge von *Actions*. Beim Schalten einer Transition wird eine an dieser modellierte *Aktivität* ausgeführt, was wiederum zur Ausführung ihrer *Actions* in der durch die Objekt- und Kontrollflüsse festgelegten Reihenfolge führt.

Abschließend sollen die Begriffe, die die Kooperation von Objekten durch Interaktion behandeln, eingeführt werden.

Eine Nachricht (engl. message) ist ein Mechanismus mit dem Objekte untereinander kommunizieren. Mit einer Nachricht wird ein Objekt aufgefordert eine Operation auszuführen oder eine Information durch ein Signal entgegenzunehmen. Eine Nachricht besteht aus einem Namen und einer Liste von Argumenten und wird an genau einen Empfänger versendet. Der Sender einer Nachricht erhält gegebenenfalls eine Antwort.

Durch einen Ereignisauftritt wird das Auftreten einer *Action* in einem bestimmten Objekt zu einem definierten Verarbeitungszeitpunkt eines Objektes bezeichnet, d. h. jede Ausführung einer *Action* führt zu einem Ereignisauftritt, der einem bestimmten Objekt zugeordnet werden kann. *Ereignisse* und Ereignisauftritte haben trotz des ähnlichen Namens nichts miteinander zu tun.

Um die Abfolge der Ereignisauftritte der Objekte eines objektorientierten Systems anzugeben, wird der Begriff des *Traces* verwendet. Ein *Trace* ist eine Sequenz von Ereignisauftritten.

3.2.1 Begriffe am Beispiel des abstrakten Reglers

Die Klasse *Controller*, die in Abbildung 3.1 gezeigt ist, beschreibt die Struktur von Reglerobjekten. Sie besitzt das Attribut x, um den Istwert aufzunehmen, während der Stellwert durch das Attribut y repräsentiert wird. Die Operation *loop*, die keinen Rückgabewert besitzt, und die Operation *compute* enthalten alle Verarbeitungsschritte des Reglers. Die Klasse *Controller* besitzt Assoziationen zur Klasse *Sensor* und zur Klasse *Actor* der Multiplizität eins, d. h. zur Laufzeit

Begriff	Definition
Objekt	Eine konkret vorhandene und agierende Einheit mit eigener Identität und definierten Grenzen, die einen Status und ein Verhalten kapselt [HK05]. Die Begriffe des Objektes und der Instanz sind synonym. Ein Objekt besitzt Instanzvariablen, die Werte annehmen und Objektvariablen, die andere Objekte referenzieren. Die Werte dieser Variablen beziehen sich auf das individuelle Objekt. In einem Objekt sind Operationen vorhanden, die anderen Objekten Dienste bereitstellen. Jedes Objekt hat eine eigene von seinen Instanzvariablen unabhängige, nicht veränderbare Identität.
Attribut	Ein Attribut beschreibt eine Instanzvariable, die in einem Objekt enthalten ist und für das jedes Objekt einen individuellen Wert aufweist. Ein Attribut besteht aus einem Namen, einem Typ und einer Sichtbarkeit. Die Sichtbarkeit beschreibt die Zugreifbarkeit eines Attributes. Ein Attribut ist vollständig unter der Kontrolle des Objektes, das es beschreibt und hat außerhalb dieses Objektes keine eigene Identität.
Operation	Operationen sind Dienste, die von einem Objekt angefordert werden können. Sie werden beschrieben durch ihre Signatur, die aus dem Operationsnamen, der Sichtbarkeit, den Parametern und einem gegebenenfalls vorhandenem Rückgabetyp besteht.
Klasse	Eine Klasse ist die Definition der Attribute, Operationen und der Semantik für eine Menge von Objekten. Alle Objekte einer Klasse entsprechen dieser Definition.
Assoziation	Eine Assoziation beschreibt eine Relation zwischen zwei Klassen.
Link	Ein Link ist die Instanz einer Assoziation und repräsentiert eine Referenz auf ein anderes Objekt. Eine Objektvariable nimmt einen Link in einem Objekt auf.
Action	Eine *Action* ist ein atomarer Verarbeitungsschritt eines Objektes, der die Werte von Attributen oder Links verändert oder den Aufruf bzw. die Entgegennahme einer Operation oder die Antwort auf einen Operationsaufruf zur Folge hat. Aktionen besitzen Parameter, um Werte entgegenzunehmen bzw. weiterzugeben.
Pin	Ein Pin ist ein Ein- oder Ausgabeparameter einer *Action* über den Werte entgegengenommen bzw. weitergeleitet werden.
Objektfluss	Ein Objektfluss beschreibt die Weiterleitung eines Objektes zwischen zwei *Actions* über Pins.
Kontrollfluss	Ein Kontrollfluss beschreibt die Weiterleitung von Kontrollinformation in Form von Token zwischen zwei *Actions* zur Gewährleistung einer bestimmten Ausführungsreihenfolge.
Aktivität	Eine *Aktivität* (engl. activity) beschreibt auf der Basis von Kontroll- und Objektflüssen die Ausführungsreihenfolge einer Menge von Actions, die in der *Aktivität* modelliert sind.
Zustand	Ein Zustand definiert eine bestimmte Situation, in der sich ein Objekt befindet. Hier wird der Kontrollzustand, der Datenzustand und der Zustand der Ereigniswarteschlange unterschieden. Der Datenzustand gibt an, welche Werte die Instanzvariablen und Objektvariablen eines Objektes besitzen. Der Kontrollzustand beschreibt, welche *Ereignisse* von einem Objekt zum gegenwärtigen Zeitpunkt verarbeitet werden können. Der Zustand der Ereigniswarteschlange gibt an, welche *Ereignisse* von einem Objekt empfangen worden sind und dies für den Empfang weiterer *Ereignisse* blockiert ist. Der Lebenszykluszustand gibt an, ob ein Objekt augenblicklich inaktiv, aktiv oder terminiert ist. Der Zustand eines Objektes besteht aus dem Daten-, dem Kontroll-, dem Lebenszykluszustand und dem Zustand der Ereigniswarteschlange.
Ereignis	Ein *Ereignis* ist der Aufruf einer Operation, das Eintreffen eines Signals oder der Ablauf einer Frist. *Ereignisse* werden in der Ereigniswarteschlange bis zum Zeitpunkt ihrer Verarbeitung zwischengespeichert.
Transition	Eine *Transition* ist ein Zustandsübergang, der durch ein *Ereignis*, das als Auslöser (engl. Trigger) bezeichnet wird, verursacht wird, falls eine bestimmte an der Transition modellierte Bedingung erfüllt ist. Transitionen, die keinen Auslöser besitzen, werden als triggerless bezeichnet. Beim Schalten einer Transition können *Aktivitäten* ausgeführt werden, die wiederum die Ausführung ihrer Actions verursachen.
Nachricht	Eine *Nachricht* (engl. message) ist ein Mechanismus mit dem Objekte untereinander kommunizieren. Mit einer Nachricht wird ein Objekt aufgefordert eine Operation auszuführen oder Information durch ein Signal entgegenzunehmen. Eine Nachricht besteht aus einem Namen sowie einer Liste von Argumenten und wird an genau einen Empfänger versendet. Der Sender einer Nachricht erhält gegebenenfalls eine Antwort.
Ereignisauftritt	Durch einen *Ereignisauftritt* (engl. EventOccurrence) wird das Auftreten einer *Action* in einem Objekt zu einem definierten Verarbeitungszeitpunkt des Objektes bezeichnet.
Trace	Ein *Trace* ist eine Sequenz von Ereignisauftritten $< E_1, E_2 \ldots, E_n >$.

Tabelle 3.1: Die zentralen Begriffe der UML

steht ein Controller-Objekt mit genau einem Sensor- und einem Aktorobjekt in Beziehung. Um die Verarbeitung durch ein Reglerobjekt zu starten, muss eine *loop* Nachricht an das Reglerobjekt gesendet werden. Ein Reglerobjekt besitzt diverse Kontrollzustände, die bei der Ausführung der Regelschleife immer wieder betreten werden. Die Regelschleife wird durch einen einmaligen Aufruf der parameterlosen *loop* Operation betreten. Dieser Aufruf verursacht ein entsprechendes *Ereignis*, das den Zustand der Ereigniswarteschlange modifiziert und eine Transition vom Zustand *init* in den Zustand *wait* auslöst. Der Kontrollzustand *stopped* wird zur Unterbrechung der Regelschleife in einem Reglerobjekt betreten. Die Kontrollzustände, die in einem Reglerobjekt in der Regelschleife immer wieder betreten werden, heißen *Idle*, *waitSensor*, *valuesComputed* und *setValueCalled*. Zwischen diesen Zuständen gibt es jeweils triggerless Transitionen an denen eine Aktivität spezifiziert ist. Die Aktivität *getValue* zwischen den Zuständen *Idle* und *waitSensor* dient der Ermittlung des Istwertes aus den Instanzvariablen x und y, die den Datenzustand des Reglerobjektes repräsentieren. Durch die Aktivität *compute* zwischen den Zuständen *waitSensor* und *valuesComputed* wird der neue Stellwert aus dem Istwert und dem alten Stellwert für das Aktorobjekt berechnet. Die Aktivität *setValue*, die zwischen den Zuständen *valuesComputed* und *setValueCalled* ausgeführt wird, setzt den neuen Stellwert. Die Aktivität *compute* enthält die Actions *lese x-Wert*, *lese y-Wert*, um die Instanzvariablen x und y auszulesen, deren Werte in den Pins der Action für die Objektflüsse zur Verfügung gestellt werden. Weiterhin gibt es die Action *berechne neuen y-Wert*, um den neuen Stellwert zu berechnen und eine Action *schreibe y-Wert*, um diesen in die Instanzvariable x zu schreiben. Durch die Kontroll- und Objektflüsse zwischen diesen *Actions* wird die oben beschriebene Reihenfolge, in der diese ausgeführt werden, angegeben. Die Ausführung jeder der oben angegebenen *Actions* in einem Reglerobjekt repräsentiert somit einen *Ereignisauftritt*. Der Trace der Aktivität *compute* des Reglerobjektes besteht aus den Ereignisauftritten der folgenden *Actions* $< lese\ x-Wert, lese\ y-Wert, berechne\ neuen\ y-Wert, schreibe\ y-Wert >$.

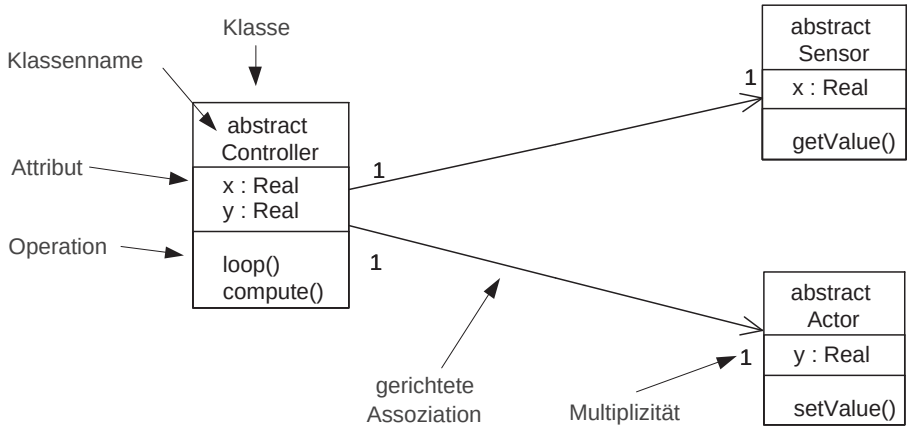


Abbildung 3.1: Das Klassendiagramm eines abstrakten Controllers

3.3 Actions

Im Metamodell von UML 2.0 wird in den Packages *Invocation Actions*, *Read and Write Actions* und *Complete Actions* des UML Metamodells die Semantik von zahlreichen, verschiedenen Typen von *Actions* definiert, von denen die wichtigsten in den folgenden Abschnitten vorgestellt werden. Hierbei wird jeweils angegeben, welche Zustandsarten durch die *Action* betroffen sind.

3.3.1 Kommunikationsbezogene Actions

Kommunikationsbezogene Actions entstammen in UML 2.0 dem Package InvocationActions. Es gibt zahlreiche Typen *kommunikationsbezogener Actions*. Hier werden *CallOperationActions* und

SendObjectActions behandelt.

Eine *CallOperationAction* ist eine *Action*, die eine Operationsaufruf-Anfrage an ein Zielobjekt weiterleitet, wo sie den Aufruf einer passenden Operation auslöst. Die Voraussetzung hierfür ist, dass die Parameter der *Action* mit den Parametern der Operation bezüglich der Anzahl und deren Typ verträglich sind. Dies entspricht dem Aufruf einer Operation auf einem anderen Objekt. Eine aufgerufene Operation wird selbst wiederum durch eine Folge von *Actions* aus einer oder mehreren *Aktivitäten* im aufgerufenen Objekt umgesetzt. Diese dienen der Verarbeitung des Operationsaufrufes, der Ermittlung des Ergebnisses und dessen Rückübertragung. Die Argumentwerte der *Action* stehen für die Ausführung der aufgerufenen Operation zur Verfügung. Wenn die *Action* als synchron spezifiziert ist, wird die Ausführung der *CallOperationAction* angehalten bis der Aufruf abgeschlossen ist und die Übertragung der Antwort mit den Ergebnissen zum Aufrufer stattgefunden hat.

Alle Werte, die als Bestandteil der Rückübertragung von Ergebnissen aus dem aufgerufenen Objekt zurückgeliefert werden, werden an den Ausgabepins der *CallOperationAction*, die das Ergebnis entgegennehmen, bereitgestellt. Nach der Übertragung der Antwort, ist die Ausführung einer *CallOperationAction* abgeschlossen. *CallOperationActions* modifizieren den Zustand der Ereigniswarteschlange des Sender- und des Empfängerobjektes. Im Senderobjekt wird eine Blockierung vorgenommen, damit keine weiteren *Ereignisse* entgegengenommen werden können. Im Empfängerobjekt wird die Ereigniswarteschlange mit einem entsprechenden *Ereignis* gefüllt. Eine *CallOperationAction* modifiziert den Datenzustand, den Zustand der Ereigniswarteschlange des Senderobjektes und den Zustand der Ereigniswarteschlange des Empfängerobjektes.

Eine *SendObjectAction* überträgt ein sog. Signal-Objekt an das Zielobjekt, indem eine Transition, die als Auslöser das Signal besitzt, ausgelöst und gegebenenfalls die Ausführung einer *Aktivität* im Zielobjekt gestartet wird. Die Werte der Attribute des Aufruf-Objektes stehen für die weitere Verarbeitung im aufgerufenen Objekt zur Verfügung. Das aufrufende Objekt setzt seine Verarbeitung augenblicklich fort. Jede Antwortnachricht wird ignoriert und nicht an das aufrufende Objekt weitergeleitet. Durch eine *SendObjectAction* wird der Zustand der Ereigniswarteschlange des Empfängerobjektes modifiziert. Durch eine *CallOperationAction* wird eine Modifikation am Datenzustand und dem Zustand der Ereigniswarteschlange des Senderobjektes sowie am Zustand der Ereigniswarteschlange des Empfängerobjektes vorgenommen.

3.3.2 Lese- und schreiborientierte Actions

Objekte, Attribute und Links besitzen Werte, die für *Actions* zugreifbar sind. Links referenzieren Objekte mit ihren Enden und können angelegt, erzeugt sowie zerstört werden. Sämtliche dieser Eigenschaften werden durch unterschiedliche *Actions* behandelt. *Lese- und schreiborientierte Actions* (*Read Write Actions*) lassen sich in *StructuralFeatureActions* und *LinkActions* untergliedern und modifizieren den Datenzustand eines Objektes. *StructuralFeatureActions* dienen dem lesenden und schreibenden Zugriff auf Instanzvariablen. *LinkActions* ermöglichen den lesenden und schreibenden Zugriff auf Objektvariablen, deren Assoziationen in einem Klassendiagramm spezifiziert worden sind.

Die Bedeutung einer *ReadStructuralFeatureAction* besteht darin, den Wert eines Attributes an einem Ausgabepin zur Verfügung zu stellen. Sollten mehrere Werte vorhanden sein, so werden diese unter Beibehaltung der definierten Ordnung ausgelesen. Aus Gründen der besseren Lesbarkeit wird die Bezeichnung *ReadAttributeAction* verwendet, falls ein Attribut gelesen werden soll. Diese Bezeichnung stammt noch aus der UML 1.5.

Um einem Attribut Werte zuzuweisen, wird eine *AddStructuralFeatureValueAction* verwendet. Die Interpretation einer solchen Action besteht darin, einer Instanzvariablen bzw. einem Link einen Wert zuzuweisen, um den Datenzustand eines Objektes zu modifizieren. Wiederum soll aus Gründen der besseren Lesbarkeit die Bezeichnung *WriteAttributeAction* verwendet, falls einem Attribut ein Wert zugewiesen werden soll. Diese Bezeichnung wurde ursprünglich in der UML 1.5 verwendet. Dies ist sinnvoll, da Links in der Arbeit nicht modifiziert werden. Eine *RemoveStructuralFeatureValueAction* erlaubt es den Wert einer Instanzvariablen zu entfernen.

Eine *CreateLinkAction* dient der Erzeugung eines Links, wodurch eine Referenz zwischen zwei

Objekten hergestellt wird. Diese *Action* besitzt keinen Rückgabewert für den Fall der Erzeugung eines Links, weil Links nicht als Werte zwischen *Actions* übertragen werden dürfen. Die Semantik dieser *Action* ist in bestimmten Fällen undefiniert. Der wichtigste Fall hierfür ist die Überschreitung der Multiplizität an einem Assoziationsende, wenn mit der *Action* ein neuer Link angelegt werden soll.

Mit einer *DestroyLinkAction* wird die Zerstörung eines Links einer Objektvariable vorgenommen, d. h. eine Referenz auf ein Objekt wird beseitigt.

Eine *ReadLinkAction* ist eine *Action* mit der über eine Assoziation in eine vorgegebene Richtung navigiert wird, um die Objekte an einem bestimmten Ende eines Links zu erhalten. Die *Action* erhält über einen Eingabepin das Objekt von dem ausgegangen wird. Der Ausgabepin der *Action* erhält als Ergebnis die Objekte, die an der Assoziation teilnehmen. Somit werden Referenzen auf Objekte, die Links einer bestimmten Assoziation sind, bereitgestellt. Die Semantik ist für den Fall, dass in eine nicht zulässige Richtung navigiert wird, undefiniert.

3.3.3 Berechnungsbezogene Actions

Berechnungsbezogene Actions (engl. Computation Actions) transformieren eine Menge von Eingabewerten in eine Menge von Ausgabewerten durch den Aufruf einer primitiven Funktion und modifizieren den Datenzustand eines Objektes auf der Basis von Berechnungen. Primitive Funktionen repräsentieren Abbildungen von Mengen mit Eingabewerten auf eine Menge von Ausgabewerten. Das Ergebnis des Aufrufes einer primitiven Funktion hängt nur von den Eingabewerten der Funktion ab und hat keine weitere Auswirkung als die Berechnung der Ausgabewerte. Eine primitive Funktion nimmt keine Interaktionen mit anderen Objekten vor. Typische primitive Funktionen enthalten Boolesche und arithmetische Operationen sowie Funktionen auf Zeichenketten.

Eine *ApplyFunctionAction* ist ein wichtiger Repräsentant dieses Actiontyps. Die Semantik einer *ApplyFunctionAction* besteht aus der Übergabe von Eingabewerten an den Eingabepins und dem Aufruf einer primitiven Funktion, die Ausgabewerte in Abhängigkeit von den Eingabewerten und der jeweiligen Funktion berechnet, um sie an den Ausgabepins der *Action* zur Verfügung zu stellen.

3.3.4 Objektbezogene Actions

In diesem Abschnitt werden die *Actions* des Packages *Complete Actions* des UML 2.0 Metamodells behandelt. *AcceptEvent-* und *AcceptCallActions* sind *Actions*, die für sie geeignete *Ereignisse* aus der Ereigniswarteschlange eines Objektes entfernen und diese sukzessiv leeren, wodurch der Zustand der Ereigniswarteschlange modifiziert wird und die *Ereignisse* für eine Änderung des Kontrollzustandes bereitgestellt werden. Diese Ereigniswarteschlange puffert *Ereignisse*, die zum gegenwärtigen Zeitpunkt nicht durch ein Objekt verarbeitet werden können, d. h. diese *Actions* können ausgeführt werden, sobald ein passendes *Ereignis* in der Warteschlange eines Objektes vorliegt.

AcceptEventActions sind nach [BHK04] immer ausführbar, wenn keine entsprechenden Kontrollflussbedingungen vorliegen und die zugehörige *Aktivität* gerade ausgeführt wird. Das bedeutet, eine derartige *Action* kann beliebig oft ausgeführt werden, solange die zugehörige *Aktivität* ausgeführt wird. Eine *AcceptEventAction* entfernt asynchrone *Ereignisse* und *Ereignisse* ohne Rückgabeparameter aus der Ereigniswarteschlange, um sofort eine parameterlose Rückantwort (engl. reply) zu liefern.

Eine *AcceptCallAction* findet Verwendung, um einen synchronen Operationsaufruf aus der Ereigniswarteschlange eines Empfängerobjektes entgegen zu nehmen. Zusätzlich zu den normalen Parametern der Operation erzeugt die *Action* ein Token, das später verwendet wird, um einer *ReplyAction* Informationen bereitzustellen, welche die Kontrolle an den Aufrufer zurückliefert.

Durch eine *ReplyAction* wird die Ausführung einer Operation, die durch eine *CallOperationAction* begonnen wurde, abgeschlossen. Eine *ReplyAction* ist eine *Action*, die eine Menge von Rückgabewerten und ein Token, das Informationen bzgl. der Rücklieferung repräsentiert und von der zuvor aufgerufenen *AcceptCallAction* erzeugt wurde, enthält. Die *ReplyAction* liefert die Werte an den Aufrufer zurück, um die Ausführung eines Operationsaufrufes abzuschließen, wobei der Datenzustand des Empfängerobjektes verändert wird. Im Anschluss an eine *AcceptCallAction* [BHK04]

ist die Ausführung maximal einer zugehörigen *ReplyAction* gestattet. Wird diese nicht vorgenommen, bleibt das aufrufende Objekt möglicherweise blockiert.

Eine *CreateObjectAction* dient der Erzeugung eines neuen Objektes einer bestimmten Klasse, wodurch der Lebenszykluszustand eines Objektes auf *aktiv* gesetzt wird. Die Klasse des zu erzeugenden Objektes muss an einem Eingabepin zur Verfügung gestellt werden. Das durch die *Action* erzeugte neue Objekt wird an einem Ausgabepin bereitgestellt.

Eine *DestroyObjectAction* zerstört ein Objekt. Das zu zerstörende Objekt wird an einen Eingabepin der *Action* übergeben. Der Lebenszykluszustand eines bestehenden Objektes wird hierbei auf *terminiert* gesetzt.

3.4 Diagramme der UML

In diesem Abschnitt werden die für diese Arbeit relevanten Diagramme der UML vorgestellt.

3.4.1 Klassendiagramme

Klassendiagramme sind die grundlegenden Diagramme zur Beschreibung statischer Modellelemente in der UML. Durch Klassendiagramme wird die Modellierung von Klassen und deren Beziehungen untereinander unterstützt. Klassen werden durch Rechtecke mit Abschnitten dargestellt. Für eine Klasse ist die Modellierung von Attributen mit Namen, Typ und Sichtbarkeit sowie von Operationen mit dem Namen, der Parameterliste und dem Rückgabewert in bestimmten Abschnitten möglich. In Abbildung 3.1 ist das Klassendiagramm mit den Klassen *abstractController*, *abstractSensor* und *abstractActor*, die einen abstrakten Regler bilden, gezeigt. Die Klasse *abstractController* besitzt die Attribute x und y vom Typ *Real*. Weiterhin sind die Operationen *compute* und *loop* modelliert. Die Assoziation zwischen den Klassen *abstractController* und *abstractSensor* in Abbildung 3.1 wird durch eine gerichtete Kante zwischen den Klassen repräsentiert, die an den Assoziationsenden jeweils eine der beiden Klassen besitzt. Gleiches gilt für die Assoziation zwischen den Klassen *abstractController* und *abstractActor*. Das Assoziationsende besitzt eine Multiplizität, die angibt, wie viele Objekte durch die Assoziation in Beziehung gesetzt werden dürfen. Hier ist die Multiplizität an den Assoziationsenden jeweils eins, was bedeutet, dass zur Ausführungszeit ein Objekt der Klasse *abstractController* mit einem Objekt der Klasse *abstractActor* und einem Objekt der Klasse *abstractSensor* über Links verbunden sind. Assoziationen können mit einem Pfeil versehen werden, um anzugeben, dass nur eine Referenz von einem bestimmten Objekt auf ein anderes vorhanden ist, man also nur von einem bestimmten Objekt zu einem anderen navigieren kann.

3.4.2 Aktivitäten

Actions werden untereinander durch Aktivitäten koordiniert [Boc03]. Die UML erlaubt es Aktivitäten mit unterschiedlicher Mächtigkeit zu spezifizieren. Die in dieser Arbeit relevanten *BasicActivities* stellen den grundlegenden Sprachumfang bereit, um *Actions* sequentiell oder in Abhängigkeit von Bedingungen auszuführen sowie Daten und Kontrollinformation zwischen *Actions* weiterzugeben. *Aktivitäten* bestehen aus Aktionsknoten (engl. action node), die jeweils eine zugeordnete *Action*, wie sie im Abschnitt 3.3 detailliert vorgestellt wurden, besitzen. Somit bündeln *Aktivitäten* *Actions*, die den Datenzustand, den Lebenszykluszustand, den Zustand der Ereigniswarteschlange und den Kontrollzustand des eigenen oder eines fremden Objektes verändern können.

Es gibt noch zwei weitere Knotenarten. Kontrollknoten, wie Entscheidungen, leiten Kontroll- und Datenwerte durch das Aktivitätsdiagramm. Objektknoten nehmen Objekte bzw. Datenwerte temporär auf.

Eine gerichtete Objektflusskante modelliert den Objektfluss zwischen zwei Aktionsknoten. Objektflusskanten verbinden Objektknoten mit Aktionsknoten, um Objekte zur Verarbeitung bereitzustellen oder zur folgenden *Action* weiterzuleiten. Ein Aktionsknoten kann ein- und ausgehende Kontroll- oder Objektflusskanten besitzen. Für die Verbindung mit ein- und ausgehenden Objektflusskanten werden *Pins* als kleine Rechtecke an dem Symbol eines Aktionsknotens modelliert, um

die Weiterleitung eines Objektes durch einen Objektfluss zu spezifizieren. Die *Pins* einer *Action* werden mit den Pins des zugehörigen Aktionsknotens verbunden. Durch die Definition einer *Action* wird angenommen, dass die *Pins* geordnet sind.

Durch einen Aktionsknoten wird ähnlich wie bei Petrinetzen eine Synchronisation von Kontroll- und Objektflüssen ermöglicht. Eine *Action*, die einem Aktionsknoten zugeordnet ist, wird erst dann ausgeführt, wenn alle eingehenden Kontroll- und Datenflüsse aufgetreten sind. Bei der Beendigung der *Action* werden die ausgehenden Daten- und Kontrollflüsse ausgelöst, indem Ausgangspins mit neuen Werten belegt und neue Token für die ausgehenden Kontrollflüsse bereitgestellt werden. Die neuen Werte für Objektflüsse werden bei der Verarbeitung durch die zugeordnete *Action* erzeugt.

In Abbildung 3.2 ist eine *Aktivität*, die die Berechnung der Stellgröße für den abstrakten Regler spezifiziert, angegeben. In der Abbildung sind die Aktionsknoten *lese x-Wert*, *lese y-Wert*, *berechne neuen y-Wert* und *schreibe y-Wert* gezeigt. Die Aktionsknoten *lese x-Wert* und *lese y-Wert* besitzen jeweils eine Zuordnung zu einer *ReadAttributeAction*. Dem Aktionsknoten *berechne neuen y-Wert* ist eine *ApplyFunctionAction* und dem Aktionsknoten *schreibe y-Wert* ist eine *WriteAttributeAction* zugeordnet. In Abbildung 3.2 sind die Objektknoten *x-Wert*, *y-Wert* und *neuer y-Wert* vom Typ *Wert* vorhanden. Eine gerichtete Kontrollflusskante modelliert einen Kontrollfluss zwischen zwei Aktionsknoten. Diese gibt an, dass ein Verarbeitungsschritt nicht beginnen darf, bevor ein anderer beendet worden ist. So darf die *Action lese y-Wert* erst ausgeführt werden, nachdem die *Action lese x-Wert* ausgeführt wurde. Der Objektknoten *x-Wert* leitet ein Wertobjekt von der *Action lese x-Wert* zur *Action berechne neuen y-Wert* über Objektflusskanten weiter.

3.4.3 Statechart-Diagramme

Statechart-Diagramme werden zur Spezifikation der Kontrollzustände jedes einzelnen Objektes einer bestimmten Klasse definiert. Statechart-Diagramme haben ihren Ursprung in den von Harel [Har87] entwickelten Statecharts. Im Folgenden wird zunächst die Zustandsmaschine beschrieben, bevor auf Kontrollzustände eingegangen wird.

Zustandsmaschine

Die Verarbeitung von Ereignissen und die Auswahl von Transitionen wird in UML 2.0 von der sog. Zustandsmaschine (engl. state machine) vorgenommen. Ereignisse, die an ein Objekt übergeben werden, werden zunächst in der Ereignis-Warteschlange der Zustandsmaschine gepuffert. Grundlegend für die Verarbeitung von Ereignissen durch die Zustandsmaschine ist die Run-To-Completion Semantik, die angibt, dass zu einem Zeitpunkt jeweils nur ein Ereignis verarbeitet wird. Damit darf eine einmal begonnene Ereignisverarbeitung nicht durch ein neu aufgetretenes *Ereignis* unterbrochen werden und zwischenzeitlich eingehende *Ereignisse* werden in der Ereigniswarteschlange gepuffert.

Darstellung von Zuständen und Transitionen in Statechart-Diagrammen

Zustände in Statechart-Diagrammen gehören zu den Kontrollzuständen eines Objektes. Wenn im Folgenden in diesem Abschnitt der Begriff *Zustand* verwendet wird, ist immer ein Kontrollzustand gemeint. Sämtliche Kontrollzustände eines Objektes bilden dessen Kontrollzustandsraum. Ein solcher Zustand ist während der Ausführung entweder aktiv oder inaktiv. Ein Zustand wird aktiv, wenn er als Ergebnis einer Transition betreten wurde und inaktiv, wenn er durch eine Transition verlassen wird. Ein Zustand wird durch ein Objekt für eine bestimmte Verweildauer betreten. Durch eine Selbsttransition kann ein Zustand verlassen und erneut betreten werden.

Die aktive Zustandskonfiguration gibt an, welche Zustände und Unterzustände in einem Statechart zu einem bestimmten Zeitpunkt aktiv sind. Mit dem Begriff aktiver Kontrollzustand soll der gegenwärtig besuchte Kontrollzustand bezeichnet werden.

Die Notation von Zuständen erfolgt in UML durch Rechtecke mit abgerundeten Kanten. Bei einem Zustand sind bestimmte Abschnitte spezifizierbar:

- Ein Abschnitt, in dem ein eindeutiger Name vergeben wird.

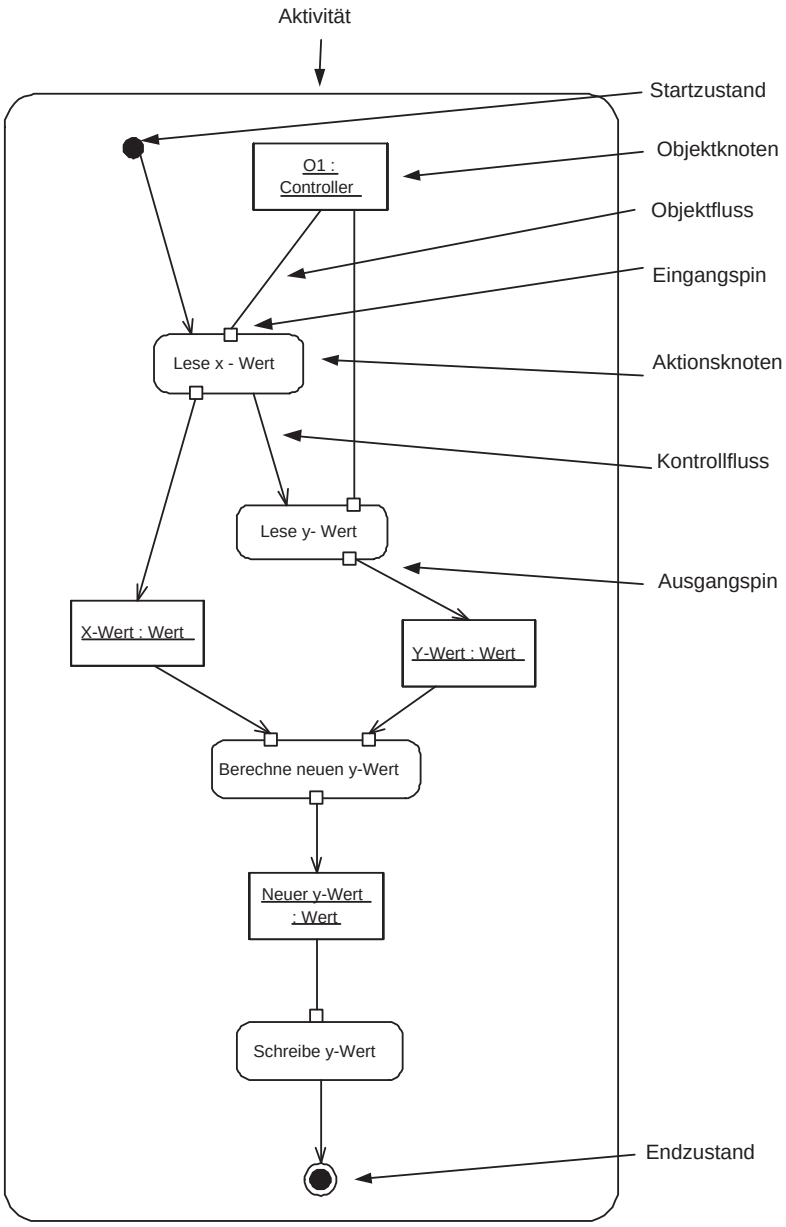


Abbildung 3.2: Die Aktivität `compute` zur Berechnung einer neuen Stellgröße

- Ein Abschnitt, der mit dem Schlüsselwort *entry* gekennzeichnet ist, durch den eine *Aktivität* definiert werden kann, die sofort beim Betreten des Zustandes ausgeführt wird. Zusätzlich ein Abschnitt, der mit dem Schlüsselwort *exit* beschriftet ist und in dem eine *Aktivität* definiert werden kann, die beim Verlassen des Zustandes ausgeführt wird. Die Spezifikation beider Abschnitte ist nicht notwendig.
- Ein Abschnitt, der mit dem Schlüsselwort *do* gekennzeichnet ist und welcher eine Aktivität enthält, die ausgeführt wird, solange ein Objekt sich in einem bestimmten Zustand befindet. Die Beendigung dieser Aktivität kann bei entsprechender Modellierung das Verlassen eines Zustandes verursachen. Auch die Spezifikation dieses Abschnitts ist optional.

Eine Transition besteht in UML 2.0 aus einer gerichteten Kante von einem Ausgangszustand in einen Zielzustand, die ein *Auslöse-Ereignis* (engl. Trigger), eine Bedingung und eine Aktivität

besitzen darf. Transitionen werden mit der syntaktischen Erweiterung Trigger[Guard]/Activity versehen. In UML gibt es vier voneinander verschiedene Ereignistypen. Ein *Change Event* wird ausgelöst, sobald eine zuvor definierte Bedingung erfüllt wird. Ein *Call Event* wird durch die Ausführung einer *CallOperationAction*, also einem Operationsaufruf auf einem fremden Objekt, verursacht. Wird eine *SendSignalAction* ausgeführt, so wird ein *Signal Event* erzeugt, das an das Empfängerobjekt weitergeleitet wird. *Timed Events* erlauben es, Zeiten für eine Transition zu spezifizieren. Eine Transition schaltet, sobald das als Auslöser spezifizierte *Ereignis* auftritt und die an der Transition angegebene Bedingung erfüllt ist. Triggerless Transitionen, für die kein *Ereignis* als Auslöser spezifiziert worden ist, dürfen schalten, sobald ein sog. *Completion Event* erzeugt wurde und die spezifizierte Bedingung erfüllt ist. Ursachen für die Erzeugung eines *Completion Events* sind die Beendigung einer *Aktivität*, die im *do*-Abschnitt eines Zustandes spezifiziert worden ist oder die Beendigung der *Entry-Aktivität*, falls keine Aktivität im *do*-Abschnitt vorhanden ist. *Completion Events* werden bevorzugt ausgeliefert und verarbeitet, sodass keine Verklemmungen auftreten können. In Abbildung 3.3 ist das Statechart-Diagramm des abstrakten Reglers mit den Zuständen *init*, *wait*, *stopped*, *waitSensor*, *valuesComputed*, *waitActuator* gezeigt. Die Transition zwischen den Zuständen *waitActuator* und *wait* ist triggerless.

Das Schalten einer Transition führt dazu, dass zunächst der Ausgangszustand unter Ausführung einer eventuell vorhandenen *Exit-Aktivität* verlassen und die an der Transition spezifizierte *Aktivität* ausgeführt wird. Anschließend wird der spezifizierte Zielzustand unter Ausführung einer eventuell vorhandenen *Entry-Aktivität* betreten. Ein solcher Zustandswechsel unter Ausführung von *Aktivitäten* und ihren *Actions* wird als Run-To-Completion-Schritt (RTC-Schritt) [Dou04] bezeichnet. Der vollständige RTC-Schritt modifiziert somit den Kontrollzustand des betrachteten Objektes. Durch die *Actions* in den *Aktivitäten* können der Daten-, der Lebenszykluszustand und der Zustand der Ereigniswarteschlange verändert werden. In Abbildung 3.4 wird der RTC-Schritt zwischen den Zuständen *waitSensor* und *valuesComputed* skizziert. Sobald ein *Ereignis* auftritt, wird erstens die *Exit-Aktivität* des Zustandes *waitSensor*, die keine *Actions* enthält, abgearbeitet. Zweitens werden die *Actions* aus der Activity *compute* an der Transition ausgeführt. Hierbei schalten die *Actions lese x-Wert*, *lese y-Wert*, *berechne neuen y-Wert* und *schreibe y-Wert*, die hier mit 2.1 bis 2.4 nummeriert sind, nacheinander. Drittens findet der Wechsel in den neuen Kontrollzustand *valuesComputes* statt. Abschließend wird eine potentiell vorhandene *Entry-Aktivität* ausgeführt. Um Konflikte beim Schalten von Completion Transitions zu vermeiden, empfiehlt die UML 2.0 Spezifikation, dass die Completion Transitions, die vom gleichen Zustand ausgehen, über verschiedene Bedingungen verfügen müssen.

Pseudo- und Endzustände

Pseudo-Zustände sind Zustände für die die Verweildauer mit null definiert ist, d. h. sie werden betreten und augenblicklich wieder verlassen. Daher dürfen Pseudo-Zustände keinen do-Abschnitt besitzen, in dem eine *Aktivität* definiert worden ist.

Ein Startzustand wird durch einen schwarz ausgefüllten Kreis dargestellt und ist der Einstiegspunkt eines Statecharts. Es ist nur ein Startzustand zulässig, der nur eine ausgehende Transition besitzt, die sofort nach dem Betreten des Startzustands ausgelöst wird.

Historienzustände (engl. history states) werden zur Modellierung in geschachtelten Zuständen verwendet, um den zuletzt besuchten Unterzustand zwischenzuspeichern. Es werden tiefe (engl. deep history state) und flache (engl. shallow history state) Historienzustände unterschieden. Tiefe Historienzustände können den zuletzt besuchten Unterzustand in beliebiger Tiefe zwischenspeichern, während das Gedächtnis von flachen Historienzuständen nur bis zur obersten Verfeinerungsstufe reicht. Beim Verlassen eines geschachtelten Zustandes, für den ein Historienzustand spezifiziert worden ist, wird der zuletzt besuchte Unterzustand durch den Historienzustand festgehalten. Um den zuletzt besuchten Unterzustand eines zusammengesetzten Zustandes wieder zu betreten, wird eine Transition auf den Historienzustand modelliert. Das Schalten dieser Transition bewirkt, dass der zuletzt besuchte Zustand wieder betreten wird. In Abbildung 3.5 besitzt der Zustand *S1* einen Historienzustand, der durch eine Transition von *S4* aus betreten wird. Endzustände definieren das Ende des Verhaltens eines Statecharts. Endzustände sind keine Pseudozustände, da ein Objekt in einem Endzustand verweilen darf. Es sind mehrere Endzustände in einem Statechart

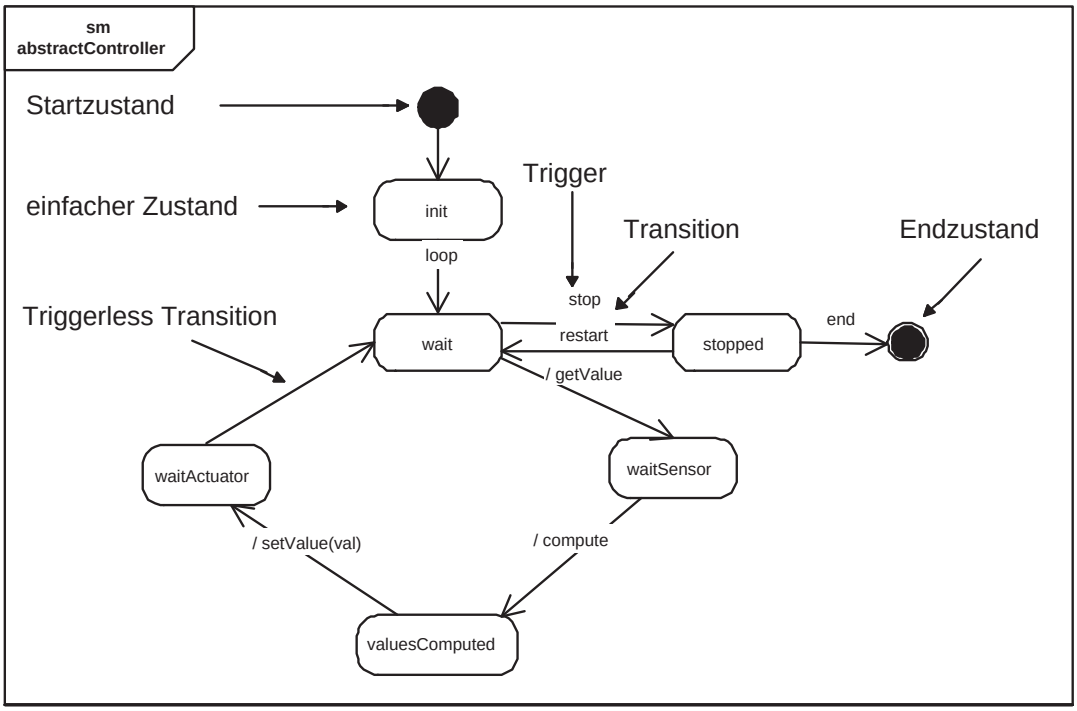


Abbildung 3.3: Das Statechart-Diagramm eines `abstractControllers`

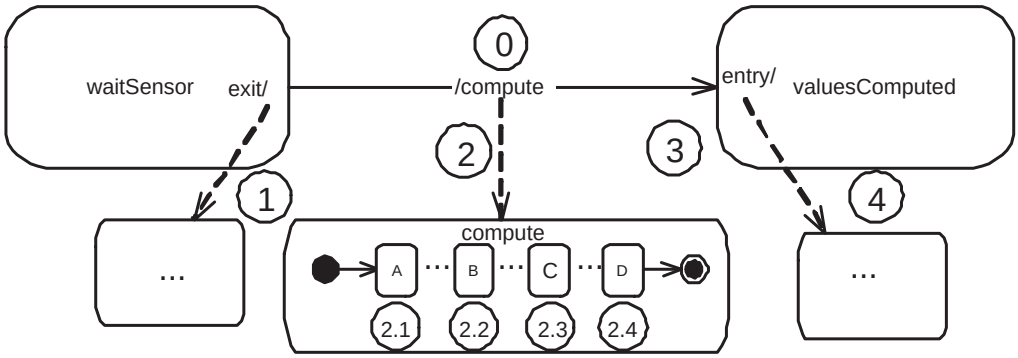


Abbildung 3.4: Ein RTC-Schritt in einem Statechart

erlaubt.

Zusammengesetzte Zustände

Zusammengesetzte Zustände (engl. composite states) gestatten die Einführung von Unterzuständen bei einem Zustand. Dadurch entstehen Hierarchien von Zuständen [Mar07]. Zusammengesetzte Zustände lassen sich in geschachtelte (engl. nested states) und nebenläufige Zustände (engl. concurrent states) unterteilen.

Ein geschachtelter Zustand besitzt eine Zustandsregion, in der Unterzustände in beliebiger Tiefe spezifiziert werden können. Geschachtelte Zustände entsprechen den ODER-Zuständen aus Statechart-Diagrammen. Wird durch eine Transition ein geschachtelter Zustand betreten, so wird gleichzeitig genau ein Unterzustand betreten. Wird ein Unterzustand durch eine Transition verlassen, so wird auch der geschachtelte Zustand, in dem er sich befindet, verlassen. Wird ein Unterzustand eines geschachtelten Zustandes betreten, dann wird auch der geschachtelte Zustand betreten.

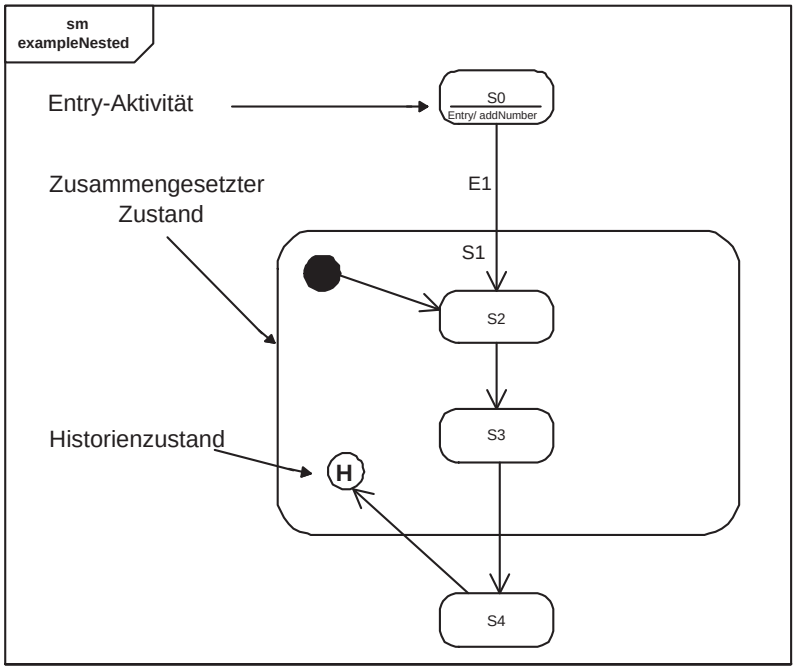


Abbildung 3.5: Ein beispielhafter geschachtelter Zustand

In Abbildung 3.5 ist der Zustand *S1* ein geschachtelter Zustand, der durch eine Transition in den Zustand *S2* betreten wird.

Nebenläufige Zustände besitzen mehr als eine Zustandsregion, in denen jeweils ein Subzustand gleichzeitig aktiv ist. Nebenläufige Zustände sind auch als UND-Zustände aus Harel Statecharts bekannt [Har87]. Beim Betreten eines nebenläufigen Zustandes durch eine Transition wird die Verarbeitung in allen Zustandsregionen zum gleichen Zeitpunkt begonnen. Das Verlassen eines nebenläufigen Zustandes durch eine Transition führt dazu, dass die Verarbeitung in allen Zustandsregionen beendet wird. Statechart-Diagramme, die keine zusammengesetzten Zustände enthalten, werden als flach bezeichnet.

3.4.4 Interaktionsdiagramme

Interaktionen werden in UML 2.0 als Sequenz- und als Kommunikationsdiagramme (früher Kollaborationsdiagramme) modelliert. Sequenzdiagramme, auf die sich diese Arbeit konzentriert, haben ihren Ursprung in den Message Sequence Charts (MSC) [Hau01]. MSCs unterstützen die Beschreibung des Zusammenwirkens mehrerer Objekte durch Szenarien. Interaktionen geben unter Berücksichtigung der zeitlichen Ordnung an, wie Objekte durch den Austausch von Nachrichten zusammenwirken. Die Darstellung wird in der Praxis durch ausführbare Beispielabläufe vorgenommen. Schwerpunktmäßig wird mit Sequenzdiagrammen somit nur das partielle Systemverhalten beschrieben. In dieser Arbeit werden auch vollständige Systemabläufe behandelt.

Interaktionen bestehen aus verschiedenen Objekten, Lebenslinien, Nachrichten und Ereignisauftritten. In Abbildung 3.6 ist ein Sequenzdiagramm angegeben, das die Interaktion zwischen den Objekten des zuvor beschriebenen abstrakten Reglers spezifiziert.

Interaktionen konzentrieren sich auf den Austausch von Informationen mittels Nachrichten zwischen Objekten. Ein Objekt besitzt einen Bezeichner und den Namen seiner Klasse.

Eine Lebenslinie repräsentiert genau einen individuellen Teilnehmer bzw. ein Objekt in einer Interaktion. Die Lebenslinie (innerhalb einer Interaktion) wählt aus dem Sequenzdiagramm nur die Ereignisauftritte aus, die ein bestimmtes Objekt betreffen. Die Anordnung von Ereignisauftritten entlang einer Lebenslinie ist signifikant, um die Reihenfolge in der diese Ereignisauftritte stattfinden, anzugeben. Jeder Ereignisauftritt wird durch eine *Action* ausgelöst, wodurch auch für

die Ausführung der *Actions* eine Ordnung innerhalb eines Objektes vorgegeben wird.

Eine Nachricht besteht aus einem Ereignisauftritt, die den Versand der Nachricht auslöst, und einem Ereignisauftritt, der den Empfang der Nachricht angibt.

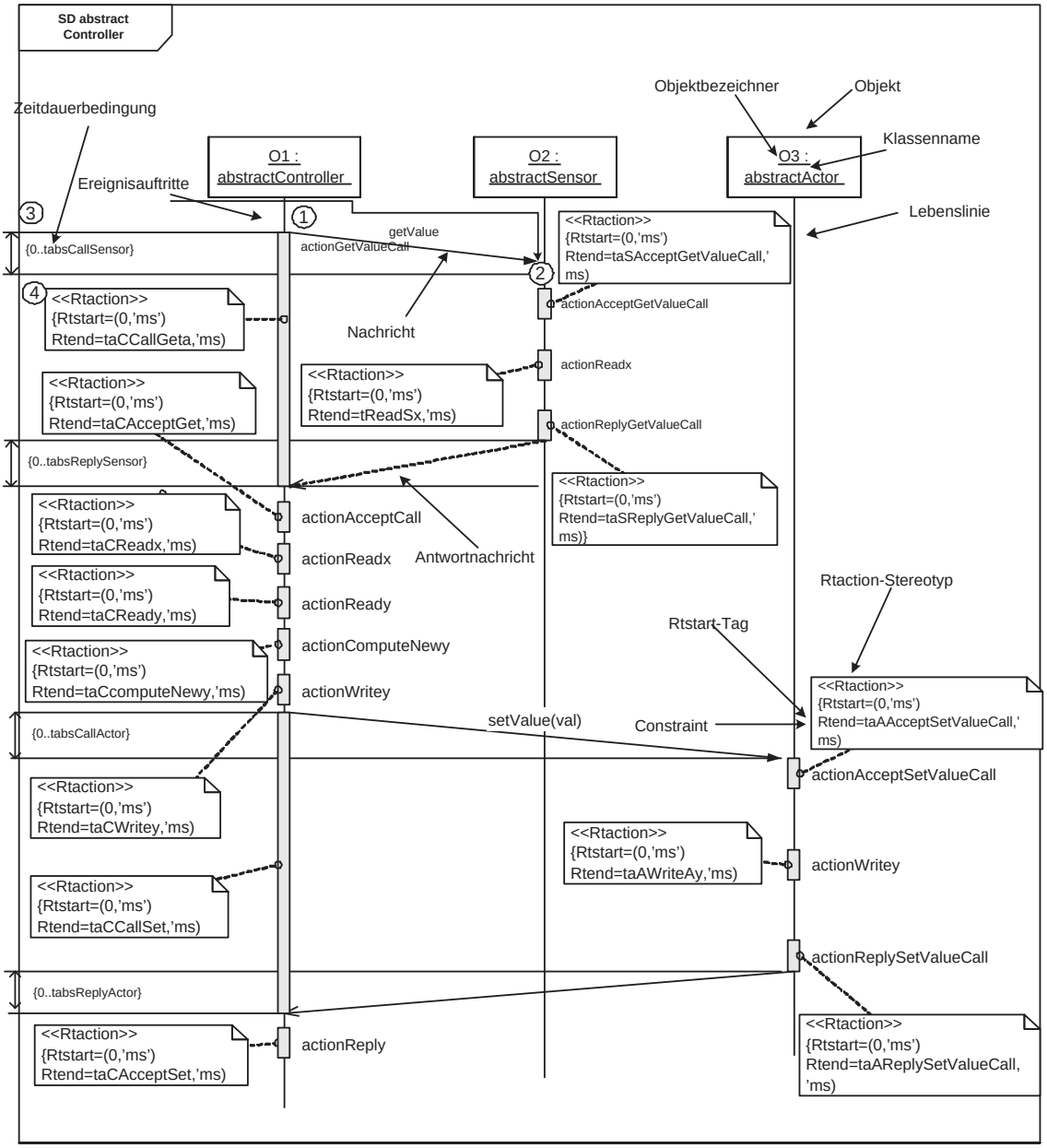


Abbildung 3.6: Ein Sequenzdiagramm eines abstrakten Reglers

Eine Nachricht, wie der *getValue*-Aufruf von *O1* nach *O2* in Abbildung 3.6 (durch Kreise mit 1 und 2 markiert), wird als eine gerichtete Kante zwischen den Lebenslinien zweier Instanzen dargestellt, wobei eine Nachricht vom Sender zum Empfänger übertragen wird. Diese Nachricht spezifiziert den Aufruf der *getValue*-Operation und den zugehörigen Beginn ihrer Ausführung.

Bei einem Operationsaufruf besitzt die Nachricht dieselben Argumente wie die aufzurufende Operation. Wenn die Nachricht einen Operationsaufruf repräsentiert, sind die Argumente der Nachricht die Werte an den Eingabepins der *CallOperationAction* auf der Lebenslinie des Senders. Diese werden als die Argumente des *CallEvents*, das als Folge des Operationsaufrufes erzeugt wird, im

Empfängerobjekt durch eine *AcceptCallAction* bereitgestellt.

Wenn eine Nachricht ein Signal repräsentiert, sind die Argumente der Nachricht die Werte der Eingabepins der *SendObjectAction* auf der Lebenslinie des Senderobjektes. Im Empfängerobjekt sind die Argumente dann im neu erzeugten *Signal Event* verfügbar und werden durch eine *AcceptEventAction* bereitgestellt.

Für den Fall, dass die Nachricht durch eine *CallOperationAction* ausgelöst wird, gibt es normalerweise eine Rückantwort (engl. reply message) vom Objekt der durch die Nachricht aufgerufenen Lebenslinie zurück zu der Lebenslinie, die den Versand der Nachricht verursacht hat. Anschließend wird die Verarbeitung auf der Lebenslinie des Senders fortgesetzt.

Es ist möglich, andere Sequenzdiagramme in einem Sequenzdiagramm zu referenzieren, wofür das Schlüsselwort *ref* verwendet wird.

Die Ereignisauftritte eines Sequenzdiagramms und deren Reihenfolge können formal durch einen Trace angegeben werden [HS03].

Ein Trace kann partiell oder total sein. Durch einen *totalen Trace* werden sämtliche Ereignisauftritte eines Sequenzdiagramms angegeben. Ein *partieller Trace* heißt auch Sub-Trace und besteht nur aus einer Teilsequenz der Ereignisauftritte des *totalen Traces* eines Sequenzdiagramms. Ein *Trace* beschreibt die Historie des Nachrichtenaustausches, der zu einem Systemablauf gehört. Der *Trace* einer einzelnen Nachricht besteht aus dem Paar, das aus den Ereignisauftritten der *CallOperationAction* und dem Ereignisauftritt der *AcceptCallAction* zusammengesetzt ist.

3.4.5 Behandlung von Zeiten in UML 2.0 und im UML RT Profile

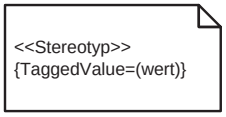
In UML 2.0 sind zusätzlich Timing-Diagramme eingeführt worden. Außerdem hat sich die Behandlung von Zeiten für Sequenzdiagramme gegenüber vorherigen UML Versionen verändert.

Zeiten in Sequenzdiagrammen

Die Behandlung von Zeiten in Sequenzdiagrammen erfolgt mit Hilfe der Object Constraint Language (OCL) durch die Angabe von OCL-Constraints, mit deren Hilfe Zeitpunkte, Zeitdauern, Zeitdauerintervalle und Zeitdauerbedingungen spezifiziert werden [JRH+03], wie in Abbildung 3.6 dargestellt. Zeitpunkte werden als kleine waagerechte Striche auf einer Lebenslinie oder als waagerechte Geraden am linken Rand des Diagramms spezifiziert, wie in Abbildung 3.6 durch die Ziffer 3 im Kreis angegeben. Zeitdauern werden als positive Zahl notiert. Zwei Zeitdauern werden durch geschweifte Klammern und zwei Auslassungspunkte verknüpft, um ein Intervall von möglichen Zeitdauern anzugeben, wie die Zeitdauer $0 \ldots t_{absCallSensor}$, die mit der Ziffer 4 im Kreis gekennzeichnet ist. Solche Intervalle dürfen in einem Sequenzdiagramm an jeder beliebigen Stelle aufgetragen werden, wodurch sie zu einer Zeitdauerbedingung werden. Speziell wird die Dauer einer Nachricht als die Zeitdauer zwischen den beiden Ereignisauftritten des Versendens und des Empfangs angegeben, wie für die *getValue*-Nachricht in Abbildung 3.6 gezeigt.

UML-Profil für Realzeit

Ein UML-Profil (engl. profile) ist ein Metamodell einer bestimmten Domäne. Um ein Profile zu definieren, wird eine zusammengehörige Menge von UML-Stereotypen verwendet, die mit Tagged Values wie folgt versehen sind: Stereotypen und Tagged Values gehören zu den Erweiterungsmecha-



nismen der UML. Ein Stereotyp ist eine Spezialisierung einer Klasse aus dem Metamodell der UML und gestattet eine leichte Anpassung an die jeweilige Domäne. Stereotypen werden typischerweise durch Verwendung von doppelten Spitzklammern [1] notiert. Tagged Values sind Eigenschaftsname-

[1] Guillemets

Wert Paare. Tagged Values werden mit Stereotypen kombiniert, um anzugeben welche zusätzlichen Informationen zu einem Stereotypen gehören. Ein Constraint ist eine benutzerdefinierte Regel, die für eine bestimmte Domäne sinnvoll ist. Um die Tagged Values eines Stereotypen mit Werten zu versehen, werden Constraints verwendet. Zusätzlich kann ein Profile eine Modellbibliothek enthalten, die zur Domäne gehört. Das sog. RT UML Profile – auch als das UML Profile for Schedulability, Performance and Time bekannt – wurde als Antwort auf einen RFP der Real-Time Analysis and Design Arbeitsgruppe der OMG eingereicht. Der Zweck des Profils besteht darin, eine gemeinsame standardisierte Art und Weise für die Spezifikation von Realzeit-Eigenschaften, wie etwa der Rechtzeitigkeit, anzugeben. In [Dou04] äußert Douglass, der Mitglied der Real-Time Analysis and Design Arbeitsgruppe ist, Folgendes zum Zweck des Profils:

> „Specifically, we wanted to be able to define a standard way for users to annotate their models with timely information and then allow performance analysis tools read those models and perform quantitative analysis on them."

Zur besseren Verständlichkeit ist das Profil in Subprofile unterteilt worden. Es gibt Subprofile zur Modellierung von Ressourcen (RTRessourceModeling), zur Nebenläufigkeit (RTConcurrency) und zur Zeitmodellierung (RTTimeModeling) in Realzeitsystemen. Weiterhin gibt es Subprofile für die Leistungsanalyse (Performance Analysis Profile) und die Analyse des Schedulings (Schedulability Analysis Profile).

Durch das Time Modeling Subprofile werden Stereotypen und Tagged Values für die Modellierung von Zeiten und der zugehörigen Konzepte angegeben. Das Subprofil erlaubt es, zeitbehaftete Informationen mit der Ausführung von *Actions* in Beziehung zu setzen, wobei das Stereotyp <<RTaction>> verwendet wird. Das Tag *RTstart* gibt die Zeit an, nach der eine *Action* gestartet wird. Diese Zeit wird hier als minimale Wartezeit interpretiert, nach deren Verstreichen die Ausführung einer *Action* beginnen muss. Mit dem Tag *RTend* wird die Zeit angegeben, nach der die *Action* abgeschlossen sein muss. Diese Zeit wird als die maximale Wartezeit interpretiert, nach der eine *Action* abgeschlossen sein muss. Alternativ ist die Verwendung des Tags *RTduration* möglich, das die Zeitdifferenz zwischen dem Tag *RTstart* und dem Tag *RTend* angibt:
$\{RTend - Rtstart < RTduration\}$

In Abbildung 3.6 ist für jede *Action* des abstrakten Reglers ein Constraint angegeben, um die minimale und die maximale Wartezeit festzulegen. So wird für die *Action lese y -Wert* das Tag *RTstart* auf 0 und das Tag *RTend* auf *tReadCx* gesetzt.

3.4.6 Zusammenwirken der einzelnen UML-Diagramme

In Abbildung 3.7 wird das Zusammenwirken der einzelnen Objekte eines Sequenzdiagramms angegeben, wobei auch die Interna eines Objektes nochmals skizziert werden.

Durch den gerade in einem Objekt aktiven Kontrollzustand und das Eintreffen eines *Ereignisses* sowie der Erfüllung einer Bedingung wird eine Transition ausgewählt. Beim Schalten dieser Transition wird deren *Aktivität* mit ihren *Actions* ausgeführt. Diese *Actions* modifizieren ihrerseits den Datenzustand, den Zustand der Ereigniswarteschlange und den Lebenszykluszustand. *Actions* sind auch das Mittel, um durch Nachrichten *Ereignisse* in die Ereigniswarteschlange eines anderen Objektes einzufügen. Damit wird auch indirekt der Kontrollzustand des anderen Objektes beeinträchtigt. Das Auftreten jeder *Action* entspricht einem Ereignisauftritt.

3.5 Softwareentwicklungsprozesse zur Erstellung korrekter Steuerungssoftware

ROPES (Rapid Object Oriented Process for Embedded Systems) ist ein bekannter Softwareentwicklungsprozess (SEP) und beschreibt die gesamte Projektentwicklung durch einen iterativen Softwareprozess [Dou99]. Die typischen Entwicklungsphasen Analyse, Design, Implementierung und Test werden in einzelne Teilphasen gegliedert, wie in Abbildung 3.8 gezeigt. In jeder Teilphase

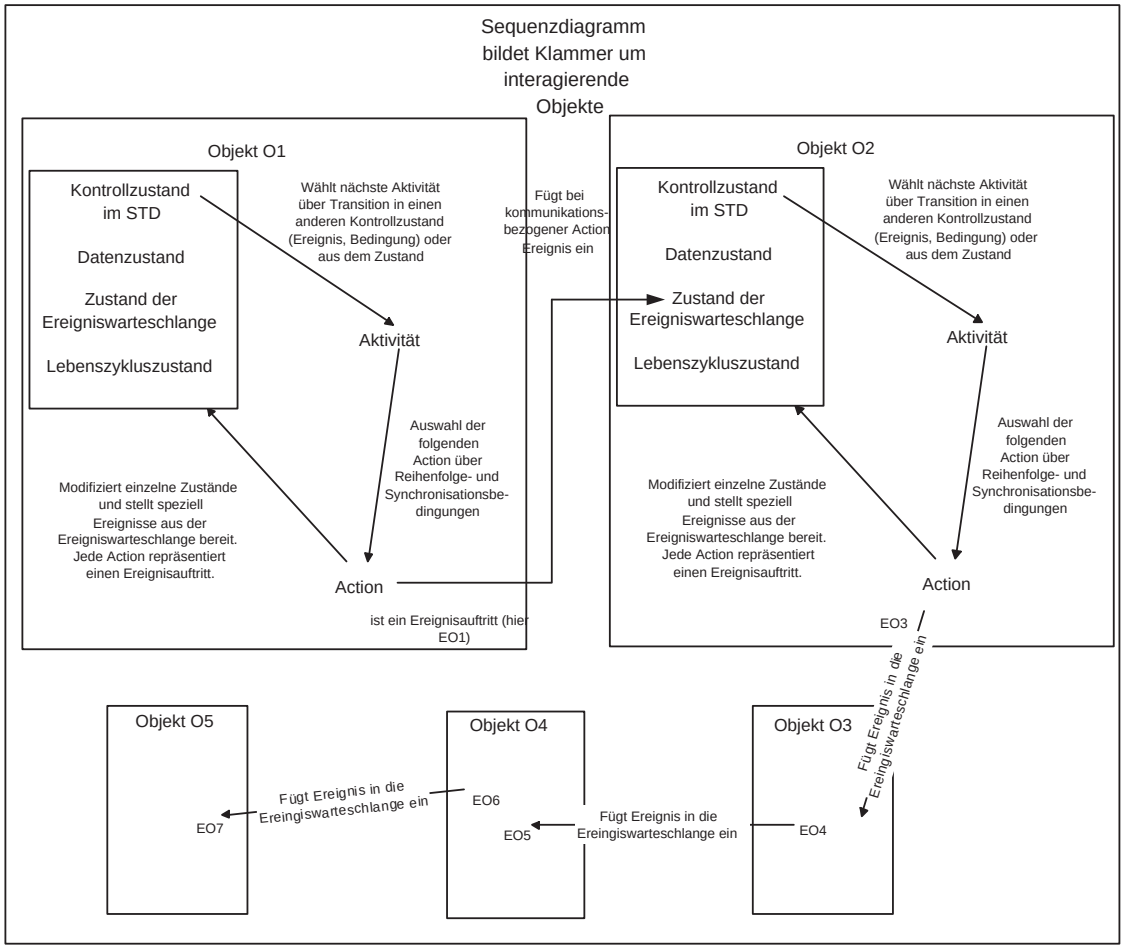


Abbildung 3.7: Das Zusammenwirken der UML-Diagramme

werden konkrete Arbeitsergebnisse erstellt, die als Artefakte nach deren Beendigung vorliegen. Die Artefakte der Phasen und Teilphasen werden im Wesentlichen durch UML-Diagramme beschrieben.

Die wesentlichen Artefakte sind das Analyse-Objektmodell, das Design-Objektmodell, die Anwendungskomponenten sowie die getestete Anwendung. Darüber hinaus wird sichtbar, dass auch weitere Artefakte auf die einzelnen Aktivitäten Einfluss haben. Alle Phasen von ROPES und deren Artefakte werden in [Dou99, Dou04] ausführlich behandelt und sollen hier nur kurz vorgestellt werden. Hier wird ein Schwerpunkt auf die Phase der Analyse und des Designs gelegt, da diese Phasen für die Entwicklung von Verfeinerungsmustern besonders wichtig sind.

Die Analyse dient der Identifikation der essentiellen Eigenschaften eines Projekts, wobei alle notwendigen Bedingungen zusammengetragen werden, die jede potentielle Implementierung erfüllen muss. Dabei wird nicht auf eine bestimmte Lösung hingearbeitet. Nach ROPES unterteilt sich die Analyse in die Teilphasen Anforderungsanalyse, Systemanalyse und Objektanalyse, von denen im Folgenden die Objektanalyse detaillierter betrachtet wird.

Bei der Anforderungsanalyse werden die Kundenvorgaben über das Verhalten und die Ziele eines Systems gesammelt und für die Objektanalyse zur Verfügung gestellt. Als Ergebnis der Anforderungsanalyse werden Use Cases (Anwendungsfälle) erstellt. In der Anforderungsanalyse werden weiterhin die besonders im Bereich der Steuerungssoftware wichtigen zeitlichen Bedingungen an das Gesamtsystem und die Schnittstellen zu anderen Systemen definiert. Auch Hazards werden ermittelt, die einen Zustand bzw. eine Menge von Bedingungen eines Systems und seiner Umgebung beschreiben, die unausweichlich zu einem Unfall – etwa durch Freisetzung von Energie oder gefährlichen Stoffen – führen.

Die Systemanalyse konzentriert sich nur auf funktionale Einheiten und nicht etwa auf später zu programmierende Objekte oder Klassen. Aufwendige Systeme werden in große Organisationseinheiten aufgeteilt und komplexe Spezifikationen des Verhaltens der Einheiten erzeugt und analysiert. Weiterhin erfolgt dadurch eine Aufteilung des Systems in einen elektronischen, einen mechanischen sowie einen softwaretechnischen Bereich sowie die Definition der Schnittstellen zwischen den Bereichen.

Das Ziel der Objektanalyse ist es die innere Struktur eines Systems herauszubilden. Dabei werden Objekte und Klassen sowie deren Beziehungen untereinander definiert und gegebenenfalls zu Domänen, d. h. eigenständigen Bereichen des Systems zusammengefasst. Die Anwendungsfälle werden herangezogen, um die Methoden und Attribute der Klassen zu ermitteln und die wichtigsten Verhaltensweisen anzugeben. Die gefundenen Mechanismen können mittels der Szenarien auf Korrektheit getestet werden. Um Aussagen über das Objektverhalten machen zu können, sind evtl. vorhandene Statecharts auf die Klassen aufzuspalten. Außerdem werden den einzelnen Klassen die zeitlichen Bedingungen zugeordnet. Das Ergebnis dieser Teilphase ist das Analyse-Objektmodell, das alle wichtigen Charakteristika einer korrekten Lösung identifiziert.

In der Designphase steht die konkrete Umsetzung im Vordergrund. Erst in dieser Teilphase wird definiert, wie bestimmte Bestandteile entworfen und für die jeweiligen Arbeitsbedingungen optimiert werden. Bei ROPES wird laut [Dou99, Dou04] das Design in die drei Teilphasen Architekturdesign, mechanistisches Design und detailliertes Design zerlegt. In jeder dieser Teilphasen wird ein bestimmter Bestandteil des gesamten Design-Objektmodells erstellt.

Beim Architekturdesign werden generelle Eigenschaften, die für die meisten Teile des Systems gelten, spezifiziert und Architekturmuster zur Fehlerbehandlung, Sicherheitsverarbeitung und Fehlertoleranz angegeben. Ferner erfolgt die Identifikation der Nebenläufigkeiten eines Systems durch die Charakterisierung einzelner Threads.

Beim mechanistischen Design werden Verfeinerungen der Kooperation zwischen den Objekten vorgenommen. Zu diesem Zweck werden dem Designmodell notwendige neue Objekte hinzugefügt. Zum Beispiel benötigt ein Controller, der viele ankommende Nachrichten mittels des Collection-Iterator-Patterns verarbeiten soll, ein zusätzliches Collection- und ein Iterator-Objekt.

Die unterste Ebene des Designs ist das detaillierte Design, welches sich den internen Strukturen und dem Verhalten der einzelnen Klassen zuwendet.

Das detaillierte Design beinhaltet die Spezifikationen für die Invarianten mit Vor- und Nachbedingungen der einzelnen Operationen, die genauen Typen und die Gültigkeitsbereiche von Attributen sowie die Behandlung von Ausnahmen. Es ist möglich, komplizierte Algorithmen als Pseudocode innerhalb des Designmodells zu vermerken. Als Ergebnis erhält man das Design-Objektmodell, das eine spezielle und geeignete Lösung definiert.

Bei der Implementierung wird das erstellte UML-Modell aus der Designphase in Quellcode für ein ausführbares Programm übersetzt. Dieser Vorgang erfolgt routiniert, da in der Designphase exakte Vorgaben gemacht wurden. Die Implementierung kann manuell oder auf der Basis von generatorbasierten Techniken – wie etwa MDA [KWB03] – erfolgen.

Besonders effizient erfolgt die Implementierung, wenn die Zielsprache objektorientiert ist. Die Verwendung einer nicht objektorientierten Zielsprache verursacht in der Regel einen Mehraufwand.

Die letzte Phase der Entwicklung ist die Testphase zur Fehlerbehebung. Bestimmte Testfälle, die nach außen hin sichtbare Ergebnisse erzeugen und sich in erster Linie aus den Szenarien der Analysephase ergeben, kommen in einem Test bei der Applikation zur Anwendung.

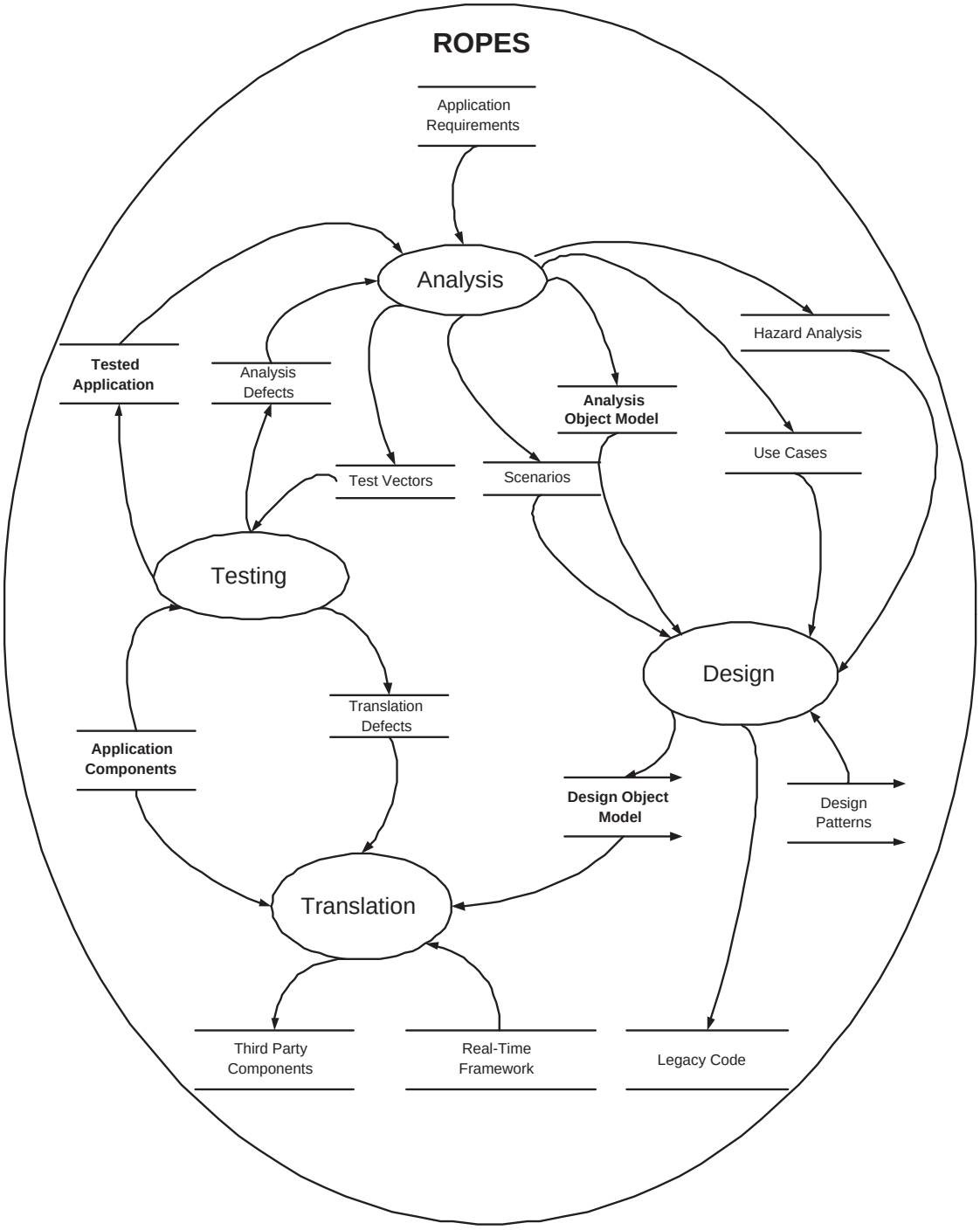


Abbildung 3.8: ROPES (entnommen aus [Dou99])

Kapitel 4

Muster

Muster sind seit vielen Jahren ein bedeutender Trend der objektorientierten Softwaretechnik. Ausgehend von der Phase des Entwurfs bei dem sog. Entwurfsmuster – hier hat insbesondere die Gang of Four [1] grundlegende Arbeit geleistet – eine große Rolle spielen [GHJV93], hat sich dieser Trend auf viele Gebiete der Softwareerstellung ausgeweitet. Gegenwärtig sind Entwurfsmuster für Realzeitsysteme Gegenstand der Forschung [SSRB00, BHS07b, DG08]. Außerdem haben auch Muster für die Analysephase im Softwareentwicklungsprozess, die zunächst von M. Fowler als Analysemuster vorgestellt worden sind [Fow99], deutlich an Popularität gewonnen. Schließlich hat B. P. Douglass Verhaltensmuster für objektorientierte Realzeitsysteme entwickelt [Dou99]. In diesem Kapitel werden der Stand der Forschung zu Mustern sowie deren Terminologie und Beziehungen aus der bestehenden Literatur vorgestellt. Dazu wird zunächst eine Klassifikation des Musterbegriffs vorgenommen. Anschließend wird dargestellt, wie Muster in dieser Arbeit - speziell unter Verwendung der UML - beschrieben werden. Abschließend werden die Begriffe des Musterkatalogs, des Mustersystems und der Mustersprache ausführlich erörtert.

4.1 Klassifikation von Mustern

Im Folgenden soll der Begriff des Musters genauer untersucht werden und einige Definitionen aus der Literatur vorgestellt werden, welche in Tabelle 4.1 mit ihrer Quelle angegeben sind. Zunächst wird der Begriff des Musters [RZ96] im Allgemeinen betrachtet. Danach besteht ein Muster aus der Abstraktion von einer bestimmten Form, welche wiederholt in bestimmten und nicht willkürlichen Kontexten auftritt (s. Definition 1 in Tabelle 4.1). Diese Definition weist einen höheren Freiheitsgrad auf als andere in der Literatur vorkommende Definitionen. Die ursprüngliche Musterdefinition nach Alexander [AIS+77] sieht ein Muster als eine zusammengesetzte dreiteilige Regel, die eine Beziehung zwischen einem bestimmten Kontext, einem Problem und einer Lösung angibt (s. Definition 2 in Tabelle 4.1). Hier tauchen die drei Begriffe Kontext, Problem und Lösung auf. Diese Begriffe werden in [Lea00] von Doug Lea folgendermaßen erklärt:

- Der Begriff des Kontextes bezieht sich auf eine wiederkehrende Menge von Situationen, in denen ein Muster anwendbar ist.
- Der Begriff des Problems bezieht sich auf eine Menge von Kräften, Zielen und Bedingungen, die in diesem Kontext auftreten.
- Die Lösung bezieht sich auf eine kanonische Form des Entwurfs oder eine Entwurfsregel, die man anwenden kann, um diese Kräfte umzusetzen.

Unter Kraft wird die allgemeine Art von Kriterium verstanden, die Software-Ingenieure verwenden, um ihre Entwürfe und Implementierungen zu rechtfertigen. Muster lösen auch eine konfliktbehaftete Menge von Zielen und Bedingungen, die in jedem Artefakt, das man erschafft, aufeinander

[1] Die Gruppe bestehend aus E. Gamma, R. Helm, R. Johnson und J. Vlissides

Begriff	Definition
1. Muster allgemein [RZ96]	Ein Muster ist die Abstraktion von einer konkreten Form, die wiederholt in einem spezifischen, nicht willkürlichen Kontext auftritt.
2. Muster allgemein [AIS+77]	Jedes Muster ist eine dreiteilige Regel, die eine Beziehung zwischen einem bestimmten Kontext, einem Problem und einer Lösung ausdrückt.
3. Form [RZ96]	Die Form eines Musters besteht aus einer Anzahl von sichtbaren und unterscheidbaren Komponenten und deren Beziehungen.
4. Konzeptionelles Muster [RZ96]	Ein konzeptionelles Muster (Analysemuster) ist ein Muster, dessen Form durch die Begriffe und die Konzepte einer Anwendungsdomäne beschrieben wird.
5. Analysemuster [Fow99]	Analysemuster[1] sind Gruppen von Konzepten, die eine gebräuchliche Struktur in der Geschäftsmodellierung darstellen. Diese können entweder für nur genau einen Bereich relevant sein oder auch mehrere Bereiche umfassen.
6. Architekturmuster [BMR+98]	Ein Architekturmuster spiegelt ein grundsätzliches Strukturierungsprinzip von Softwaresystemen wider. Es beschreibt eine Menge vordefinierter Subsysteme, spezifiziert deren jeweiligen Zuständigkeitsbereich und enthält Regeln zur Organisation der Beziehungen zwischen den Subsystemen.
7. Entwurfsmuster [GHJV93]	Ein Entwurfsmuster beschreibt ein Schema zur Verfeinerung von Subsystemen oder Komponenten eines Softwaresystems oder den Beziehungen zwischen ihnen. Es beschreibt eine häufig auftretende Struktur (und das zugehörige Verhalten) von miteinander kommunizierenden Komponenten, die ein allgemeines Entwurfsproblem in einem speziellen Kontext lösen.
8. Verhaltensmuster [Dou99]	Verhaltensmuster strukturieren keine Objekte, sondern Zustände und Transitionen. Trotzdem lassen sich die gleichen Prinzipien, wie bei den Objekten anwenden. Verhaltensmuster sind Sammlungen von Zuständen und Zustandsübergängen, die allgemein genug sind, um ein oder mehrere Analyse- bzw. Entwurfsprobleme zu lösen.
9. Interaktionsmuster [GB98] UML-Metamodell	Bei der Sicht von außen wirkt ein Interaktions-/ Entwurfsmuster wie eine parametrisierte Kollaboration. Als eine Kollaboration – also dem Kontext für eine Menge von Interaktionen – stellt ein Muster eine Menge von Abstraktionen dar, deren Struktur und deren Verhalten zusammenwirken, um eine nützliche Funktion auszuführen.

Tabelle 4.1: Die Definitionen von Begriffen aus dem Musterumfeld.

[1] Analysemuster werden bei der objektorientierten Analyse während der Softwareentwicklung eingesetzt.

treffen. Diese Kräfte stammen beispielsweise aus den Bereichen Korrektheit, Ressourcen, Struktur, Konstruktion und Benutzung. Wenn man den Begriff der Korrektheit eines Musters genauer untersucht, stößt man in der Literatur schließlich auf folgende Aspekte [Lea00]:

- Vollständigkeit und Korrektheit einer Lösung
- statische und dynamische Typsicherheit
- Multithreaded Safety und Liveness
- Fehlertoleranz und Transaktionsfähigkeit
- Sicherheit und Robustheit

Im Weiteren soll kurz der Begriff der Form eines Musters nach [RZ96] genauer analysiert werden (s. Definition 3 in Tabelle 4.1). Der Begriff der Form eines Musters wird durch seine Repräsentation aus einer Menge unterscheidbarer Komponenten und deren Beziehungen definiert. Damit hat ein Muster sowohl strukturelle als auch Verhaltensaspekte. Weiterhin kann nach [RZ96] die Form eines Musters formalisiert werden, aber nicht dessen Kontext. Bei Mustern unterscheidet man je nach zugehörigem Abstraktionsniveau und der Entwicklungsphase des Softwareentwicklungsprozesses, in der sie verwendet werden, zwischen unterschiedlichen Arten. Andere Unterscheidungsmöglichkeiten – wie z. B. nach der Musterkörnung und der Musterbeschreibung – werden in [QC99] aufgeführt. In Tabelle 4.2 wird eine an entwicklungsphasenorientierte Klassifikation von Mustern, die sich stark nach [RZ96] richtet, angegeben. Die Tabelle gibt die Muster aus den drei wichtigsten Phasen des objektorientierten Softwareentwicklungsprozesses – nämlich der Anforderungsanalyse, dem Entwurf und der Implementierung – wieder. Durch die Spalten wird zwischen strukturellen und verhaltensorientierten Aspekten eines Musters unterschieden.

Entwicklungsphase der Software	Struktur eines Musters	Internes Verhalten einer Musterklasse
Anforderungsanalyse	Analysemuster bzw. konzeptionelle Muster	Verhaltensmuster einzelner Klassen von Analysemustern
Entwurf	Architektur & Entwurfsmuster (Diese gliedern sich nach [GHJV93] in structural, behavioral u. creational.)	Muster, die das Verhalten einzelner Klassen von Entwurfsmustern beschreiben.
Implementierung	Idiome (im Text erklärt) z. B. zur Erzeugung u. Zerstörung von Objekten	Verhaltenbeschreibung von Implementationsklassen

Tabelle 4.2: Die Klassifikation von Mustern nach Phasen des Entwicklungsprozesses

Nach dieser Klassifizierung von Mustern wird nun die Anforderungsanalyse betrachtet. Um etwas Sinnvolles zu entwerfen, ist ein konzeptionelles Modell der Anwendungsdomäne notwendig, an dem sich die Weiterentwicklung einer Software orientiert [Jac82]. Dafür ist es zunächst nicht erforderlich, dass das Modell der Anwendungsdomäne formal ist. Für gewöhnlich besteht ein solches Modell aus einer Menge von Spezifikationen. Zwischen diesen Spezifikationen bestehen Beziehungen, welche auf den Konzepten und Begriffen der zugehörigen Anwendungsdomäne beruhen, d. h. es sollte die Sprache der Anwendungsdomäne und der an der Entwicklung beteiligten Personen verwendet werden. Muster sollten die Sprache des Anwendungsfeldes mit den Begriffen und Bestandteilen, die in dem konzeptionellen Modell der Anwendungsdomäne Verwendung finden, in Beziehung setzen. Es werden daher in [RZ96] konzeptionelle Muster eingeführt (s. Definition 4 in Tabelle 4.1), deren Form durch die Begriffe und Konzepte einer Anwendungsdomäne vorgegeben werden. Fowler [Fow99] definiert erstmals Analysemuster für die objektorientierte Modellierung betriebswirtschaftlicher Software (s. Definition 5 in Tabelle 4.1), die der Idee wie konzeptionelle Muster für die Domäne der Geschäftssoftware entsprechen. Konzeptionelle Muster beeinflussen die

Wahrnehmung des zugehörigen Anwendungsgebietes. Deshalb sollten konzeptionelle Muster auf ein beschränktes Anwendungsgebiet ausgerichtet sein. Eine besondere Herausforderung besteht darin, ein geeignetes Abstraktionsniveau zu finden. In dieser Arbeit soll der Begriff Analysemuster weiterverwendet werden.

Auch beim Entwurf von Software finden Muster Anwendung. Dies sind Entwurfs- und Architekturmuster. Entwurfsmuster werden originär von Beck und Cunningham [BC87] sowie in Gamma's Dissertation vorgestellt. Das erste Buch [GHJV93] zum Thema der Entwurfsmuster wurde von der legendären Viererbande (Gang of Four) publiziert. In [GHJV93], [BMR+96] und [SSRB00] finden sich Definitionen für diese Arten von Mustern. Architekturmuster sind Schablonen für konkrete Softwarearchitekturen (s. Definition 6 in Tabelle 4.1). Durch sie werden systemweite, strukturelle Eigenschaften einer Anwendung vorgegeben und beeinflussen die Architektur der Subsysteme. Architekturmuster beeinflussen im Rahmen von Grundsatzentscheidungen den Entwurf eines Softwaresystems. Ihre Auswahl findet in der Phase des Grobentwurfs statt. Entwurfsmuster hingegen sind Muster, die sich auf einem mittleren Abstraktionsniveau befinden (s. Definition 7 in Tabelle 4.1). Ihr Anteil am Umfang innerhalb der Architektur eines Softwaresystems, den sie beeinflussen, ist geringer, weil sie während des Feinentwurfs verwendet werden. Sie sind aber im Allgemeinen unabhängig von einer speziellen Programmiersprache und einem speziellen Programmierparadigma. Wird ein Entwurfsmuster angewendet, so führt dies zu keinen Auswirkungen auf die grundsätzliche Struktur des Softwaresystems, kann aber von großer Bedeutung für die Architektur eines Subsystems sein. Entwurfsmuster sind in dieser Arbeit später von großer Bedeutung. Bei den Entwurfsmustern hat Gamma [GHJV93] eine detailliertere Gliederung in Erzeugungsmuster (creational patterns), strukturelle Muster (structural patterns) und verhaltensorientierte Muster (behavioral patterns) vorgenommen. Hierbei gestatten Erzeugungsmuster es technische Details vom Instantiierungsvorgang zu abstrahieren. Wenn man ein komplexes, objektorientiertes System entwirft, muss man sich auf zusammengesetzte Objekte, die klassenbasierte Vererbung besitzen, konzentrieren. Erzeugungsmuster werden verwendet, um die Instantiierung zu delegieren bzw. vom Verhalten zu abstrahieren und Instantiierungs– sowie Kompositionsdetails zu verbergen.

Strukturelle Muster konzentrieren sich auf die Komposition von Klassen und Objekten zu größeren Strukturen. Sie behandeln Laufzeitkompositionen, die einen dynamischeren Charakter als mehrfache Vererbung besitzen, Object sharing sowie Schnittstellenanpassung und das dynamische Hinzufügen von Verantwortlichkeiten zu Objekten. Verhaltensorientierte Muster behandeln die Verkapselung von Algorithmen sowie die Verwaltung oder Delegation der Verantwortlichkeiten zwischen Objekten. Sie konzentrieren sich mehr auf die Kommunikation und die Interaktion von Objekten, die dynamischen Schnittstellen, die Objektkomposition und die Objektabhängigkeiten. Im Bereich der Entwurfsmuster für objektorientierte Systeme hat Douglass [Dou99, Dou04] zahlreiche Publikationen erstellt.

Da neben strukturellen Aspekten auch Verhaltensaspekte durch Muster abgedeckt können werden sollen, ist es nahe liegend, auch Verhaltensmuster [Dou99] zu definieren (s. Definition 8 in Tabelle 4.1). Verhaltensmuster strukturieren das Verhalten von Objekten als Sammlungen von Zuständen und deren Transitionen. Wenn diese von allgemeiner Natur sind, lassen sich durch ihre Anwendung Probleme bei der Analyse und beim Entwurf von Software lösen. Ein Muster, das Verhaltensmuster einzelner Klassen kombiniert, erzwingt bestimmte Interaktionen.

Schließlich kann auch die Interaktion von Objekten beim Entwurf und zur Laufzeit in Muster gefasst werden. Da Diagramme zur Darstellung von Interaktionen noch relativ neu sind und erst durch die UML an Popularität gewinnen, hat sich noch keine allgemein akzeptierte Bezeichnung etabliert. Im Folgenden sollen solche Muster als Interaktionsmuster [Esk99] bezeichnet werden, die beispielsweise durch ein Sequenzdiagramm dargestellt werden können. In Interaktionsmustern werden also strukturelle Aspekte mit Verhaltensaspekten vereint. Für die Implementierung existieren ebenfalls Muster, die als Implementierungsmuster bzw. Idiome bezeichnet werden. Idiome sind sich wiederholende Konstrukte aus verschiedenen Programmiersprachen, die etwa eine Aufgabe bzw. einen kleinen Algorithmus repräsentieren. Sie sind kein Bestandteil einer speziellen Programmiersprache. Beispiele für Idiome sind Endlosschleifen, Inkrementierungen von Zählern oder ein Swap zwischen zwei Variablen. Idiome spielen in dieser Arbeit aber eine untergeordnete Rolle. Weiterhin gilt nach [RZ96]:

„Conceptual patterns logically precede design patterns which logically precede programming patterns. “

Dies stimmt auch mit der Orientierung an den Entwicklungsphasen aus Tabelle 4.2 und der zugehörigen Abstraktion überein. Doug Lea weist in [Lea00] außerdem darauf hin, dass zwischen Mustern und einigen Gebieten der modernen Softwaretechnik – wie domänenspezifische Architekturen und Wiederverwendung bei der Softwareentwicklung – ein enger Zusammenhang besteht.

4.2 Musterbeschreibung

Die Beschreibung von Mustern ist stark durch die Entwicklung der für die Spezifikation zur Verfügung stehenden Sprachen geprägt worden. Die ursprüngliche bei Alexander verwendete Form beschreibt ein Muster in Textform. Gemäß seiner oben genannten Musterdefinition umfasst diese einen Namen und jeweils einen Abschnitt, in dem der Kontext, das Problem und die Lösung erläutert werden.

Für das Reglermuster, das eine Lösung für ein typisches Regelungsproblem ist, werden diese Abschnitte hier angegeben:

- **Name**: Reglermuster
- **Kontext**: In technischen Systemen wird die Einhaltung bestimmter Zustände benötigt, um die Sicherheit und Effektivität einer Anlage zu gewährleisten.
- **Problem**: Istwerte eines technischen Systems müssen mit vorgegebenen Sollwerten verglichen und korrigiert werden, um bestimmte Zustände einzuhalten.
- **Lösung**: Durch das Reglermuster werden Sensorobjekte und Aktorobjekte mit einem Reglerobjekt in Beziehung gesetzt. Durch das Reglerobjekt können bestimmte Regelstrategien realisiert werden, um die Regeldifferenz möglichst klein zu halten. Stellgrößen werden an sog. Aktorobjekte weitergegeben, um die Sollwerte einzuhalten.

Quibeldey-Cirkel stellt in [QC99] vor, wie hypertextartige Dokumente zur Musterbeschreibung verwendet werden können. Als Hypertextsprachen finden hier HTML und XML Verwendung. Durch Buschmann werden in [GHJV93] außerdem auch Klassendiagramme verwendet. Die von Buschmann mitentwickelten Interaktionsdiagramme treten zum ersten Mal in [BMR+96] in Form von Sequenzdiagrammen zur Musterbeschreibung auf. Die UML setzt zur Musterbeschreibung ursprünglich auf Klassendiagramme. In dieser Arbeit soll das Konzept von Buschmann [BMR+96] der Anwendung von Sequenzdiagrammen zur Spezifikation von Interaktionsmustern eingesetzt werden. Deshalb sollen die zu einer Kollaboration von Objekten gehörigen typischen Interaktionen, wie auch schon bei Buschmann durch Sequenzdiagramme geschehen, spezifiziert werden. Es ist klar, dass typische Interaktionen nicht immer vollständig sein können. Zusätzlich sollen Analyse- und Entwurfsmuster auf die gleiche Art und Weise dargestellt werden. Um die strukturellen Bestandteile der Lösung, die ein Muster liefert, zu beschreiben werden Klassendiagramme eingesetzt; während für die dynamische Interaktion zwischen den Instanzen der Lösung eines Musters Interaktionsdiagramme – genauer Sequenzdiagramme – verwendet werden. Im Folgenden wird die Beschreibung mittels dieser Diagramme detaillierter erläutert. Klassendiagramme geben eine kompakte Übersicht über die Strukturen eines Musters. In den Klassendiagrammen werden die Klassen eines Musters ebenso, wie die Vererbungs-, Assoziations+, Aggregations- und Kompositionsbeziehungen sowie die Rollen von Objekten angegeben. Außerdem werden für die entsprechenden Beziehungen Multiplizitäten modelliert. Am Beispiel des Musters in Abbildung 3.1 erkennt man ein Klassendiagramm, das zur Spezifikation des Reglermusters verwendet wird. Es besteht aus den Klassen *abstractController*, *abstractActor*, *abstractSensor*. Die Klasse *abstractController* enthält die Operationen *loop()* und *compute()*, um Verantwortlichkeiten für die Klasse anzugeben. Diese Operationen sollen der Reihe nach kurz vorgestellt werden:

- *loop()* repräsentiert die Regelschleife, die kontinuierlich durchlaufen wird.

- *compute()* behandelt die Berechnung des neuen Stellwertes.

Die Klasse *abstractActor* stellt die abstrakte Form eines Aktors dar. Ein abstrakter Aktor verfügt über die abstrakte Operation *setValue()*, mit der der Wert des Aktors gesetzt werden kann. Die Klasse *abstractSensor* ist ähnlich aufgebaut, sie besitzt eine Operation *getValue()*, mit der Werte aus einem abstraktem Sensor ausgelesen werden können. Weiterhin können in einem abstraktem Sensor die vom Sensor eingelesenen Werte mit dem Attribut x festgehalten werden. Abstrakte Regler, wie die der Klasse *abstractController*, stehen mit abstrakten Aktoren und Sensoren in Beziehung, was durch Assoziationen im Klassendiagramm ausgedrückt wird. Weiterhin können bei der Musterbeschreibung Beziehungen mit Rollen versehen werden. Klassen können mit Stereotypen bzw. Eigenschaften attributiert werden. Dynamische Aspekte werden durch Interaktionen mit Hilfe von Sequenzdiagrammen spezifiziert.

Zur Beschreibung von Verhaltensmustern (auch Zustandsmuster genannt), die das Verhalten von Objekten strukturieren, wird eine an Statechart-Diagrammen der UML angelehnte Notation verwendet, die auch in [Dou99] Anwendung fand. Abbildung 3.3 zeigt das UML-Statechart zur Klasse *abstractController*, das aus mehreren Zuständen, die in einem Kreis angeordnet sind, aufgebaut ist.

4.3 Mustersammlungen und Mustersysteme

Muster, wie sie oben vorgestellt wurden, stehen nicht für sich allein, sondern treten in Sammlungen auf. Musterkataloge, Mustersysteme und Mustersprachen stellen Möglichkeiten dar, Muster für einen Bereich bzw. eine Wissensdomäne zu kollektionieren.

Zunächst werden Musterkataloge kurz betrachtet. In einem Musterkatalog werden Muster verhältnismäßig lose gesammelt (s. Definition 1 in Tabelle 4.1). Ein Musterkatalog fügt einen kleinen Teil von Struktur und Organisation zu einer Mustersammlung zusammen, geht aber gewöhnlich nicht weit darüber hinaus, nur die offensichtlichen Strukturen und Beziehungen aufzuzeigen, wenn er überhaupt darauf eingeht.

In einem Mustersystem sind neben der Ansammlung von Mustern auch der Sachverhalt, wie sie sich gegenseitig ergänzen, von Interesse. Das erste Mustersystem wurde von Buschmann et. al. in [BMR+96] vorgestellt. Diese Abhängigkeiten und die gegenseitige Ergänzung werden in Form von Beziehungen zwischen den einzelnen Mustern eines Mustersystems angegeben. Ein Mustersystem hat also einen klaren Schwerpunkt bei der Verbindung bzw. Verknüpfung von Mustern. In [BMR+96] wird die Definition eines Mustersystems zunächst für Softwarearchitekturen angegeben (s. Definition 2 in Tabelle 4.1). Neben der Sammlung von Mustern für Software-Architektur werden auch Richtlinien für ihre Implementierung, ihre Kombination sowie ihre praktische Verwendung in der Software-Entwicklung erläutert.

Nach [BMR+96] versteht man unter einem Mustersystem eine zusammenhängende Menge von Mustern, die in Beziehung stehen, um die Konstruktion und Evolution von vollständigen Architekturen zu unterstützen. Es ist nicht nur durch in Beziehung stehende Gruppen und Untergruppen auf verschiedenen Granularitätsebenen organisiert, sondern es beschreibt auch die vielen Beziehungen zwischen Mustern und deren Gruppierungen und wie sie kombiniert und zusammengesetzt werden können, um komplexere Probleme zu lösen. Die Muster in einem Mustersystem sollten in einem konsistenten und zusammenhängendem Stil beschrieben werden und sollten eine ausreichend breite Grundmenge von Problemen und Lösungen überdecken, um zu ermöglichen entscheidende Teile von Architekturen zu bauen. Ein Hauptinteresse für die Verwendung von Mustersystemen liegt in der Unterstützung der Entwicklung von Softwaresystemen, die eine hohe Qualität besitzen.

Um das Ziel zu erreichen, das ein Mustersystem auch Richtlinien für die Implementierung und Kombination von Mustern enthalten muss, sollte ein Mustersystem nach [BMR+96] den folgenden Anforderungen – die um Verifikationsaspekte erweitert wurden, um im Rahmen dieser Arbeit zu gelten – genügen:

- Die Anzahl der Muster sollte ausreichend groß sein, um die notwendige Anzahl von Kombinationen für die anforderungsgerechte Erstellung einer Software zur Verfügung zu stellen.

- Alle Muster sollten auf einheitliche Art und Weise beschrieben werden. Die Struktur dieser Beschreibung muss sowohl die Kernaussage eines Musters umfassen als auch eine präzise Aussage seiner Details ermöglichen.
- Die vielfältigen Beziehungen zwischen den Mustern sollten aufzeigt werden. Für jedes Muster muss das Mustersystem beispielsweise klarstellen, welche anderen Muster es verfeinert, welche es offen legt, mit welchen Mustern es kombiniert werden kann, welche Benutzungsbeziehungen (Uses-Relation) bestehen und welche Alternativen es zu ihm gibt. Dies sollte auch für die Beziehungen zwischen Mustern unterschiedlicher Typen gelten.
- Seine Bestandteile sollten geeignet angeordnet sein. Ein Benutzer sollte in der Lage, sein schnell das Muster zu finden, das ihm bei seinen konkreten Analyse- und Entwurfsproblemen hilft. Außerdem sollte ein Benutzer alternative Lösungen für sein Problem finden können, die von anderen Mustern geprägt sind.
- Die Entwicklung der Software eines Systems sollte unterstützt werden. Ein Mustersystem sollte zeigen, wie man seine Muster anwendet.
- Die Definition und Instantiierung von Beziehungen zu Mustern aus fremden Mustersystemen, die Wissen aus anderen Domänen aufnehmen, sollte unterstützt werden.
- Es sollte seine eigene Evolution unterstützen. Im Zuge der Weiterentwicklung einer Domäne wird sich auch ein Mustersystem weiterentwickeln. Existierende Muster werden sich ändern, ihre Beschreibung wird verbessert, neue oder fehlende Muster werden hinzugefügt, existierende Muster veralten und werden entfernt. Weiterhin werden neue Beziehungen eingeführt bzw. existierende verändert oder gelöscht.
- Geeignete Softwareprozesse zur Änderung, Einführung und Löschung von Mustern, bzw. Festlegung von Beziehungen zwischen ihnen sollten bereitgestellt werden, bzw. sollte der Entwicklungsphasen-gerechte Einsatz seiner Muster unterstützt werden.
- Es sollte die Verifikation von Softwaremodellen unterstützen bzw. forcieren, indem innerhalb des Systems bestimmte Beziehungen als korrekt nachgewiesen werden, bzw. bei Benutzung von Mustern aus dem System bestimmte Eigenschaften leicht nachgewiesen werden können.

Ein weiterer Begriff, der in diesem Zusammenhang von Bedeutung ist, ist der der Mustersprache (engl. pattern language), der von Alexander [AIS+77] geprägt wurde (s. Definition 3 in Tabelle 4.1). Bei dieser Definition stehen Beziehungen zwischen Mustern innerhalb des gleichen Kontextes im Vordergrund. Auch an eine Klassifizierung der Muster in Kategorien wird gedacht. Ein Mustersystem fügt einem Musterkatalog tiefe Struktur, reiche Musterinteraktion und Einheitlichkeit hinzu. Beide Mustersysteme und Mustersprachen formen kohärente und eng verwobene Muster, um die Probleme in einer speziellen Domäne zu beschreiben und zu lösen. Aber eine Mustersprache ist robuster, umfassender und vollständiger als ein Mustersystem. Der Hauptunterschied besteht darin, dass Mustersprachen idealerweise vollständig (engl. computational complete) sind, indem sie alle möglichen Kombinationen und Variationen benutzen, um vollständige Architekturen aufzuzeigen. In der Praxis kann es jedoch sehr schwierig sein, den Unterschied zwischen einem Mustersystem und einer Mustersprache herauszufinden.

Auch in [QC99] werden Musterkataloge gegen Mustersprachen abgegrenzt. Während Mustersysteme eine zusammenhängende Sammlung von Mustern über ein relativ breites Thema sind, sollte eine Mustersprache mehr als nur über ein breites Thema verfügen. Eine Mustersprache korrespondiert letztlich in einer Sammlung, die auf einer zentralen Denkweise beruht und die eine Art "Mega-Pattern" bildet. Die gesamte Sprache bildet somit ein grundlegendes, gemeinsames Problem mit einem zugehörigen Kontext, den Kräften, der Lösung sowie dem resultierenden Kontext und einem Entwurfshandbuch (in dem jedes Muster auf einer Ebene adressiert wird). Diese zweckorientierte Kohärenz ist es, die der Mustersprache die Bedeutung einer Hülle gibt. Ein Mustersystem besitzt nicht notwendigerweise alle diese Bestandteile. Es kann sich auf ein breiteres oder engeres Thema konzentrieren, hat aber nicht notwendigerweise eine klare Aufgabe oder eine

Agenda und kann auf viele Weisen dazu führen, dass die Beziehungen zwischen Mustern schwerer zu finden sind, bzw. dass einige wichtige Lücken im Problemraum ungefüllt bleiben (womit man keine vollständige Lösung bzw. Hülle erhält). Aber der vielleicht größte Unterschied zwischen Mustersprachen und Mustersystemen besteht darin, dass Mustersprachen nicht auf einmal festgelegt werden. Sie entwickeln sich stattdessen stückweise aus Mustersystemen durch stetige Evolution (auch ein Mustersystem erwächst aus einem Musterkatalog). Genauso wie Mustersprachen helfen, vollständige Architekturen wachsen zu lassen, können Mustersysteme dazu dienen, um inkrementell in vollständige Mustersprachen überzugehen.

Kapitel 5

Temporal Logic of Actions (TLA)

TLA ist eine von Leslie Lamport [Lam91b, Lam94] entwickelte Sprache, die auf einer linearen Temporalzeitlogik erster Ordnung und Zermelo-Fränkel Mengentheorie beruht. In diesem Kapitel werden grundlegende Begriffe aus TLA kurz angegeben. Die Syntax und Semantik von TLA wird in [Lam91a, Lam92], [Lam94, Lam95b] und [Her98] vorgestellt. In Abbildung 5.1 wird eine Übersicht der Syntax und Semantik von TLA angegeben. Die Syntax von TLA definiert Formeln, Prädikate und Zustandsfunktionen. Eine Formel enthält einen unären oder binären, logischen bzw. temporalen Operator. Ein Prädikat ist ein Boolescher Ausdruck, der Konstanten und Variablen enthält oder in dem der binäre Enabled-Operator vorkommt. Eine Zustandsfunktion ist ein nicht-Boolescher Ausdruck, der sich aus Konstanten und Variablen zusammensetzt.

TLA-Formeln beziehen sich auf Modelle in der Form von Zustandstransitionssystemen (engl. State-Transition-System, abgekürzt STS). Ein STS wird durch ein Tripel (S, S_0, N) definiert. Hierbei gibt S den Zustandsraum an, der die Menge der möglichen Zustände definiert. S_0 gibt die Menge der Initialzustände des Modells mit $S_0 \subseteq S$ an. Die Zustandsübergangsrelation N beschreibt die Transitionen zwischen den Zuständen. Der Zustandsraum wird durch die Zustandsvariablen einer TLA-Spezifikation definiert. Ein Zustand ist eine Belegung von Zustandsvariablen mit geeigneten Werten. Der Ablauf, der einem System zugrunde liegt, entspricht einer unendlich langen Zustandsfolge. Eine solche Zustandsfolge wird auch als Verhalten bezeichnet:

$$\sigma = \langle s_0, s_1, s_2, \ldots \rangle$$

Treten dieselben Zustände direkt aufeinanderfolgend in dieser Zustandsfolge auf, so spricht man von Stotterschritten. Mit einer Zustandsfolge, deren Ende aus einer unendlichen Anzahl von Stotterschritten besteht, wird die Terminierung eines beliebigen Systems beschrieben. Eine unendliche Zustandsfolge, in der ab einer bestimmten Stelle nur noch Stotterschritte s_n auftreten, hat somit folgendes Aussehen:

$$\sigma = \langle s_0, s_1, s_2, \ldots s_n, s_n, \ldots \rangle$$

Im Folgenden wird die Spezifikation von Modellen auf der Grundlage von Zuständen und Zustandsfolgen erörtert.

5.1 Sicherheitseigenschaften

Sicherheitseigenschaften (engl. safety properties) geben den Rahmen vor, in dem sich ein System bewegen darf. Korrektes Verhalten beschreibt die Menge erlaubten Verhaltens ohne ein Verhalten zu erzwingen. Sicherheitseigenschaften werden auf der Basis des STS eines Systems als Zustandsfolgen des spezifizierten Systems beschrieben. Die Transitionen geben dabei keine Zwänge, sondern Möglichkeiten zum Schalten vor. Das Stottern des Systems an einer bestimmten Stelle, ohne dass eine Transition stattfindet, ist jederzeit möglich. Die Verletzung einer Sicherheitseigenschaft in einem Systemablauf kann in einem endlichen Teilstück der zugehörigen Zustandsfolge aufgedeckt

Syntax

$\langle formula \rangle \triangleq \langle predicate \rangle \mid \Box[\langle action \rangle]_{\langle state\ function \rangle} \mid \neg\langle formula \rangle \mid \langle formula \rangle \wedge \langle formula \rangle \mid \Box\,\langle formula \rangle$

$\langle action \rangle \triangleq$ boolean-valued expression containing constant symbols, variables, and primed variables

$\langle predicate \rangle \triangleq \langle action \rangle$ with no primed variables $\mid Enabled\langle action \rangle$

$\langle state\ function \rangle \triangleq$ nonboolean expression containing constant symbols and variables

Semantics

$s[[F]] \triangleq F(\forall\, v \in \mathsf{Var} : s[[v]]/v)$

$s[[\pi]]t \triangleq \pi(\forall\, v \in \mathsf{Var} : s[[v]]/v,\, t[[v]]/v')$

$\models \pi \triangleq \forall s, t \in \mathsf{St} : s[[\pi]]t$

$\sigma[[T \wedge U]] \triangleq \sigma[[T]] \wedge \sigma[[U]]$

$\sigma[[\neg T]] \triangleq \neg\sigma[[T]]$

$\models F \triangleq \forall \sigma \in \mathsf{St}^\infty : \sigma[[F]]$

$s[[Enabled\ \pi]] \triangleq \exists t \in \mathsf{St} : s[[\pi]]t$

$\langle s_0, s_1, \ldots\rangle[[\Box T]] \triangleq \forall n \in Nat : \langle s_n, s_{n+1}, \ldots\rangle[[T]]$

$\langle s_0, s_1, \ldots\rangle[[\pi]] \triangleq s_0[[\pi]]s_1$

Additional notation

$F' \triangleq F(\forall\, v \in \mathsf{Var} : v'/v)$

$[\pi]_w \triangleq \pi \vee (w' = w)$

$\langle \pi \rangle_w \triangleq \pi \wedge (w' \neq w)$

$Unchanged\ w \triangleq w' = w$

$\Diamond T \triangleq \neg\Box\neg T$

$T \rightsquigarrow U \triangleq \Box(T \Rightarrow \Diamond U)$

$WF_w(\pi) \triangleq \Box\Diamond\ \langle \pi \rangle_w \vee \Box\Diamond\neg Enabled\langle \pi \rangle_w$

$SF_w(\pi) \triangleq \Box\Diamond\ \langle \pi \rangle_w \vee \Diamond\Box\neg Enabled\langle \pi \rangle_w$

where F is a $\langle state\ function \rangle$; π is an $\langle action \rangle$ or $\langle predicate \rangle$; T and U are $\langle formula \rangle$ s; w is a $\langle state\ function \rangle$; s, s_0, s_1, ... are states; σ is a behavior; $(\forall\, v \in \mathsf{Var} : \ldots/v, \ldots/v')$ denotes substitution for all variables v

Abbildung 5.1: Die Logik TLA (entnommen aus [Lam94] mit geänderten Bezeichnern)

werden. Jetzt wird die Semantik der für Sicherheitseigenschaften relevanten temporalen Formeln und Operatoren vorgestellt. Die Wertebelegung einer Zustandsvariable x im Zustand s wird durch s[[x]] angegeben. Die Semantik einer Zustandsfunktion in einem Zustand s wird durch die Berechnung des Funktionswertes unter der Variablenbelegung aus dem Zustand s definiert:

$$s[[f]] = f(\exists\ 'v' \in Var : s[[v]]/v)$$

Um Transitionen zwischen Zuständen zu spezifizieren, ist der Begriff der Aktion von Bedeutung. Eine Aktion ist eine Funktion, die einen Booleschen Wert s[[A]]t einem Zustandspaar zuweist. Hierbei wird s als gegenwärtiger und t als nachfolgender Zustand betrachtet. Der gegenwärtige Zustand wird durch einen Variablennamen v angegeben, der jeweils durch s[[v]] zu substituieren ist. Der nachfolgende Zustand wird durch gestrichene Variablennamen v' notiert, die durch t[[v]] substituiert werden. Die Semantik wird formal durch folgenden Ausdruck festgelegt:

$$s[[A]]t = A(\forall\ \ 'v' \in Var : s[[v]]/v, t[[v]]/v')$$

Ein Paar von Zuständen s, t heißt ein A-Schritt, gdw. s[[A]]t wahr ist. Wird durch die Aktion A eine atomare Operation eines Programms dargestellt, dann ist das Paar s, t ein A-Schritt, wenn der Zustand t bei der Ausführung der Operation im Zustand s erreicht wird. Aktionen definieren Transitionsklassen, da von einem Zustand aus mehrere Transitionen möglich sind. Aktionendefinitionen dürfen typisierte Datenparameter p besitzen. Damit gilt der Ausdruck $\exists p : \alpha(p)$ für eine Aktion α. Für die Anwendung des Aktionsbegriffs auf Zustandsfolgen $\langle s_0, s_1, s_2, \ldots\rangle$ definiert man, dass [[A]] wahr ist, wenn das Zustandspaar s_0, s_1 ein A-Schritt ist:

$$\langle s_0, s_1, \ldots \rangle [[A]] \triangleq s_0 [[A]] s_1$$

Eine Aktion A heißt gültig notiert durch $\models A$, wenn jeder Schritt ein A-Schritt ist, oder formal:

$$\models A \triangleq \forall s, t \in St : s[[A]]t$$

Mit dem temporalen $\Box$ Operator wird zugesichert, dass eine Formel F für alle s_n gilt:

$$\langle s_0, s_1, \ldots \rangle [[\Box F]] \triangleq \forall n \in Nat : \langle s_n, s_{n+1}, \ldots \rangle [[F]]$$

Ein Verhalten erfüllt $F \wedge G$, wenn sowohl F als auch G erfüllt sind:

$$\sigma[[F \wedge G]] \triangleq \sigma[[F]] \wedge \sigma[[G]]$$

Ein Verhalten erfüllt $\neg F$, wenn F nicht erfüllt wird:

$$\sigma[[\neg F]] \triangleq \neg \sigma[[F]]$$

Eine temporale Formel F wird gültig genannt, notiert durch $\models F$, wenn sie für jedes mögliche Verhalten gilt. Durch St^∞ wird jedes mögliche Verhalten angegeben:

$$\models F \triangleq \forall \sigma \in St^\infty : \sigma[[F]]$$

Eine Aktion A ist in einem gegenwärtig vorliegenden Zustand s schaltbereit (englisch enabled), wenn von diesem Zustand aus ein Zustandsübergang möglich ist, der dieser Aktion entspricht, was im Folgenden formal aufgeführt ist:

$$s[[EnabledA]]t \triangleq \exists t \in St : s[[A]]t$$

Ein Aktionen- oder ein Stotterschritt wird verkürzt mit $[A]_f$ notiert:

$$[A]_f \triangleq A \vee (f' = f)$$

Hierbei gibt f das Tupel der Bezeichner aller in einer Spezifikation verwendeten Variablen an. $\langle A \rangle_f$ notiert einen echten Aktionenschritt, ohne einen Stotterschritt zuzulassen:

$$\langle A \rangle_f \triangleq A \wedge (f' \neq f)$$

Mit dem Schlüsselwort *Unchanged* $f \triangleq f' = f$ wird markiert, welche Zustandsvariablen eines Aktionenschrittes nicht verändert werden. Für einen eingeschränkten Sprachumfang von TLA, der nur Sicherheitseigenschaften betrifft, ist an der Universität Dortmund ein Simulationswerkzeug entwickelt worden [Gra93].

5.2 Lebendigkeitseigenschaften

Der gewünschte Fortschritt in einem System wird durch Lebendigkeitseigenschaften (engl. liveness properties) beschrieben. Durch sie werden keine Aktionen erlaubt oder verboten, sondern sie forcieren, dass sich ein System in bestimmte Richtungen bewegt und bestimmte Zustände einnimmt, wenn das aufgrund der Sicherheitseigenschaften möglich ist. Eine E-Mail, die abgeschickt wird, muss irgendwann bei ihrem Empfänger ankommen und ist damit ein Beispiel für eine Lebendigkeitsanforderung. Sicherheits- und Lebendigkeitsanforderungen stehen miteinander in Beziehung. Um die Verletzung einer Lebendigkeitseigenschaft in einem Systemablauf aufzudecken, kann es erforderlich sein, den vollständigen – möglicherweise unendlichen – Systemablauf unter Einbeziehung des zukünftigen Ablaufs zu untersuchen. TLA definiert den $\Diamond$–Operator (eventually) und den $\leadsto$–Operator (Leads-To) auf der Grundlage des temporalen $\Box$–Operators:

$$\Diamond F \triangleq \neg \Box \neg F$$

Mit dem Leads-To-Operator wird spezifiziert, dass – wenn eine bestimmte Formel F gültig ist – irgendwann in der Zukunft eine Eigenschaft G gilt:

$$F \rightsquigarrow G \triangleq \Box(F \Rightarrow \Diamond G)$$

Die schwache Fairness (durch WF abgekürzt) erzwingt Zustandsfolgen, in denen unendlich viele Zustandsübergänge einer Ausführung der Aktion $\langle A \rangle_f$ entsprechen oder in denen die Aktion A in unendlich vielen Zuständen nicht schaltbereit ist:

$$WF_f(A) \triangleq \Box\Diamond\langle A \rangle_f \vee \Box\Diamond\neg Enabled\langle A \rangle_f$$

Durch die starke Fairness (Abkürzung SF) werden Zustandsfolgen erzwungen, in denen unendlich viele Zustandsübergänge einer Ausführung der Aktion $\langle A \rangle_f$ entsprechen oder in denen die Aktion $\langle A \rangle_f$ nur in endlich vielen Zuständen schaltbar ist:

$$SF_f(A) \triangleq \Box\Diamond\langle A \rangle_f \vee \Diamond\Box\neg Enabled\langle A \rangle_f$$

Eine Lebendigkeitseigenschaft garantiert, dass erwünschte Zustände erreicht werden.

In einem nebenläufigen System unterscheidet man zwischen schwacher und starker Fairness. Schwache Fairness (engl. weak fairness, justice) bedeutet, dass eine Aktion ausgeführt werden muss, wenn die Ausführung dieser ab einem bestimmten Zeitpunkt immer möglich ist. Anders: Wird eine Aktion nur endlich oft ausgeführt, so ist diese in einem Verhalten unendlich oft nicht ausführbar. Es versichert, dass die Aktion schließlich ausgeführt wird oder deren Ausführung – wenn auch nur für eine bestimmte Zeitspanne – unmöglich wird.

Starke Fairness (engl. strong fairness, compassion) bedeutet, dass eine Aktion ausgeführt werden muss, wenn die Ausführung dieser unendlich oft möglich ist. Anders: Wird eine Aktion nur endlich

Syntax

$$\begin{array}{lcl}
\langle \textit{general formula} \rangle & \triangleq & \langle \textit{formula} \rangle \mid \exists\!\!\!\exists\ \langle \textit{variable} \rangle : \langle \textit{general formula} \rangle \\
& & \mid\ \exists\ \langle \textit{rigid variable} \rangle : \langle \textit{general formula} \rangle \\
& & \mid\ \langle \textit{general formula} \rangle \wedge \langle \textit{general formula} \rangle \\
& & \mid\ \neg\langle \textit{general formula} \rangle \\
\langle \textit{formula} \rangle & \triangleq & \text{a simple TLA formula (siehe Abbildung 5.1)}
\end{array}$$

Semantics

$$\langle s_0, s_1, \ldots \rangle =_w \langle t_0, t_1, \ldots \rangle \triangleq \forall\, \mathsf{n} \in \mathsf{Nat} : \forall\, v \in \mathsf{Var} \setminus \{w\} : s_{\mathsf{n}}[\![v]\!] = t_{\mathsf{n}}[\![v]\!]$$

$$\begin{array}{lcl}
\natural\langle s_0, s_1, s_2, \ldots \rangle & \triangleq & \textbf{if}\ \forall \mathsf{n} \in \mathsf{Nat} : s_{\mathsf{n}} = s_0 \\
& & \quad \textbf{then}\ \langle s_0, s_0, s_0, \ldots \rangle \\
& & \quad \textbf{else}\ \ \textbf{if}\ s_1 = s_0\ \textbf{then}\ \natural\langle s_1, s_2, s_3, \ldots \rangle \\
& & \qquad\qquad\qquad\ \ \textbf{else}\ \ \langle s_0 \rangle \circ \natural\langle s_1, s_2, \ldots \rangle
\end{array}$$

$$\sigma[\![\exists\!\!\!\exists\ w : T]\!] \triangleq \exists\, \rho, \tau \in \mathbf{St}^\infty : (\natural\sigma = \natural\rho) \wedge (\rho =_w \tau) \wedge \tau[\![T]\!]$$

$$\sigma[\![\exists\, \mathsf{c} : T]\!] \triangleq \exists \mathsf{c} \in \mathsf{Val} : \sigma[\![T]\!]$$

Proof Rules

$$E1.\ \vdash T(f/w) \Rightarrow \exists\!\!\!\exists\ w : T \qquad E2.\ \frac{T \Rightarrow U \quad w \text{ does not occur free in } U}{(\exists\!\!\!\exists\ w : T) \Rightarrow U}$$

$$F1.\ \vdash T(\mathsf{e}/\mathsf{c}) \Rightarrow \exists\, \mathsf{c} : T \qquad F2.\ \frac{T \Rightarrow U \quad \mathsf{c} \text{ does not occur free in } U}{(\exists\, \mathsf{c} : T) \Rightarrow U}$$

where w is a $\langle \textit{variable} \rangle$ — T, U are $\langle \textit{general formula} \rangle$s
f is a state function — $s, s_0, t_0, s_1, t_1, \ldots$ are states
c is a $\langle \textit{rigid variable} \rangle$ — σ is a behavior
e is a constant expression — $\circ$ denotes concatenation of sequences

Abbildung 5.2: Die Quantoren in TLA (entnommen aus [Lam94] mit geänderten Bezeichnern)

oft ausgeführt, so ist diese in einem Verhalten nur endlich oft ausführbar. Es versichert, dass die Aktion schließlich ausgeführt wird oder, dass deren Ausführung schließlich für immer unmöglich wird.

Ist ein Verhalten stark fair bezüglich einer Aktion, so ist es auch schwach fair für diese Aktion.

5.3 Kanonische Formel

Insgesamt wird ein System durch eine temporale Formel spezifiziert, in der die Sicherheitseigenschaften und die Lebendigkeitseigenschaften miteinander konjugiert werden. Diese temporale Formel wird als kanonische Formel bezeichnet. Die Sicherheitseigenschaften werden aus den Initialzuständen und der mit dem always-Operator umschlossenen Disjunktion der Aktionen spezifiziert. Die Zustandsvariablen des Systems werden als Zustandsfunktion angeben, die bei einem Stotterschritt nicht geändert wird. Die Einführung neuer Lebendigkeitseigenschaften führt häufig auch zur Einführung neuer Sicherheitseigenschaften. Nach einem Vorschlag aus [AS85] kann man Lebendigkeitsanforderungen indirekt durch Fairnessanforderungen an Aktionen spezifizieren (s. Abbildung 5.1). Daher werden Fairnessanforderungen in der kanonischen Formel für jede Aktion separat spezifiziert:

$$Sys = Init \wedge \Box[A_1 \vee \ldots \vee A_n]_f \wedge WF/SF(A_1) \wedge \ldots WF/SF(A_n)$$

5.4 Beweisregeln für TLA

TLA stellt außer Syntax und Semantik Schlussregeln zum Beweis von Theoremen bereit, die in Abbildung 5.3 angegeben sind. Zusätzlich werden spezielle Quantoren definiert (s. Abbildung 5.2).

Die Regeln STL1-STL6, die Lattice Regel und die grundlegenden Regeln TLA1 und TLA2 stellen ein relativ vollständiges Beweissystem für den Beweis von Systemen zur Verfügung, die mittels TLA spezifiziert worden sind.

Die Regel TLA1 stellt das Induktionsprinzip bereit, um die Formel $\Box P$ zu beweisen. Hierdurch wird die Tatsache zugesichert, dass ein Prädikat P immer wahr ist, wenn P initial gilt und durch jeden Schritt für den P wahr ist P wahr bleibt. Die Regel TLA2 folgt unmittelbar aus der Gültigkeit von STL4 und STL5.

Zum Nachweis einer Invarianzeigenschaft $\Box I$ eines Systems wird die Regel INV1 eingesetzt. Die Hypothese sichert zu, dass ein $[\pi]_w$ -Schritt I nicht falsifizieren kann. Durch die Konklusion wird zugesichert, dass wenn I initial gilt und jeder Schritt ein $[\pi]_w$ -Schritt ist, I immer wahr ist.

Die Regel TLA1 stellt das Induktionsprinzip, um die Formel P zu beweisen, bereit. Hierdurch wird die Tatsache zugesichert, dass ein Prädikat P immer wahr ist, wenn P initial gilt und durch jeden Schritt für den P wahr ist P wahr bleibt.

Aus der Gültigkeit von STL4 und STL5 folgt unmittelbar die Regel TLA2. Die Regeln WF1 und SF1 bilden die Basis für Beweise von Lebendigkeitseigenschaften. Jede dieser Regeln besitzt jeweils drei Prämissen.

Die Regel WF1 wird angewendet, um die Leads-To-Eigenschaft $F \leadsto G$ aus einer schwachen Fairnessbedingung $WF_w(A)$ nachzuweisen.

In der Regel SF1 sind die ersten beiden Prämissen mit denen der Regel WF1 identisch. Nur die dritte Prämisse (s. Abbildung 5.3) wird modifiziert. Dies ist notwendig, da die starke Fairness fordert, dass die Aktion α auch ausgeführt wird, wenn sie nicht kontinuierlich schaltbereit ist. Dafür ist die Prämisse $F \Rightarrow Enabled\ \langle\alpha\rangle_w$ aus der WF1 Regel zu stark. Stattdessen wird die Prämisse aus (5.1) verwendet. T bezeichnet hierbei eine temporallogische Formel, etwa eine Invariante bzw. eine Fairnessangabe.

$$\Box F \wedge \Box[\pi]_w \wedge \Box T \Rightarrow \Diamond Enabled\ \langle\alpha\rangle_w \tag{5.1}$$

Die Lattice Regel gestattet den Beweis einer Leads-To-Eigenschaft durch Verwendung einer fundierten Relation. Eine Relation $\prec \subseteq S \times S$ auf einer Menge S heißt fundiert, wenn es keine unendliche Folge $d_0 \prec d_1 \prec d_2 \ldots$ gibt. Fundierte Relationen sind irreflexiv und antisymmetrisch. Ein Beispiel für eine fundierte Relation ist etwa $(\mathbb{N}, <)$. Das Vorliegen einer solchen fundierten

Relation ist eine Prämisse für die Lattice Regel. Die zusätzliche Prämisse der Lattice Regel enthält die Formel H. Die Prämisse drückt aus, dass man entweder U erreicht hat oder sich in der Relation vorwärts bewegt hat. Die Prämisse wird häufig durch die wiederholte Anwendung der WF1 bzw. SF1 Regel bewiesen.

Für einen Verfeinerungsbeweis wird eine Zustandsabbildung zwischen den Zuständen des Fein- und des Grobsystems angegeben. Das Initialisierungsprädikat des Feinsystems muss das Initialisierungsprädikat des Grobsystems implizieren. Jeder $\langle A \rangle_f$-Schritt einer Aktion des Feinsystems führt entweder zu einem Schritt des Grobsystems oder zu einem Stotterschritt. Zum Nachweis der Korrektheit der Verfeinerung werden Invariantenbeweise durchgeführt. Verfeinerungsbeweise werden detaillierter in Abschnitt 5.5 behandelt. Die Regeln WF2 und SF2 ermöglichen bei Verfeinerungen zu beweisen, dass Fairnessanforderungen, die im verfeinerten System gelten, auch auf das Grobsystem übertragen werden können.

The Rules of Simple Temporal Logic

STL1. $\dfrac{T \text{ provable by propositional logic}}{\Box T}$

STL2. $\vdash \Box T \Rightarrow T$

STL3. $\vdash \Box\Box T \equiv \Box T$

STL4. $\dfrac{T \Rightarrow U}{\Box T \Rightarrow \Box U}$

STL5. $\vdash \Box(T \wedge U) \equiv (\Box T) \wedge (\Box U)$

STL6. $\vdash (\Diamond\Box T) \wedge (\Diamond\Box U) \equiv \Diamond\Box(T \wedge U)$

STL7. $\vdash \Box\Diamond\Box T \equiv \Diamond\Box T$

LATTICE. $\succ$ a well-founded partial order on a set S

$$\frac{T \wedge (\mathsf{c} \in \mathsf{S}) \Rightarrow (H_{\mathsf{c}} \leadsto (U \vee \exists \mathsf{d} \in \mathsf{S} : (\mathsf{c} \succ \mathsf{d}) \wedge H_{\mathsf{d}}))}{T \Rightarrow ((\exists \mathsf{c} \in \mathsf{S} : H_{\mathsf{c}}) \leadsto U)}$$

The Basic Rules of TLA

TLA1. $\dfrac{F \wedge (w' = w) \Rightarrow F'}{\Box F \equiv F \wedge \Box[F \Rightarrow F']_w}$

TLA2. $\dfrac{F \wedge [\alpha]_w \Rightarrow G \wedge [\beta]_u}{\Box F \wedge \Box[\alpha]_w \Rightarrow \Box G \wedge \Box[\beta]_u}$

Additional Rules

INV1. $\dfrac{I \wedge [\pi]_w \Rightarrow I'}{I \wedge \Box[\pi]_w \Rightarrow \Box I}$

INV2. $\vdash \Box I \Rightarrow (\Box[\pi]_w \equiv \Box[\pi \wedge I \wedge I']_w)$

WF1.

$$\frac{\begin{array}{l} F \wedge [\pi]_w \Rightarrow (F' \vee G') \\ F \wedge \langle \pi \wedge \alpha \rangle_w \Rightarrow G' \\ F \Rightarrow \mathit{Enabled}\ \langle \alpha \rangle_w \end{array}}{\Box[\pi]_w \wedge \mathrm{WF}_w(\alpha) \Rightarrow (F \leadsto G)}$$

WF2.

$$\frac{\begin{array}{l} \langle \pi \wedge \beta \rangle_w \Rightarrow \langle \overline{\lambda} \rangle_{\overline{u}} \\ F \wedge F' \wedge \langle \pi \wedge \alpha \rangle_w \wedge \overline{\mathit{Enabled}\ \langle \lambda \rangle_u} \Rightarrow \beta \\ F \wedge \overline{\mathit{Enabled}\ \langle \lambda \rangle_u} \Rightarrow \mathit{Enabled}\ \langle \alpha \rangle_w \\ \Box[\pi \wedge \neg\beta]_w \wedge \mathrm{WF}_w(\alpha) \wedge \Box T \\ \qquad \wedge \Diamond\Box\overline{\mathit{Enabled}\ \langle \lambda \rangle_u} \Rightarrow \Diamond\Box F \end{array}}{\Box[\pi]_w \wedge \mathrm{WF}_w(\alpha) \wedge \Box T \Rightarrow \overline{\mathrm{WF}_u(\lambda)}}$$

SF1.

$$\frac{\begin{array}{l} F \wedge [\pi]_w \Rightarrow (F' \vee G') \\ F \wedge \langle \pi \wedge \alpha \rangle_w \Rightarrow G' \\ \Box F \wedge \Box[\pi]_w \wedge \Box T \Rightarrow \Diamond \mathit{Enabled}\ \langle \alpha \rangle_w \end{array}}{\Box[\pi]_w \wedge \mathrm{SF}_w(\alpha) \wedge \Box T \Rightarrow (F \leadsto G)}$$

SF2.

$$\frac{\begin{array}{l} \langle \pi \wedge \beta \rangle_w \Rightarrow \langle \overline{\lambda} \rangle_{\overline{u}} \\ F \wedge F' \wedge \langle \pi \wedge \alpha \rangle_w \Rightarrow \beta \\ F \wedge \overline{\mathit{Enabled}\ \langle \lambda \rangle_u} \Rightarrow \mathit{Enabled}\ \langle \alpha \rangle_w \\ \Box[\pi \wedge \neg\beta]_w \wedge \mathrm{SF}_w(\alpha) \wedge \Box T \\ \qquad \wedge \Box\Diamond\overline{\mathit{Enabled}\ \langle \lambda \rangle_u} \Rightarrow \Diamond\Box F \end{array}}{\Box[\pi]_w \wedge \mathrm{SF}_w(\alpha) \wedge \Box T \Rightarrow \overline{\mathrm{SF}_u(\lambda)}}$$

where T, U, H_{c} are TLA formulas; F, G, I are predicates; α, β, π, λ are actions; w, u are state functions

Abbildung 5.3: Die Axiome und Beweisregeln von TLA (entnommen aus [Lam94] mit geänderten Bezeichnern)

5.5 Verfeinerung von TLA-Spezifikationen

Der Softwareprozess bei der Entwicklung von Programmen verläuft mittels schrittweiser Verfeinerung. Bereits Wirth [Wir71] weist daraufhin, dass beim Programmentwurf aufeinanderfolgende Sequenzen von Verfeinerungsschritten ausgeführt werden. Durch jeden Verfeinerungsschritt werden Entwurfsentscheidungen impliziert, die aus einem abstrakteren Programm schließlich zur Implementierung führen.

Der Begriff der Verfeinerung ist seitdem von zahlreichen Wissenschaftlern erforscht und von Programmen auf Spezifikationen mit unterschiedlichen funktionalen und nicht-funktionalen Eigenschaften übertragen worden. Der Begriff der Datenverfeinerung, der die Verfeinerung von abstrakten Datentypen in sequentiellen Programmen behandelt, geht auf Guttag [Gut97] und Nipkov [Nip86] zurück. Von besonderem Interesse ist die Korrektheit der Verfeinerung von einem abstrakten Programm in ein konkretes Programm und deren Nachweis. Der Nachweis der Korrektheit einer Verfeinerung steht in engem Zusammenhang mit der Beobachtung der Belegung mit Werten, die die Variablen des konkreten Programms bei dessen Ausführung annehmen. Die Korrektheit einer Verfeinerung wird in [GM93] folgendermaßen definiert:

> „Given two programs, one called concrete and the other called abstract, the concrete program implements (or refines) the abstract program whenever the use of the concrete program does not lead to an observation which is not also an observation of the abstract program."

Broy unterscheidet in [BS01] die Glass-Box- und die Black-Box Verfeinerung. Die Black-Box-Sicht zeichnet sich dadurch aus, dass nur die Interaktion einer Komponente mit ihrer Umgebung betrachtet wird. In der Glass-Box Sicht werden hingegen auch ihre Interna, wie die Zustandssicht und ihre Struktur, beschrieben. Eine gute Einführung und Abgrenzung der verschiedenen Kalküle zur Verfeinerung und deren Beweistechniken, die auch die historische Entwicklung umfasst, wird in [dRE98] gegeben.

5.5.1 Verfeinerung nach Abadi und Lamport

Im Folgenden wird der von M. Abadi und L. Lamport geprägte und für diese Arbeit grundlegende Verfeinerungsbegriff [AL90, AL91a, AL88], der auch die für verteilte Systeme wichtige Nebenläufigkeit umfasst, vorgestellt. Dort wird auf die Transitivität von korrekten Verfeinerungsschritten hingewiesen, die von einer abstrakten Spezifikation S_0 über eine Folge von nachweisbar korrekten Entwurfsschritten s_j über die Spezifikationen S_j zur Spezifikation S_n führen:

$$S_0 \rightarrow S_1 \rightarrow S_j \rightarrow \ldots \rightarrow S_n$$

Im Folgenden wird durch das Quadrupel $S_1 = (\Sigma_1; F_1; N_1; L_1)$ die konkrete Spezifikation eines Systems notiert und mit $S_2 = (\Sigma_2; F_2; N_2; L_2)$ die abstrakte Spezifikation eines Systems angegeben. Mit Σ_1 wird der Zustandsraum von S_1 bezeichnet. F_1 bezeichnet die Menge der Initialzustände von S_1. N_1 notiert die Zustandsübergangsrelation $\Sigma_1 \times \Sigma_1$ von S_1. Die Lebendigkeitseigenschaften von S_1 werden mit L_1 angegeben. Die Bezeichnungen für die Spezifikation S_2 gelten analog.

Für den Zustandsraum Σ einer Spezifikation gilt: $\Sigma_I \times \Sigma_E$, wobei mit Σ_I die internen (oder unsichtbaren) Variablen einer Spezifikation notiert und die externen Variablen einer Spezifikation mit Σ_E angegeben werden. Mit Π_E wird die Projektion $\Pi_E : \Sigma_I \times \Sigma_E \rightarrow \Sigma_E$ notiert.

Zur Durchführung von Verfeinerungsbeweisen werden die Eigenschaften von Zustandsabbildungen (durch eine Funktion mit Zustandsmengen im Bild- und Wertebereich) zwischen den Zuständen von S_1 und S_2 untersucht. Lamport und Abadi beweisen in [AL91a] einen Satz, der aussagt, dass sobald eine Funktion $f : S_1 \mapsto S_2$ mit den Eigenschaften eines Refinement Mappings (s. Abbildung 5.4) existiert, $S_1 \Rightarrow S_2$ eine korrekte Verfeinerung ist und S_2 durch S_1 korrekt implementiert wird. Weiterhin wird eine Existenzaussage bezüglich der Existenz von f gemacht. Diese Funktion f existiert gdw. $S_1 \Rightarrow S_2$ gilt und die später aufgeführten Voraussetzungen erfüllt sind. Im Folgenden seien $f : \Sigma_1 \rightarrow \Sigma_2$ sowie die Systeme für S_1 und S_2 gegeben. Die Zustandsabbildung f muss folgende vier Eigenschaften (R1-R4) erfüllen, damit sie ein Refinement Mapping ist:

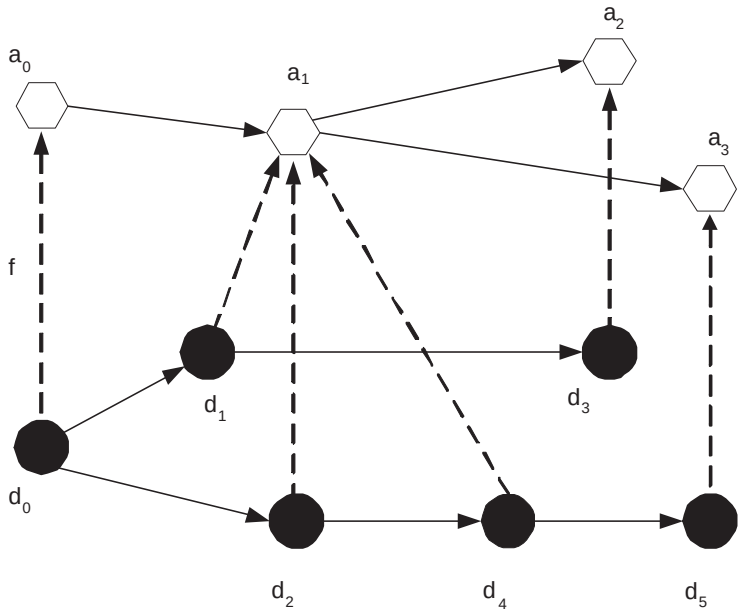


Abbildung 5.4: Ein Refinement Mapping

- R1: Die externen Zustandskomponenten von S_1 werden durch die Funktion f nicht verändert, oder für alle $s \in \Sigma_1$ gilt $\Pi_E(f(s)) \in \Pi_E(s)$. Das Refinement-Mapping belässt somit die externen Variablen unverändert und berücksichtigt nur die internen Variablen.

- R2: Die Initialzustandstreue, d. h. Initialzustände von S_1, werden auf Initialzustände von S_2 abgebildet mit formaler Notation $f(F1) \subseteq F2$.

- R3: Die Transitionstreue, d. h. wenn $\langle s, t\rangle \in N_1$ dann ist $\langle f(s), f(t)\rangle \in N_2$ oder $f(s) = f(t)$. Die Funktion f bildet Aktionen aus N_1 auf Stotterschritte oder Aktionen aus N_2 ab. Jede Transition aus dem verfeinerten System wird auf eine Transition oder einen Stotterschritt des abstrakten Systems abgebildet.

- R4: Die f-Bilder von Abläufen von S_1 entsprechen den Fairnessanforderungen von L_2.

Für den Satz aus [AL91a] gelten die nachfolgend angegebenen Voraussetzungen. Die Spezifikation S_1 ist machine closed. Das bedeutet, dass durch jede zusätzliche Eigenschaft, die zur Spezifikation von Lebendigkeitseigenschaften benutzt wird, keine neue Sicherheitseigenschaft für das STS von S_2 eingeführt wird. Das STS spezifiziert also soviel wie möglich. Dies ist garantiert, wenn nur starke und schwache Fairness zur Spezifikation der Lebendigkeitseigenschaften von Aktionen verwendet werden.

Die Spezifikation S_2 besitzt nur einen endlichen nicht-sichtbaren Nicht-Determinismus. Für jede endliche Anzahl von Schritten, die ein außen sichtbares S_2 beschriebenes Verhalten annehmen kann, gibt es nur eine endliche Anzahl von Auswahlmöglichkeiten für die internen Zustandskomponenten. Diese Eigenschaft wird mit FIN (Finite Invisible Non-Determinism) bezeichnet.

Die Spezifikation S_2 ist intern kontinuierlich. Für jedes vollständige Verhalten, das nicht erlaubt ist, kann bestimmt werden, dass es nicht erlaubt ist, indem nur die extern sichtbaren Komponenten und ein Teil des vollständigen Verhaltens betrachtet werden. Diese Eigenschaft wird mit IC (Internally Continuous) abgekürzt.

Sind diese Voraussetzungen erfüllt, lässt sich immer ein Refinement Mapping für den Beweis $S_1^{hp} \Rightarrow S_2$ angegeben.

Nach [AL91a] ist die Existenz eines Refinement Mappings eine hinreichende aber nicht notwendige Bedingung für die Existenz einer korrekten Verfeinerung. Es gibt bestimmte Situationen bei

denen kein Refinement Mapping f zwischen S_2 und S_1 existiert. Wenn in S_1 die Anzahl der bisher betroffenen Transitionen zwischengespeichert wird, oder werden in S_1 nicht-deterministische Entscheidungen früher getroffen als in S_2, oder aber wenn in S_1 abstrakter modelliert wird als in S_2, lässt sich kein Refinement-Mapping finden. In diesen Fällen ist es möglich eine erweiterte Spezifikation S_1^{hp}, welche durch die Einführung künstlicher Hilfsvariablen in den Zustandsraum von S_1 erzeugt wird, anzugeben. Durch die Einführung von Hilfsvariablen wird die ursprüngliche Spezifikation von S_1 nicht verändert, da die Variablen nur in einer Zwischenebene für die Verfeinerungsbeweise existieren. Es wird zwischen Historien-, Prophezeiungs- und Stotterhilfsvariablen unterschieden. Historienvariablen sammeln in der Art eines Logbuchs Informationen über den bisherigen Systemablauf. Sie besitzen einen Read-Only Charakter, d. h. vorgefallene Ereignisse werden dokumentiert. Prophezeiungsvariablen stellen bereits gegenwärtig Informationen über zu einem späteren Zeitpunkt nicht-deterministisch gewählte Alternativen zur Verfügung. Dadurch werden nicht-deterministische Auswahlvorgänge zeitlich nach vorn verschoben. Stotterhilfsvariablen werden selten verwendet, um in einem Stotterschritt eines Systems einen echten Zustandsübergang einzufügen. Das STS zu S_1^{hp} wird aus S_1 durch die Einführung zusätzlicher Prophezeiungs- und Historienvariablen gebildet.

5.5.2 Durchführung von Verfeinerungsbeweisen

Zum Beweis der Bedingungen R1, R2 und R3 werden jeweils nur einzelne Zustände und Zustandspaare untersucht. Der Nachweis der Eigenschaften von R4 zieht die Betrachtung ganzer Zustandsfolgen nach sich. Praktisch sind für den Beweis die Namen von Zustandsvariablen zu verändern, neue Zustandsvariablen einzuführen oder die Namen umzubenennen. Zum Beweis werden in der Regel Invarianten verwendet, die es erlauben zu beweisen, ob eine Aktion die Eigenschaft R3 erfüllt. Die Beweisstrukturen sind sehr breit und betreffen jede Aktion des Grob- und des Feinsystems.

```
TickGS(t : Real) ≜ ! Aktion Tick des Grobsystems für Zeitschritt
      LET
         TimerProg(action, timerVariable) ≜
         IF(enabled(action))
         THEN timerVariable + now′ - now
         ELSE timerVariable
      IN
         ∧ now′ = t
         ∧ now′ > now
         ∧ now′ ≤ now + ϵ
         ∧ enabled(Dequeue) ⇒ now′ ≤ now + (tDequeue - TimerDequeue)
         ...
         ∧ timerDequeue′ = TimerProg(Dequeue, timerDequeue)
         ...;
```

Abbildung 5.5: Das IF-THEN-ELSE und das LET-IN Konstrukt

5.6 Spezifikationssprache TLA+

Um TLA-Formeln in einer Spezifikation zu verwenden, wurde die Spezifikationssprache TLA+ entwickelt. TLA+ beruht stärker auf mathematischen Formalismen als auf den typischen Notationen einer Programmiersprache, wie in [LB03] betont wird. TLA+ ist im Gegensatz zu anderen Spezifikationssprachen grundsätzlich nicht typisiert. TLA+ erlaubt zunächst die Verwendung der temporalen Operatoren aus TLA. Zusätzlich wurden das IF-THEN-ELSE (ITE) Konstrukt, das CASE-Konstrukt und das LET-IN-Konstrukt aufgenommen.

Die ersten beiden Konstrukte entsprechen den in Programmiersprachen üblichen Fallunterscheidungen. Das LET-IN-Konstrukt erlaubt die Bindung von Symbolen an Definitionen, wobei später

in einer Spezifikation auftretende Symbole textuell durch ihre Definition ersetzt werden. Die Abbildung 5.5 zeigt, wie in der Aktion *Tick* das Symbol *TimerProgress* als ein Operator mit einem ITE Konstrukt definiert wird. In der Bedingung wird geprüft, ob die als Parameter übergebene Aktion *action* schaltbereit ist. Ergibt die Auswertung der Bedingung wahr ist das Ergebnis die Summe $timerVariable + now' - now$, sonst ist es der Wert des Parameters *timerDequeue*.

Weiterhin wird der CHOOSE Operator unterstützt, der in der Logik auch als Hilberts ϵ-Operator bekannt ist [Lam03]. Dieser Operator erlaubt die Spezifikation eines eindeutig definierten Wertes und ist bei der abstrakten Definition von Datenstrukturen sehr nützlich. Der Ausdruck $x = CHOOSE\ n\ :\ n \in \{1, 2, 3\}$ selektiert einen Wert für x aus der Menge $\{1, 2, 3\}$. TLA+ erlaubt es Funktionen – wie in der Mathematik – zu spezifizieren. Eine Funktion f hat einen Wertebereich, der mit DOMAIN notiert wird und ermöglicht es einem Element x des Wertebereichs, den Wert $f[x]$ zuzuordnen. Der Ausdruck $[fexcept![c] = e]$ entspricht der Funktion $\hat{f}$, die identisch zu f ist, außer dass $\hat{f}[c] = e$ gilt. Allgemein kann man $[fexcept![c_1] = e_1; \ldots; ![c_n] = e_n]$ angeben, um in f die Werte c_1 bis c_n durch e_1 bis e_n zu ersetzen. Datenstrukturen, wie Felder und Rekords, werden in TLA+ auf der Basis von Funktionen behandelt. Ein Rekord mit den Komponenten $comp_1$ und $comp_2$ wird durch $[comp_1 \mapsto val_1; comp_2 \mapsto val_2]$ angegeben, wobei val_1 und val_2 die Werte der Komponenten angeben. Für die Änderung einzelner Komponenten des Rekords gibt es wiederum einen Ausdruck mit dem **except** Schlüsselwort. So wird durch $[r\ except!.c_1 = e_1, \ldots, !.c_n = e_n]$ das Rekord r, in dem die Komponenten c_1 bis c_n durch die Werte e_1 bis e_n modifiziert worden sind, angegeben.

Weiterhin werden TLA+-Spezifikation auf der Basis von Modulen strukturiert, um die Erstellung umfangreicherer Spezifikationen zu unterstützen. In Abbildung 5.6 wird eine typische TLA+-Spezifikation am Beispiel eines Aktors gezeigt. Die TLA+-Spezifikation importiert das Modul *TLC*. Sie besitzt die Zustandsvariablen *x*, *sActor*, *qu*. Mit dem Initialisierungsprädikat *INIT* werden die Zustandsvariablen *qu*, *x* und *sActor* initialisiert. Weiterhin besitzt das Modul die vier Aktionen *enqueue*, *dequeue*, *process*, *setValueReply*. In der Zustandsübergangsrelation *Next* wird die Disjunktion der Aktionen gebildet. Die Invariante *Inv* gibt an, dass *sActor* die Werte *"init"*, *"enqueued"*, *"dequeued"* und *"processed"* annehmen darf. Die Spezifikation *Spec* wird durch Konjunktion von *INIT*, der Zustandsübergangsrelation *Next* und den Fairnesseigenschaften gebildet.

5.7 Model-Checking

Eine andere Methode zur Verifikation der Spezifikation eines Systems – neben der in Abschnitt 5.3 vorgestellten Deduktion mittels Beweisregeln – stellt das Model-Checking dar, bei dem die Korrektheit eines Systems nicht durch Handbeweise, sondern im Wesentlichen durch Rechnerunterstützung nachgewiesen wird [GV08]. Auch für das Model-Checking werden Spezifikationen auf der Basis von endlichen Zustandstransitionssystemen verwendet. Ebenso lassen sich mit Model-Checking Sicherheits- und Lebendigkeitseigenschaften sowie korrekte Verfeinerung nachweisen. Weiterhin existieren Model-Checker, die die Verifikation von Realzeiteigenschaften vornehmen können. Eine gute Einführung zum Thema Model-Checking wird in [CGP00] gegeben. Das Model-Checking Problem wird dort folgendermaßen definiert:

> „Given a Kripke structure M = (S, R, L) that represents a finite-state concurrent system and a temporal logic formula f expressing some desired specification, find the set of all states in S that satisfy f $\{s \in S \mid M, s \models f\}$. "

Eingriffe durch an der Verifikation beteiligte Personen sind beim praktischen Einsatz des Model-Checking zur Analyse der Verifikationsergebnisse – etwa um Fehlerausgaben zu behandeln – notwendig.

Das wesentliche Problem beim Model-Checking besteht in der Explosion des Zustandsraumes, dessen Größe vom kartesischen Produkt $\mid A_1 \mid \times \mid A_2 \mid \times \ldots \times \mid A_n \mid$ der beteiligten Komponenten abhängt und somit exponentiell in Abhängigkeit von der Anzahl der Variablen des spezifizierten Systems anwachsen kann. In der Literatur [BBAF+01] werden Maßnahmen gegen das Problem der Zustandsraumexplosion aufgeführt. Hier werden die Techniken zur Datenabstraktion bzw. Einschränkung der Zustände angegeben, die für TLC relevant sind.

```
MODULE Actor
EXTENDS TLC, Sequences
CONSTANT
 m
VARIABLES
 x, sActor, qu

INIT ≜
 ∧ qu = ≪ ≫
 ∧ x = 0
 ∧ sActor = "init"

ACTIONS

enqueue(message) ≜
 ∧ sActor = "init"
 ∧ sActor′ ="enqueued"
 ∧ qu′ = append(qu, value)
 ∧ UNCHANGED x

dequeue() ≜
 ∧ sActor = "enqueued"
 ∧ sActor′ = "dequeued"
 ∧ x′ = head(qu)
 ∧ qu′ = tail(qu)

process() ≜
 ∧ sActor = "dequeued"
 ∧ sActor′ = "processed"
 ∧ UNCHANGED ⟨qu, x⟩

setValueCallReply() ≜
 ∧ sActor = "processed"
 ∧ sActor′ = "init"
 ∧ UNCHANGED ⟨qu, x⟩

Inv ≜  sActor = "init" ∨ sActor = "enqueued" ∨ sActor = "dequeued" ∨ sActor = "processed"

Next ≜ ∃ message ∈ m : ∨ process
                       ∨ enqueue(message)
                       ∨ dequeue
                       ∨ setValueCallReply

vars = ⟨⟨ x, state, qu ⟩⟩

Spec ≜ INIT ∧ □[Next]_vars ∧ WF_vars(enqueue) ∧ WF_vars(dequeue) ∧ WF_vars(process)
                                ∧ WF_vars(setValueCallReply)
```

Abbildung 5.6: Das Modul `Actor`

5.7.1 Ansätze und Werkzeuge für Model-Checking

Zur Durchführung des Model-Checking existieren zahlreiche Werkzeuge, die als Model-Checker bezeichnet werden. Zahlreiche Werkzeuge für Model-Checking werden in [BBAF+01] vorgestellt. *SMV* (Symbolic Model Verification) und *NuSMV* sind Model-Checker, die von E. Clarke und Emerson in Stanford entwickelt wurden. Beide beruhen auf der Eingabesprache CTL (Computational Tree Logic). NuSMV und SMV bieten Simulationsunterstüzung. SMV ist im WWW unter der URL http://www.cs.cmu.edu/modelcheck/smv.html verfügbar. Die Homepage für den Model-Checker NuSMV ist http://nusmv.irst.itc.it.

Spin ist ein Model-Checker, der von G. Holzmann bei den Bell Labs entwickelt wurde und über die der Programmiersprache C ähnliche Eingabesprache *Promela* verfügt. Spin stellt Simulations-

unterstüzung bereit. Spin ist im WWW unter der URL http://netlib.bell-labs.com/ netlib/spin/-whatisspin.html vorhanden. Für die Übersetzung von cTLA in *Promela* ist an der Universität Dortmund das Werkzeug cTLA2PC [RKK05] entwickelt worden.

Der Model-Checker *Uppaal* ist in Århus gebaut worden und verfügt über die Möglichkeit zur Spezifikation von Realzeiteigenschaften. Die Eingabe erfolgt über einen grafischen Editor. Der Model-Checker ist im WWW unter der URL http://www.uppaal.com verfügbar.

Der Model-Checker *COSPAN* besitzt die Eingabesprache S/R, die auf Büchi-Automaten beruht. COSPAN ist ein kommerzielles Produkt der Bell Labs.

Viele dieser Werkzeuge benutzen zur Zustandsrepräsentation optimierte Datenstrukturen, die auf sog. Binary Decision Diagrams (BDD) beruhen. Diese Datenstruktur erlaubt eine kompakte Darstellung eines Binärbaums, auf die effizient zugegriffen werden kann.

5.7.2 Temporal Logic Checker (TLC)

Der Model-Checker TLC, der von Yuan Yu bei Compaq entwickelt wurde [YML99] und gegenwärtig von Microsoft weiterentwickelt wird, unterstützt die Verifikation von TLA+-Spezifikationen. Der vollständig in Java implementierte Model-Checker ist gegenwärtig über Lamports Homepage [1] in der Version 2.001 verfügbar. Für das Werkzeug ist in [Lam03] eine gute Einführung vorhanden. TLC ist entwickelt worden, da die Eingabesprachen der existierenden Model Checker – etwa COSPAN mit S/R – wegen großer semantischer Unterschiede nicht geeignet waren TLA+-Spezifikationen zu übersetzen.

Diese Probleme sind durch TLC behoben worden. TLC ist nach [YML99] nicht dafür konstruiert worden, vollständige Systeme zu verifizieren, sondern um Entwurfsprozesse durch formale Verifikation zu unterstützen:

> „The systems that interest us, are too large and complicated to be completely verified by model checking. They may contain errors that can be found only by formal reasoning. We want to apply a model checker to finite-state models of the high-level design, both to catch simple design errors and to help us write a proof. Our experience suggests that using a model checker to debug proof assertions can speed the proof process. The specification language must therefore be well suited to formal reasoning."

TLC soll die Aufwände reduzieren, die entstehen, um Entwurfssprachen zur Verifikation in TLA+ zu übersetzen. Es wird deshalb folgendes Ziel in [YML99] formuliert:

> „We want to check the actual specification of a system, written by its designers. Getting engineers to specify and check their design while they are developing it should improve the entire design process. It will also eliminate the effort of debugging a translation from the design to the model checker's language. Engineers will write these specifications only in a language powerful enough to express the wide range of abstractions that they normally use."

5.7.3 Von TLC unterstützte Funktionalität

TLC unterstützt die Spezifikationssprache TLA+ mit einigen Einschränkungen, die in [Lam03] vorgestellt werden. TLC kann nur TLA+-Spezifikationen überprüfen, die eine kanonische Formel besitzen und in denen keine universelle temporale Quantifikation auftritt [KSH07]. Weiterhin unterstützt TLC nicht die präzise Semantik des CHOOSE-Operators, da diese sehr mächtig ist. Außerdem unterscheidet sich die Behandlung von Zeichenketten, die in TLA+ als Funktionen behandelt werden. Bei TLC sind Zeichenketten primitive Werte. Durch TLC werden die Verifikation von Invarianten, die Verifikation der Schritt-Simulation bei einem Refinement-Mapping sowie der Nachweis von Lebendigkeitseigenschaften für eine TLA+-Spezifikation unterstützt. Die Erkennung

[1] www.lamport.org

von Deadlocks wird angeboten, die einen Zustand beschreiben, in dem in der Zustandsübergangsrelation *Next* keine Aktion mehr enabled ist und auch kein Stotterschritt mehr möglich ist. Weiterhin werden TLA+-Teilmodule für TLC angeboten, die den Nachweis von Realzeiteigenschaften unterstützen.

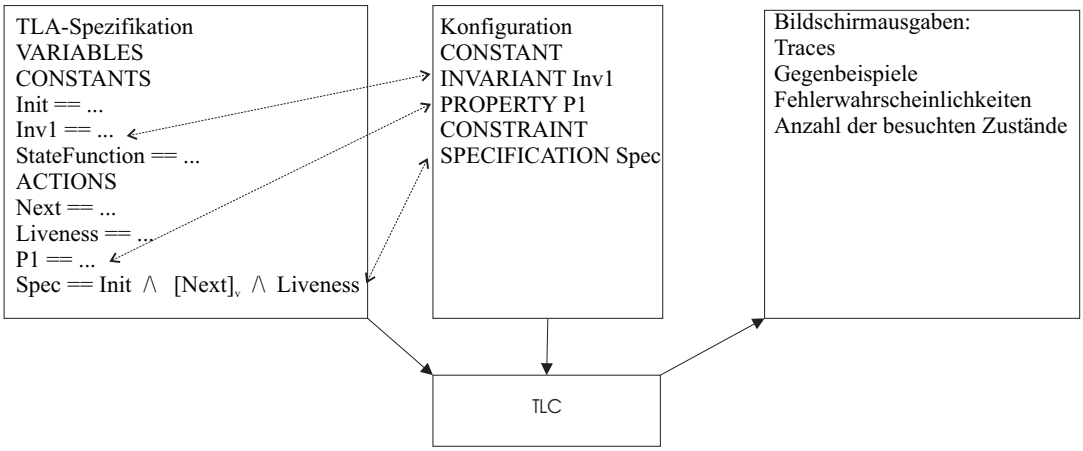


Abbildung 5.7: Die Architekturbestandteile der einzelnen Ebenen von TLC

Als Eingabe erhält TLC eine TLA+-Spezifikation und eine Konfigurationsbeschreibung, wie sie in Abbildung 5.7 angegeben sind. In der Konfigurationsbeschreibung werden alle Eigenschaften, die für die TLA+-Spezifikation mit TLC nachgewiesen werden sollen, angegeben. Die TLA+-Spezifikation enthält die Variablendefinitionen, die kanonische Formel mit dem Initialisierungsprädikat und der Zustandsübergangsrelation, die Definition von Invarianten sowie die Lebendigkeitseigenschaften. TLA+-Spezifikationen beschreiben nicht notwendigerweise endliche Zustandstransitionssysteme. Für TLC wird die Anzahl der Zustände durch die Auswahl eines Modells begrenzt, indem Konstanten in der Konfigurationsbeschreibung festgelegt werden, die etwa die möglichen Parameter für eine Aktion oder die Länge einer Warteschlange festlegen. Wie durch die gestrichelten Doppelpfeile in Abbildung 5.7 gezeigt, werden die Konstanten sowohl in der TLA+-Spezifikation als auch in der Konfigurationsbeschreibung angegeben. Mit dem Schlüsselwort SPECIFICATION wird der Bezeichner für die kanonische Formel in der Konfigurationsbeschreibung deklariert, falls diese Lebendigkeitseigenschaften besitzt. Alternativ hierfür können die Schlüsselworte INIT zur Deklaration des Initialisierungsprädikates und das Schlüsselwort NEXT zur Deklaration der Zustandsübergangsrelation verwendet werden, wenn die kanonische Formel keine Lebendigkeitseigenschaften enthält. Mit dem Schlüsselwort PROPERTY wird in der Konfigurationsbeschreibung eine Eigenschaft *P1* deklariert, die durch die zu verifizierende TLA+-Spezifikation *Spec* impliziert werden muss. Die Sicherheitseigenschaften von *P1* werden durch die Konjunktion $ImpliedInit \wedge [ImpliedAction]_{pvars}$ angegeben. Die Lebendigkeitseigenschaften werden durch *ImpliedTemporal* notiert:

$$Spec \equiv Init \wedge \Box[Next]_{vars} \wedge Temporal$$
$$Prop \equiv ImpliedInit \wedge \Box[ImpliedAction]_{pvars} \wedge ImpliedTemporal$$

Der Nachweis dieser Eigenschaft *Prop* wird für die zugehörigen Sicherheitseigenschaften, die durch die Sicherheitseigenschaften von *Spec* impliziert werden müssen, und die Lebendigkeitseigenschaften, die durch *Spec* impliziert werden müssen, getrennt durchgeführt:

$$Init \wedge \Box[Next]_{vars} \Rightarrow ImpliedInit \wedge \Box[ImpliedAction]_{pvars}$$
$$Spec \Rightarrow ImpliedTemporal$$

Alternativ können Invarianten in der angegeben Konfigurationsbeschreibung, die in der TLA+-Spezifikation definiert sind, durch das Schlüsselwort INVARIANT – gefolgt von dem Bezeichner

eines Zustandsprädikates – deklariert werden. Das Schlüsselwort CONSTRAINT erlaubt die Einschränkung der Zustände einer TLA-Spezifikation durch ein Zustandsprädikat. Jeder Zustand, der erreicht wird, muss dieses Zustandsprädikat erfüllen. Jeder Zustand, der die folgende Formel erfüllt, heißt erreichbar:

$$Init \wedge \ \Box[Next]_{vars} \wedge \Box Constr$$

Mit einer Einschränkung wird die Menge der erreichbaren Zustände begrenzt. Die Schlüsselwörter INVARIANT, PROPERTY und CONSTRAINT können auch im Plural formuliert werden, womit jeweils mehrere Eigenschaften bzw. Einschränkungen in einer durch Kommata separierten Liste angegeben werden. Im Verlauf des Model-Checking werden von TLC Ausgaben erzeugt. Falls alle in der Konfigurationsbeschreibung angegebenen Temporalformeln beim Model-Checking erfüllt sind, gibt TLC eine entsprechend Meldung und die Anzahl der erreichten Zustände aus. Falls eine Sicherheitseigenschaft nicht erfüllt wird, gibt TLC die Zustandsfolge (Trace) bis zu dem Zustand aus, in dem die Sicherheitseigenschaft verletzt ist. Für den Fall der Verletzung einer Lebendigkeitseigenschaft wird ein Gegenbeispiel ausgegeben.

5.7.4 Arbeitsweise von TLC

Zur Durchführung des Model-Checking werden von TLC zwei Datenstrukturen verwaltet, nämlich ein gerichteter Graph *Graph*, der eine Untermenge aller erreichbaren Zustände enthält, die TLC bisher gefunden hat, und einer Warteschlange *qu*, die alle Zustände aus *Graph* enthält, deren Nachfolgezustände noch nicht besucht worden sind. Im Folgenden wird der Algorithmus, den TLC beim Model-Checking verfolgt, kurz beschrieben. TLC generiert zunächst die Initialzustände, die sich aus dem Initialisierungsprädikat ergeben, und trägt sie in *qu* und *Graph* ein, wenn die Invarianten und die implizierten Initialisierungsprädikate (ImpliedInit) aller Einschränkungen erfüllt sind. Für jeden Initialzustand s, der außerdem diese Eigenschaft erfüllt, wird weiterhin eine Eigenkante $s \rightarrow s$ in *Graph* eingefügt.

Die Zustandsübergangsrelation wird durch TLC so umgeformt, dass sie aus einer Disjunktion *Nextmax* aus so vielen Subaktionen wie möglich zusammengesetzt ist. Anschließend werden ein oder mehrere Worker Threads – möglicherweise parallel – ausgeführt, von denen jeder den folgenden Subalgorithmus ausführt:

1. Zustand s = Head(qu) zuweisen.
2. Für jede Subaktion A_i aus *Nextmax* wird der nächste Zustand t generiert, sodass $s, t \models A_i$ gilt. Existiert kein neuer Zustand t für alle Aktionen A_i der umgeschriebenen Zustandsübergangsrelation, liegt ein Deadlock vor, der mit einem Trace über die bisher besuchten Zustände ausgegeben wird. Für jeden hierdurch generierten Zustand t wird Folgendes durchgeführt:
   - Es wird geprüft, ob alle Invarianten und ImpliedAction für den Schritt $s \rightarrow t$ gelten. Wenn nicht, hält TLC an und eine Fehlermeldung wird ausgegeben.
   - Wenn die Constraints im Zustand t erfüllt sind und die Aktionconstraints für den Schritt $s \rightarrow t$ erfüllt sind, wird die Kante $s \rightarrow t$ in *Graph* eingefügt. Wenn $t \notin Graph$ gilt, werden t und $t \rightarrow t$ in *Graph* eingefügt.

In Abbildung 5.8 ist der Ablauf des Model-Checking für die TLA+-Beispielspezifikation aus Abbildung 5.6 gezeigt. Die Konfiguration von TLC für diese Spezifikation ist in Abbildung 5.7 angegeben. Diese enthält neben der Deklaration der Spezifikation *Spec* die Deklaration der Invariante *Inv*. Weiterhin wird die Konstante m angegeben, die nur den Wert *d1* annehmen darf.

Bei der Verifikation dieses TLA+-Moduls werden die vier Zustände s_1 bis s_4 erzeugt und als Knoten in *qu* und *Graph* eingefügt. Außerdem werden die Kanten $s_1 \rightarrow s_2$, $s_2 \rightarrow s_3$, $s_3 \rightarrow s_4$ nacheinander in *Graph* aufgenommen. Zunächst wird der Initialzustand s_1 ermittelt und in *qu* eingefügt.

Es wird geprüft, ob *Inv* gilt. Da die Spezifikation keine Constraints enthält, entfällt deren Prüfung. Als Folgezustand wird s_2 berechnet und in *qu* eingefügt. Es wird wiederum geprüft, ob

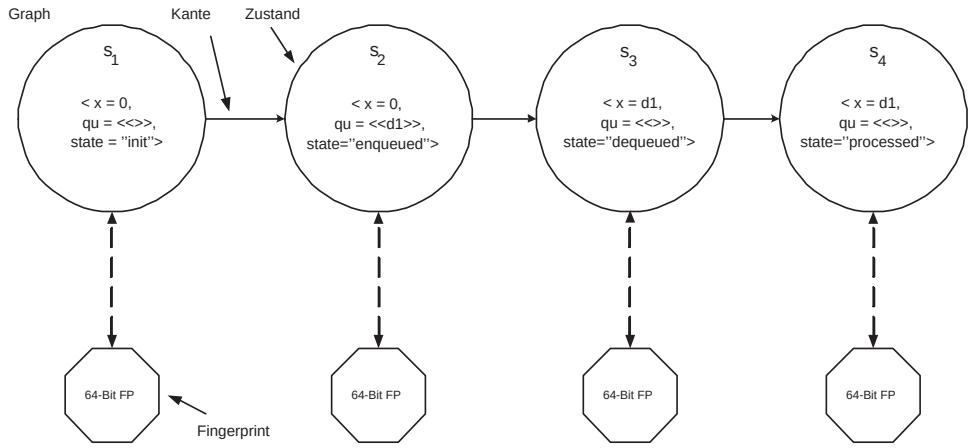


Abbildung 5.8: Die Ausführung von TLC am Beispiel von `ActorMod`

Inv gilt. Da *Inv* gilt wird $s_1 \rightarrow s_2$ in *Graph* eingefügt. Die Berechnung von s_3 erfolgt ebenso. Da s_4 ein Zustand ohne Nachfolger ist, terminiert der Algorithmus nach dessen Berechnung. Eine Terminierung der Berechnung von *Graph* erfolgt nur, wenn die Menge der erreichbaren Zustände endlich ist. Ist die Menge der erreichbaren Zustände unendlich, läuft TLC endlos weiter, bis alle Ressourcen verbraucht sind oder eine Unterbrechung durch den Anwender erfolgt. Mittels der Definition von Konstanten lässt sich die Anzahl der erreichten Zustände reduzieren. Dies entspricht der Idee der Datenabstraktion. Auch mit der Angabe von Einschränkungen (Constraints) lässt sich die Anzahl der erreichbaren Zustände reduzieren.

TLC verwendet eine explizite Zustandsrepräsentation anstatt BDDs, einer auf Binärbäumen beruhenden Datenstruktur. Jeder Zustand besitzt einen 64-Bit Fingerprint, wie in Abbildung 5.8 durch die Oktaeder gezeigt. Die Zugriffe auf Zustände werden nicht über den Hauptspeicher vorgenommen, sondern bei jedem Zugriff eines Zustandes erfolgt ein Plattenzugriff. Lamport weist in [YML99] daraufhin, dass TLC der einzige Model-Checker ist, der über eine solche Zustandsverwaltung verfügt.

Zur Durchführung von Verfeinerungsbeweisen wird in [Lam03] das neu geschaffene INSTANCE Schlüsselwort verwendet, mit dem die Instantiierung eines Modules, das als Feinsystem verwendet wird, möglich ist. Z. B. wird durch:

```
INSTANCE Actor sActor ← x
```

das Beispielmodul *Actor* instantiiert und die Zustandsvariable *sActor* durch die Zustandsvariable x substituiert.

Für TLC sind einige Hilfsmodule vorhanden, die teilweise direkt in Java programmiert sind, und durch das Schlüsselwort **extends** in eine TLA+-Spezifikation importiert werden. Mit den Modulen Naturals und Integer werden natürliche und ganze Zahlen unterstützt. Mit dem Modul Strings wird die Verarbeitung von Zeichenketten ermöglicht. Das Modul *Sequences*, das Sequenzen endlicher Länge und ist für die Beschreibung von Statecharts sehr nützlich. So wird die Sequenz der Zahlen 3, 2, 1 durch $\langle 1, 2, 3 \rangle$ dargestellt. Die leere Sequence wird durch << >> dargestellt. Es gibt Operatoren, um den Anfang (*head*), den Rumpf (*tail*) und die Verkettung ($\circ$) einer Sequence zu bilden [Lam03]. Mit dem Operator *SubSeq* werden Teilsequenzen einer vorgegebenen Länge aus einer Sequence gebildet. Auch das in [YML99] erwähnte Modul *Reals* wird in der gegenwärtigen Version unterstützt. *TLC* ist auch ein Modul, das nützliche Hilfsroutinen zum Debugging – etwa die Ausgabe eines Variablenwertes während des Model-Checking einer Spezifikation – besitzt. Weiterhin werden Module für grundlegende Datenstrukturen, wie Mengen und Bags, bereitgestellt. Abschließend soll das Modul *MCRealTime* genannt werden, das Aktionen und temporale Formeln, die in TLA+ spezifiziert sind, für die Verifikation von Realzeiteigenschaften anbietet.

Wenn die Lebendigkeitsanforderungen $SF_{var}(A_1) \wedge SF_{var}(A_2) \wedge \ldots \wedge SF_var(A_n)$ vorliegen, werden diese durch TLC folgendermaßen expandiert:

$$(\Diamond\Box\neg Enabled\langle A1\rangle_{var} \vee \Box\Diamond\langle A1\rangle_{var}) \vee \ldots \vee (\Diamond\Box\neg Enabled\langle An\rangle_{var} \vee \Box\Diamond\langle An\rangle_{var})$$

Für das Model-Checking wird dieser Ausdruck in disjunktive Normalform übersetzt, womit 2^n Disjunkte entstehen. Dabei wird das Model-Checking bei vielen Lebendigkeitsanforderungen mit starker Fairness erschwert. Bei schwacher Fairness liegt dieses Problem nicht vor, weil $WF_{var}(A) == \Box\Diamond(\neg Enabled\langle A\rangle_{var} \vee \langle A\rangle_{var})$ gilt. Daher sollte starke Fairness für Aktionen in TLC nur, wenn es absolut nötig ist, verwendet werden.

Kapitel 6

Compositional Temporal Logic of Actions (cTLA)

In diesem Kapitel werden zunächst die Möglichkeiten vorgestellt, um die Korrektheit von Mustern auf der Grundlage der formalen Spezifikationssprache cTLA [HK94a, HK94b, HK95, Her98, HK00] zu behandeln. Die Sprache cTLA ist eine Weiterentwicklung von TLA, die an der Universität Dortmund entwickelt wurde. TLA ist zunächst nicht kompositional, auch wenn bereits vor einigen Jahren entsprechende Erweiterungen vorgeschlagen worden sind. Mittels temporaler Logik formulierte Aussagen beschreiben Sicherheits- und Lebendigkeitseigenschaften eines Zustandstransitionssystems, das zur Modellierung eines existierenden Systems verwendet werden kann. In der Spezifikationssprache cTLA wird TLA um Konstrukte zur Prozesskomposition ergänzt, die eine Verwandtschaft zu den in der Spezifikationssprache LOTOS eingesetzten Prozesskonstrukten besitzen. Mit cTLA liegt nicht der einzige Ansatz zur kompositionalen Erweiterung von TLA vor. Mit der objektorientierten Spezifikationssprache DisCo2000[1] [JKS91] von der Universität Tampere existiert ein verwandter Ansatz, der ebenfalls auf TLA basiert.

In cTLA findet die Bildung von Prozessen und die Spezifikation von Realzeiteigenschaften besondere Beachtung, da diese für Steuerungssoftware besonders relevant sind. Weiterhin wird darauf eingegangen, wie die korrekte Verfeinerung zwischen Spezifikationen und Implementierungen nachgewiesen werden kann, um die Korrektheit von Verfeinerungsmustern zu verifizieren.

6.1 Syntax von cTLA

Mit cTLA spezifizierte Systeme werden aus Prozessen zusammengesetzt. Ein Prozess wird durch ein STS modelliert, dessen Struktur mittels eines Prozesstyps unter Verwendung einer programmiersprachen-ähnlichen Semantik angegeben wird. In Abbildung 6.1 wird beispielhaft der Prozesstyp *abstractActor*, der das Objekt einer abstrakten Regler-Task modelliert, angegeben. Dieser arbeitet in einem Regelkreis mit einem Sensor und einem Aktor zusammen. Der Header eines Prozesses besteht aus dem Schlüsselwort **PROCESS**, dem Prozesstypnamen – hier *abstractActor* – und einer Liste von generischen Parametern. Mit dem Importkonstrukt ist es möglich, die Inklusion von anderen Modulen – etwa einer Queue zur Aufnahme von Nachrichten – mit zusätzlichen Symboldefinitionen anzugeben.

Der Zustandsraum eines Prozesses wird durch die Variablen, die im **VARIABLES** Abschnitt der Abbildung 6.1 angegeben werden, aufgespannt. Der Zustandsraum eines Prozesstyps *abstractActor* besteht demnach aus den Variablen y, *sActor* und *qu*. Die Zustandsvariable y vom Datentyp Real modelliert den Wert eines Aktors. Die Zustandsvariable *sActor* modelliert den Kontrollzustand eines Prozesses vom Typ *abstractActor*. Mit der Zustandsvariable *qu* wird eine Warteschlange für die Abarbeitung von Aufrufen an ein Objekt modelliert.

[1] http://disco.cs.tut.fi/index.html

```
PROCESS abstractActor

IMPORT TLC, Sequences, Naturals

VARIABLES
 y : INTEGER;
 sActor : {"init", "enqueued", "dequeued", "processed"};
 qu : SEQUENCE OF INTEGER;

INIT ≜
 ∧ qu = ≪ ≫
 ∧ y = 0
 ∧ sActor = "init"

ACTIONS

enqueue(value : INTEGER) ≜
 ∧ sActor = "init"
 ∧ sActor′ ="enqueued"
 ∧ qu′ = append(qu, value)
 ∧ UNCHANGED y;

dequeue(cEO: String) ≜
 ∧ sActor = "enqueued"
 ∧ sActor′ = "dequeued"
 ∧ y′ = head(qu)
 ∧ qu′ = tail(qu);

process() ≜
 ∧ sActor = "dequeued"
 ∧ sActor′ = "processed"
 ∧ UNCHANGED ⟨qu, y⟩;

setValueCallReply(value : INTEGER) ≜
 ∧ sActor = "processed"
 ∧ sActor′ = "init"
 ∧ UNCHANGED ⟨qu, y⟩;

WF : enqueue, dequeue, process, setValueCallReply;

END abstractActor
```

Abbildung 6.1: Der Prozesstyp `abstractActor`

Ein Prädikat, dem das Schlüsselwort *INIT* vorangestellt wird, gibt die Menge der Initialzustände eines Prozesses an. Die Queue des Sensors wird initial auf den Wert << >> (leere Queue) gesetzt und die Zustandsvariable *sActor* auf *"init"*.

Der Abschnitt der Aktionendefinitionen wird durch das Schlüsselwort **ACTIONS** eingeleitet. Im Prozesstyp *abstractActor* sind das die Aktionen *enqueue*, *dequeue*, *process* und *setValueCallReply*. Die Variablen und Aktionen beschreiben das Zustandstransitionssystem eines beliebigen Aktors, der Werte für die Stellgröße vom Regler erhält.

Die Aktionen *enqueue* und *dequeue* dienen dem Einfügen und Entfernen von Nachrichten aus der Queue. Beim Schalten der Aktion *dequeue* wird auch die Zustandsvariable y auf den aktuellen Wert gesetzt. Die Aktion *setVallCallReply* liefert eine parameterlose Antwort an den Aufrufer.

Mit dem Initialisierungsprädikat und den Aktionendefinitionen werden die Sicherheitseigenschaften des Prozesses *setValueCallReply* beschrieben. Weiterhin besteht eine Lebendigkeitsanforderung an die Aktion *process* dieses Prozesses, die mit dem Schlüsselwort WF gekennzeichnet ist und die Behandlung der Aktion *process* mit schwacher Fairness vorschreibt.

In Abbildung 6.2 wird außerdem der Prozesstyp *SequenceDiagram* gezeigt, der verwendet wird, um das Verhalten von Sequenzdiagrammen mit dem im Abschnitt 3.4.4 vorgestellten Sprachumfang zu spezifizieren. Dieser besitzt den Prozessparameter *SetOfTraces*, um zulässige Traces anzugeben.

Initialisiert wird die Zustandsvariable *currentTrace* mit der leeren Sequence << >> im Initialisierungsprädikat des Prozesses. Der Prozess besitzt diese Aktion *permittedActionOfSD*, um zu prüfen, ob ein einzelner Ereignisauftritt einer UML-Action durch den *Trace* zulässig ist. Um den durch eine Action bereitgestellten Ereignisauftritt entgegenzunehmen, besitzt die Aktion den Parameter *cEO*.

```
PROCESS SequenceDiagram(SetOfTraces : SUBSET(Sequence))
 IMPORT Sequences;

CONSTANT

MaxTraceLength ≜ 10  ! constant defining maximal length of a trace

setOfPossibleTraces ≜ ! compute the transitive closure of SetOfTraces
UNION { {SubSeq(d, 1, 2) : d ∈ SetOfTraces},
        {SubSeq(d, 1, 3) : d ∈ SetOfTraces}, ...
        {SubSeq(d, 1, MaxTraceLength) : d ∈ SetOfTraces}};


VARIABLES
 currentTrace : Sequence;

INIT ≜ currentTrace = ≪ ≫;

ACTIONS

    permittedActionOfSD(cEO : EventOccurrence) ≜
    currentTrace ∘ cEO ∈ setOfPossibleTraces ∧
    currentTrace′ = currentTrace ∘ cEO;

END
```

Abbildung 6.2: Der Prozesstyp `SequenceDiagram`

Ähnlich wie in LOTOS [Hog89] unterstützt cTLA die Komposition von Systemen aus Prozessen. Die Prozesse interagieren durch gemeinsame Aktionen (engl. Joint Actions). Mittels Aktionenparametern von gemeinsamen Aktionen wird der Datenaustausch zwischen einzelnen Prozessen modelliert. Die Variablen eines Prozesses sind privat, auf sie kann daher nicht durch einen fremden Prozess zugegriffen werden.

Auch das STS eines Systems wird durch einen Prozesstyp spezifiziert. Der aus den Variablen aller Prozesse gebildete Vektor bildet den Systemzustand. Ein Systemverhalten ist eine Sequenz aus Systemzuständen, die den einzelnen Systemaktionen entsprechen. Jede Systemaktion wird durch die logische Konjunktion von Prozessaktionen gebildet. In einer solchen Konjunktion wird jeder Prozess entweder durch eine tatsächliche Prozessaktion oder durch die Pseudo-Aktion **stutter** einbezogen, die angibt, dass ein Prozess bei der betrachteten Systemaktion keine Zustandsänderung vornimmt.

Es liegen somit drei Fälle für das Schalten von gemeinsamen Aktionen vor:

- Die Aktionen unterschiedlicher Prozesse schalten synchron. Die Systemaktion *ActionDequeue* schaltet beispielsweise, falls der Parameterwert für *cEO* dies zulässt, wobei die Aktionen *abstActor.dequeue* und *sd.permittedActionOfSD* gemeinsam schalten.

- Die Blockade von Aktionen aus anderen Prozessen findet statt, wenn gekoppelte Aktionen nicht schalten können. Falls z. B. die Parameterwerte der Aktionen *abstActor.dequeue* und *sd.permittedActionOfSD* für *cEO* nicht übereinstimmen, kann eine Blockade der Systemaktion *ActionDequeue* erfolgen.

- Das Schalten von zu unterschiedlichen Prozessen gehörenden Aktionen erfolgt ohne gegenseitige Interaktion, weil Prozesse stottern. Für den Fall, dass die Aktionen *abstActor.dequeue* und *sd.permittedActionOfSD* gemeinsam schalten, stottert der Prozess *aT*.

Die in Abbildung 6.3 angegebene Spezifikation *ActorComposition* modelliert ein System, das aus den Prozessen *Actor* und *abstractTask* des abstrakten Regelkreises zusammengesetzt ist. In dem Teil des Prozesses, der durch das Schlüsselwort **PROCESSES** eingeleitet wird, erfolgt eine Deklaration der einzelnen Prozesse des Systems.

```
PROCESS ActorComposition
 PROCESSES
  aT : abstractTask(k1); ! abstrakte Task
  abstActor : abstractActor();  ! abstrakter Aktor
  sd : permittedActionOfSD; ! Prozess für Sequenzdiagramm

ACTIONS

SetValueCallActor(value : INTEGER, cEO : EventOccurrence) ≙ ! SysAct to model
getValue call
     ∧ aT.actionSetValueCall(value, cEO)
     ∧ abstActor.enqueue(value)
     ∧ sd.permittedActionOfSD(cEO);

ActorDequeue(cEO : EventOccurrence) ≙ ! SysAct modelling dequeuing in sensor
process
     ∧ aT.stutter
     ∧ abstActor.dequeue(cEO)
     ∧ sd.permittedActionOfSD(cEO);

ActorProcess(cEO : EventOccurrence) ≙ ! SysAct to model sensor processing
     ∧ aT.stutter
     ∧ abstActor.process()
     ∧ sd.stutter;


getCallActorReply(value : INTEGER, cEO : EventOccurrence) ≙ ! SysAct to model
sensor reply
     ∧ aT.enqueue(value)
     ∧ abstActor.setValueCallReply(value, cEO)
     ∧ sd.permittedActionOfSD(cEO);

END ActorComposition
```

Abbildung 6.3: Der Prozesstyp `ActorComposition` als Beispiel zur Komposition von Prozessen

So ist der Prozess *aT* eine Instanz des Prozesses *abstractTask* und der Prozess *abstActor* ist eine Instanz des Prozesses *abstractActor*. Im Abschnitt ACTIONS dieses Prozesses werden die Systemaktionen durch Konjunktionen von Prozessaktionen deklariert. So werden die lokalen Prozessaktionen *process* der Prozesse *aS* und *aT* zur Systemaktion *ActorProcess* zusammengeschaltet, während der Prozess *abstractTask* einen Stotterschritt unternimmt.

Durch die Spezifikationstechnik cTLA wird das Superpositionsprinzip unterstützt, das garantiert, dass eine Eigenschaft, die durch ein System oder ein Subsystem erfüllt wird, auch eine Eigenschaft von jedem System ist, das den Prozess oder das Subsystem enthält [KSJ88]. Dies ist notwendig für die Strukturierung der Verifikation in Subsystem-Implikationen. In Bezug auf Sicherheitseigenschaften, die nur Initialzustände und Zustandstransitionen einschränken, wird Superposition relativ einfach garantiert, weil die Zustandsvariablen privat für jeden Prozess gelten und nicht durch fremde Prozesse zugreifbar sind.

In Bezug auf Lebendigkeitseigenschaften ist die Behandlung der Superposition komplizierter. In einem System können Prozessaktionen mit den Aktionen anderer Prozesse gekoppelt werden. Somit kann die Umgebung eines Prozesses eine Prozessaktion blockieren und aufgrund der Blockade können die Fairnessannahmen für diese Prozessaktion verletzt werden, weil diese Prozessaktion wegen ihrer Kopplung mit anderen Aktionen nicht schalten kann. Im Gegensatz zu TLA, bei dem Fairnessanforderungen direkt in die kanonische Formel übernommen werden, benutzt cTLA nur bedingte Fairnessanforderungen, die eine Einschränkung bzgl. der Fairnessanforderungen an eine

Prozessaktion bilden. Die WF/SF-Konstrukte von cTLA beziehen sich auf Zeiträume, in denen eine Prozessaktion sowohl schaltbereit ist als auch nicht durch die Umgebung des Prozesses blockiert wird. Somit können Blockierungen von Aktionen durch Prozesse der Umgebung keine Verletzung der Fairnesseigenschaften einer Prozessaktion herbeiführen. Beispielsweise hat die Prozessaktion *process* nur die Lebendigkeitsanforderung, dass sie immer wieder ausgeführt werden muss und der Prozess *abstAct* hat nur die Lebendigkeitsanforderung, dass die Aktion *process* ebenfalls immer wieder ausgeführt und nicht zu oft blockiert wird.

Einerseits unterstützt die Einschränkung von Lebendigkeitseigenschaften auf die bedingte Fairness direkt die Superposition. Andererseits sind bedingte Fairnessannahmen manchmal nicht hinreichend, um absolute Lebendigkeitseigenschaften auszudrücken, die von besonderem Interesse für den Systementwurf sind. Deshalb muss diesem Sachverhalt besondere Beachtung gewidmet werden. Häufig können absolute Lebendigkeitseigenschaften durch Verwendung einer zusätzlichen Bedingung verhindern, dass die Umgebung des Prozesses die fairen Aktionen zu oft blockiert.

6.2 Semantik von cTLA

In TLA beschreiben kanonische Formeln die Sicherheits- und Lebendigkeitseigenschaften von Zustandstransitionssystemen. Mit Inferenzregeln wird die syntaktische Deduktion von gültigen Formeln zum Beweis von Sicherheits- und Verfeinerungseigenschaften unterstützt. In cTLA werden die Begriffe und Beweistechniken aus TLA übernommen. Es ist eine Erweiterung von TLA, die den expliziten Begriff des Prozesses, des Prozesstyps und der Prozesskomposition kennt, wobei der Superpositionscharakter der Komposition von besonderem Interesse ist. Offensichtlich ist das unterschiedliche Erscheinungsbild von cTLA- und TLA-Spezifikationen, weil in cTLA die kanonischen Formeln nicht explizit aufgeführt werden.

Die TLA-Formel, die einer Prozessinstanz entspricht, hängt von den Parameterbelegungen und der Definition des Prozesstyps ab. Die Parameter der Prozesstypen sind generisch. Es gilt die übliche Konvention, dass die formalen Vorkommen von Parametern in der Spezifikation ersetzt werden. Für Prozesstyp-Definitionen gibt es die beiden Formen zur Spezifikation von einfachen Prozessen und von Systemen. In beiden Fällen können die TLA-Formeln der Prozessinstanzen in flacher Form angegeben werden und beziehen sich direkt auf die Definitionen von Prädikaten und Aktionen der jeweiligen Prozesstypen.

Für die Verdeutlichung eines einfachen Prozesstyps sei *abstAct* eine Prozessinstanz des Prozesstyps *abstractActor* aus Abbildung 6.1. Die folgende TLA-Formel $\widehat{abstAct}$ stimmt mit *abstActor* überein. Hierbei geben die Symbole *INIT* und die Aktionen des Prozesses *enqueue*, *dequeue*, *process*, *setValueCallReply* und *abstActor* an:

$$
\begin{aligned}
\widehat{abstAct} = \; & INIT \;\wedge\; \Box\,[\\
& \exists\, value \in Real :: enqueue(value) \;\vee \\
& \exists\, cEO \in EO :: dequeue(cEO) \;\vee \\
& \exists\, cEO \in EO :: process(cEO) \;\vee \\
\exists\, value \in Real, \exists\, cEO \in EO :: & \; setValueCallReply(value, cEO)]_{(sActor,y,qu)} \;\wedge \\
& \forall\, cEO \in EO :: WF_{(y,sActor,qu)}(process \;\wedge\; cEO \in e_{process})
\end{aligned}
$$

Mit $\widehat{abstAct}$ wird eine gewöhnliche TLA-Formel angegeben, die die Sicherheits- und Lebendigkeitseigenschaften eines Zustandstransitionssystems mit den Zustandsvariablen *y*, *sActor* und *qu* beschreibt. Bis auf die letzte Zeile beziehen sich alle Zeilen auf Sicherheitseigenschaften des Prozesses und definieren die Initialzustände und die Zustandsübergangsrelation des STS. In der letzten Zeile wird die Lebendigkeit durch Fairnessannahmen der Zustandsübergangsrelation hinzugenommen, wie am Ende des Abschnitts 6.1 bereits erläutert. Um Spezifikationen, die widersprüchliche Fairnessangaben besitzen können und die die Nichterfüllbarkeit der Spezifikation zur Folge haben, auszuschließen, werden so genannte Umgebungsbereitmengen-Variablen eingeführt, wodurch direkt das Superpositionsprinzip unterstützt wird. Verletzungen der Fairnessanforderungen würden die Superposition beeinträchtigen. Während sich Fairness-Operationen gewöhnlich direkt auf die Aktionen der Zustandsübergangsrelation beziehen, wird die schwache Fairness nicht auf die Aktion *process*, sondern auf eine Sub-Relation nämlich auf die bedingte Aktion

$process(cEO) \wedge cEO \in e_{process}$ bezogen, in der die zusätzliche Umgebungsbereitmengen-Variable $e_{process}$ adressiert wird. Es wird angenommen, dass für jede faire Aktion eines Prozesses eine derartige Umgebungsbereitmengen-Variable existiert. Die Variable wird zwischen dem Prozess und ihrer Umgebung geteilt. Sie wird durch die Umgebung geschrieben und durch den Prozess gelesen. Sie agiert als eine Abstraktion der Prozessumgebung und gibt die gegenwärtige Schaltbereitschaft der Umgebung für die Aktion an. Der Wert der Variablen ist die Menge der Werte von Aktionenparametern, für die die Umgebung gegenwärtig die Schaltung der Aktion zulässt. Für den Fall, dass die Umgebungsbereitmengen-Variable $e_{process}$ leer ist, wird angenommen, dass die Umgebung die Aktion *process* unter jeder möglichen Parametrierung blockiert. Im Gegensatz dazu beschreibt die TLA-Formel $\widehat{abstAct} \wedge abstActe_{process} = cEOType$ – wobei *cEOType* die Menge aller möglichen Parametrierungen des Aktionenparameters *cEO* angibt – ein STS, das den Prozess *abstAct* in einer Umgebung modelliert, die immer Zustandstransitionen toleriert, die durch die Aktion *process* ausgelöst werden.

Die TLA-Formel eines Systems wird an einer Erweiterung des Beispiels erklärt. Dazu wird auf den Prozesstyp *abstractActor* aus Abbildung 10.5 Bezug genommen. Wie bereits angegeben, werden Instanzen dieses Prozesstyps aus je drei Prozessinstanzen gebildet. Weil alle anderen Prozesse außer *abstractActor* keine Lebendigkeitseigenschaften ausdrücken, beziehen sich Fairnessannahmen nur auf die Aktionen von *abstractActor*. Im Folgenden sei $aC : ActorComposition$ eine Instanz dieses Systemtyps. Diese entspricht folgender kanonischer TLA-Formel $\widetilde{aC}$:

$\widetilde{aC} = aT.INIT \wedge sd.INIT \wedge abstAct.INIT \wedge$
$\Box[$
$\exists\, cEO \in EO :: permittedActionOfSD(cEO)$
$\exists\, cEO \in EO, \exists\, value \in Real :: actionSetValueCall(cEO, value)$
$\exists\, value \in Real :: enqueue(value) \vee$
$\exists\, cEO \in EO :: dequeue(cEO) \vee$
$\exists\, cEO \in EO :: process(cEO) \vee$
$\exists\, cEO \in EO :: setValueCallReply(cEO)]_{(x,y,sync,qu,sActor,qu)} \wedge \forall\, cEO \in EO ::$
$WF_{(aT.x,\ldots,aT.state,abstAct.sActor,\ldots,abstAct.qu,sd.currentTrace)}(process(cEO) \wedge cEO \in e_{process})$

Die Formel $\widetilde{aC}$ referenziert die Definitionen der initialen Prädikate der Prozessinstanzen. Somit steht *abstAct.INIT* für das Initialisierungsprädikat von *abstractActor*. Des Weiteren verwendet $\widetilde{aC}$ die Aktionendefinitionen des Prozesstyps *abstractComposition*, welche Konjunktionen von Prozessaktionen sind. Das sind hier $sd.permittedActionOfSD(cEO), \ldots, abstActor.process(cEO)$. Weil die Aktion *ActorProcess* die faire Prozessaktion *abstActor.process* enthält, wird sie auch durch ein Fairness-Statement begleitet. Die letzte Zeile der Formel $\widetilde{aC}$ gibt eine schwache Fairnessanforderung für die Systemaktion *ActorProcess* an. Wie in einfachen Prozessen, ist die Fairnessannahme durch eine Umgebungsbereitmengen-Variable bedingt, wobei das Symbol $e_{process}$ diese zugehörige Umgebungsbereitmengen-Variable für die Systemaktion *ActorProcess* notiert.

Für die Argumentation bzgl. der Prozesskompositionen werden nicht nur TLA-Formeln der Form *aC* verwendet. Weiterhin existiert eine TLA-Formel, die eine Konjunktion der TLA-Formeln des Prozesses ist, der sie bildet. Somit ist die kompositionale Formel:
$\widetilde{SWS} = \widehat{abstractTask} \wedge \widehat{SD} \wedge \widehat{abtractActor} \wedge SCC \wedge$
$\Box\, (abstAct.e_{process} = \{cEO \in cEOType\,, \exists\, value \in ValueType ::$
$Enabled(abstAct.process(cEO)) \wedge \ldots \wedge Enabled(sd.possibleMessageOfSD(cEO, value))\}$
Zusätzlich zur Formel des Prozesses, durch den sie gebildet wird, fügt die kompositionale Formel zwei Invarianten zusammen. Die so genannte Kopplungsbedingung *SCC* drückt die Sicherheitsbedingungen aus, die sich aus der speziellen Aktionenkopplung eines Systems ergeben. So wird zum Beispiel im Hinblick auf die Aktion *abstActor.process(cEO)* angegeben, dass diese Aktion in Kombination mit dem gleichzeitigen Schalten der Aktionen *sd.permittedActionOfSD(cEO)* auftritt. Die letzte Invariante definiert die Umgebungsbereitmengen-Variable $abstAct.e_{process}$ des Prozesses *abstActor* als Zustandsfunktion. Die Aktion *process* des Prozesses *abstActor* wird durch die Umgebung von *abstActor* für den Parameterwert *cEO* toleriert, genau dann wenn die Umgebung von *SWS* die Systemaktion *ActorProcess* zulässt und alle anderen Prozesse von *SWS*, die mit dem Prozess *abstActor* über die Systemaktion *ActorProcess* gekoppelt sind, schaltbereit sind.

Außer der Syntax und der oben angegebenen TLA-Transformationsregeln beinhaltet die cTLA-

Sprachdefinition einige Restriktionen, die sich auf den Inhalt von Systemaktionen – speziell bezüglich des Auftretens von fairen Aktionen – beziehen. Diese Bedingungen drücken die Annahmen aus, dass faire Aktionen disjunkt sind und jede faire Aktion in genau einer Systemktion auftritt. Unter diesen Bedingungen kann man beweisen, dass die kompositionale TLA-Formel einer Systeminstanz äquivalent zur direkten kanonischen Formel der Systeminstanz ist. Somit gilt nach [HK00] immer $\widehat{aC} \Leftrightarrow \widetilde{aC}$. Außerdem bedeutet dies, dass eine kompositionale Formel $\widetilde{aC}$ in eine Prozessformel $\widehat{aC}$ umgewandelt werden kann.

Aus der Äquivalenz der komposionalen Formel mit der direkten kanonischen Formel einer Systeminstanz kann durch Inferenz geschlossen werden, dass die kompositionale Formel frei von Widersprüchen ist. In Kombination mit der kompositionalen Formel, die die Formel des sie bildenden Prozesses verbindet, kann geschlossen werden, dass durch die Prozesskomposition die konsistente Konjunktion der Prozesse impliziert wird. Daher impliziert eine Systemformel die Formel jedes sie bildenden Prozesses. Das bedeutet, dass die Prozesskomposition idealen Superpositionscharakter besitzt, weil die Sicherheits- und Lebendigkeitseigenschaften des Prozesses auch Eigenschaften des sie beinhaltenden Prozesses sind. Weil die logische Konjunktion kommutativ und assoziativ ist, kann die Superposition nicht nur auf einzelne Prozesse sondern auch auf Subsysteme angewendet werden.

Superposition erleichtert somit die formale Verifikation von Systemeigenschaften. Um zu zeigen, dass ein System die Eigenschaften besitzt, die durch eine TLA-Formel ausgedrückt werden, ist es ausreichend, ein Subsystem *Sys* von S zu finden, für das die TLA-Formel $Sys \Rightarrow P$ – wobei P eine Eigenschaft angibt – gezeigt werden kann. Diese Implikation bildet ein Theorem. Hierdurch ist die breite Anwendbarkeit von Theoremen gewährleistet, die bestimmte Eigenschaften verifizieren.

6.3 Behandlung von Zeiten in cTLA

Abadi und Lamport haben Zeiten und Timer (Realzeit-Uhren) in TLA [Lam03], [AL91b] eingeführt. Die dort vorgestellten Ansätze unterstützen allerdings keine Kompositionalität. Die wesentliche Idee der Behandlung von Realzeit in cTLA-Prozessen [HK97a], [HK97b] besteht in der Verwendung einer reellwertigen Zustandsvariablen, deren Wert von einer Aktion *Tick* in kleinen Schritten hochgezählt wird. Weiterhin können minimale und maximale Wartezeiten für das Schalten von Aktionen modelliert werden, wobei prinzipiell gilt, dass Wartezeiten in Zuständen konsumiert werden. In einer realzeitfähigen cTLA-Spezifikation wird jede Aktion mit einem entsprechenden Schlüsselwort versehen. Diese Schlüsselwörter und die zugehörigen Definitionen werden in den nachfolgenden Abschnitten vorgestellt.

6.3.1 Globale Zustandsvariable *now*

Um Realzeitmechanismen wie Zeitgeber zu spezifizieren, wird die globale Zustandsvariable *now* verwendet. Hierbei bedeutet die Eigenschaft global, dass auf die Zustandsvariable auch aus anderen Prozessen zugegriffen werden darf. Sie kann in jedem Prozess in der Initialisierungsbedingung sowie den einzelnen Aktionen verwendet werden. Die Zustandsvariable *now* ist vom Datentyp Real und wird im Initialisierungsprädikat stets auf den Wert null initialisiert.

6.3.2 Aktion *Tick*

Die globale Uhrenvariable *now* wird durch eine Aktion *Tick* in kleinen Schritten und lebendig inkrementiert. Das Zeitraster, welches durch Schaltfolgen der Aktion *Tick* vorgegeben wird, kann beliebig fein eingestellt werden. Die *Tick* Aktionen jedes Prozesses werden miteinander gekoppelt. Für Zeitschritte der Aktion *Tick* wird eine obere Schranke (*clockresolution*) angegeben. Hierdurch wird garantiert, dass die Zeit in ausreichend kleinen Schritten fortschreitet. Auf die Zustandsvariable *now* darf nur durch die Aktion *Tick* eines Prozesses zugegriffen werden. Für die Aktion *Tick* wird immer starke Fairness gefordert.

Die Aktion *Tick* ist auch für die Modellierung kontinuierlicher Vorgänge von Bedeutung [HGK98].

6.3.3 Zeno-Verhalten

Durch explizite Uhren modellierte Realzeiteigenschaften unterliegen der Gefahr, so genanntes Zeno-Verhalten zu beschreiben. Der Begriff geht auf den griechischen Philosophen Zeno zurück, der das Paradoxon aufgestellt hat, dass ein Pfeil zunächst die erste Hälfte zum Ziel, dann das nächste Viertel der Distanz, anschließend das nächste Achtel usw. zurücklegt, was dazu führt, dass das Ziel nicht in endlicher Zeit erreicht wird.

Bei einer TLA-Spezifikation liegt Zeno-Verhalten vor, wenn eine Uhrenvariable einen bestimmten Wert niemals überschreitet bzw. immer unterhalb einer Zeitschranke bleibt, was zur physikalischen Interpretation führt, dass die Zeit stehen bleibt. Ein Beispiel für Zeno-Verhalten ist ein Timer, dessen Wert sich von 0.9 auf 0.99 und anschließend auf 0.999 usw. erhöht, aber niemals 1 erreicht. Dieses in der Realität nicht erfüllbare Verhalten kann z. B. dadurch auftreten, dass eine Aktion A im geforderten Zeitintervall $[t_{min}, t_{max}]$, das die minimalen und maximalen Wartezeiten einer Spezifikation angibt, niemals schaltbar ist. Um Zeno-Verhalten auszuschließen, machen Abadi und Lamport [AL91b] t_{min} und t_{max} nicht von der aktuellen Zeit *now*, sondern von speziellen Timern abhängig. Diese Timer schreiten nur fort, wenn die Aktion A schaltbar ist. Die Timer können sich somit der oberen Zeitschranke nur annähern, wenn die Aktion A schaltbar ist. Dadurch wird Zeno-Verhalten ausgeschlossen.

Lamport schließt in [Lam03] Zeno-Verhalten einer TLA-Spezifikation dadurch aus, dass die Lebendigkeitseigenschaften der kanonischen Formel, die die Zustandsübergangsrelation *Next* besitzt, mit der Fairnessanforderung

$$NZ == \forall r \in \mathbb{R} : WF_{now}(Next \wedge (now' > r))$$

konjugiert wird. Dadurch wird sichergestellt, dass die durch die Uhrenvariable *now* beschriebene Zeit über alle Grenzen wächst.

Eine Zeno-Spezifikation besitzt ein endliches Verhalten, das die Sicherheitseigenschaften erfüllt, aber nicht auf ein unendliches Verhalten, das sowohl die Sicherheitseigenschaften als auch *NZ* erfüllt, übertragen werden kann. Eine Spezifikation, die Zeno-Verhalten zeigt, ist vermutlich nicht korrekt, weil die Bedingungen, die zur Begrenzung der Zeit führen, das System auf unbeabsichtigte Weise einschränken. Mit Non-Zeno wird eine Spezifikation bezeichnet, die kein Zeno-Verhalten zeigt.

6.3.4 Aktionen mit minimaler Wartezeit

Das Konstrukt MIN TIME stellt sicher, dass eine Aktion erst schaltet, wenn sie mindestens über einen bestimmten Zeitraum hinweg schaltbereit war. Hierdurch wird garantiert, dass zwischen dem Betreten und Verlassen eines Zustandes bestimmte zeitliche Mindestabstände vorliegen (minimale Verweildauer). Bei der Spezifikation müssen diese Zeiten als persistent oder volatil (veränderlich) spezifiziert werden. Bei einer volatilen, minimalen Wartezeit wird für eine Aktion a das folgende Konstrukt, das in [HK97a, HK97b] definiert wird, für die Aktion verwendet:

V MIN TIME a : mt

Dies bedeutet, wenn a schaltet, war es für einen Zeitraum der Länge *mt* ohne zwischenzeitliches Schalten schaltbereit.

Die Übersetzung des Konstruktes in flaches cTLA wird in Abbildung 6.4 gezeigt. Es werden die Aktionen *a*, *act* und *Tick* angegeben. Die Aktion *act*, an die keine Realzeitanforderungen gestellt werden, besitzt die Schaltbedingung *actenab* und den Schritt *actnext*. Für die Aktion *a* existieren Anforderungen bzgl. der minimalen Wartezeit. Die Bezeichner *aenab* und *anext* geben die Schaltbedingung und den Schritt der Aktion *a* an. Mit *vta* wird ein Feld bezeichnet, das für jede Belegung des Parameters p der Aktion *a* speichert, für wie viele Zeiteinheiten die Aktion mit der Parameterbelegung seit dem Initialzustand oder der letzten Schaltung der Aktion $a(p)$ schaltbereit war. Die Aktion $a(p)$ darf nur ausgeführt werden, wenn $vta[p] \geq mt$ gilt. Bei jeder Ausführung von $a(p)$ wird vta(p) auf null zurückgesetzt, da der Zeitraum, bis zu dem *a* erneut schalten darf, von vorne beginnt. Das geschieht durch Zuweisung von null an $vta(p)$ mit dem Konstrukt

$pta' = [vta; Except\ p \mapsto 0]$. In der Aktion *Tick* wird der Wert von *vta[p]* um die Größe des Zeitschritts $t - now$ erhöht, wenn die Aktion $a(p)$ schaltbar ist. Ansonsten stottert die Zustandsvariable $vta[p]$. Neben der Schaltbedingung *aenab* von a wird auch die Umgebungsbereitmengen-Variable e_a berücksichtigt, die angibt, ob die Schaltung der Aktion a(p) durch die mit ihr gekoppelten Aktionen aus anderen Prozessen toleriert wird. Der Wert von *vta[p]* zeigt den Zeitraum an, den die Aktion *a(p)* seit ihrer letzten Schaltung oder seit dem Initialzustand schaltbar war.

Die Spezifikation einer persistenten, minimalen Wartezeit für die Aktion a fordert, dass a insgesamt über Zeiträume, deren Summe *mt* beträgt, schaltbereit war, auch wenn zwischenzeitlich Unterbrechungen der Schaltbereitschaft aufgetreten sind. Dazu wird das folgende Konstrukt verwendet:

P MIN TIME a : mt

Die Umsetzung des P MIN TIME Konstruktes wirkt sich nur auf die Aktion *Tick* aus. In Abbildung 6.5 ist gezeigt, wie die Aktion *Tick* für eine persistente, minimale Wartezeit modifiziert wird. Dazu bleibt im ELSE-Zweig der IF-Anweisung die Zustandsvariable *pta*, die als Zeitgeber für die persistente Wartezeit fungiert, unverändert, auch wenn die Aktion zwischenzeitlich nicht schaltbereit ist.

```
PROCESS RT1
VARIABLES
v : type;
now : Real; ! Uhrenvariable
vta : [ptype ↦ Real]; ! Zeitgeber

CONST clockresolution ≙ cr; ! Maximale Größe f. Zeitschritt

INIT ≙ vip(v, now) ∧
       now = 0 ∧
       vta = [p ∈ ptype ↦ 0];
ACTIONS
     Tick( t : Real) ≙ ! Zeitschritt
                       t > now ∧
                       t ≤ now + clockresolution ∧
                       vta' = [p ∈ ptype ↦ IF (aenab(v, now, p) ∧ p ∈ e_a)
                                           THEN vta[p] + t - now
                                           ELSE 0 ∧
                       now' = t ∧
                       v' = v;

     act( p : ptype) ≙
        actenab(v, now, p) ∧ ! Schaltbedingung f. act
        v' = actnext(v, now, p); ∧  ! Neuer Wert für Zustandsvariable v
        UNCHANGED ⟨now, vta⟩ ; ! now und vta stottern

     a( p : ptype) ≙
        aenab(v, now, p) ∧ ! Schaltbedingung f. a
        vta[p] > mt ∧
        v' = anext(v, now, p) ∧ ! Neuer Wert für Zustandsvariable v
        now' = now ∧
        vta' = [vta; Except p ↦ 0]; ! Rücksetzen des Timers

     SF : Tick;
END
```

Abbildung 6.4: Prozesstyp RT1 als Beispiel für volatile, minimale Wartezeiten

```
Tick( t : Real) ≜ ! Zeitschritt
                  t > now ∧
                  t ≤ now + clockresolution ∧
                  pta′ = [p ∈ ptype ↦ IF (aenab(v, now, p) ∧ p ∈ e_a)
                                     THEN pta[p] + t - now
                                     ELSE pta[p] ∧
                  now′ = t ∧
                  v′ = v;
```

Abbildung 6.5: Die Aktion `Tick` für persistente, minimale Wartezeiten

6.3.5 Aktionen mit maximaler Wartezeit

Das Konstrukt **MAX TIME** wird verwendet, um das Schalten einer Aktion zu erzwingen, wenn diese über einen bestimmten Zeitraum hindurch schaltbereit war. Hierdurch wird sichergestellt, dass die Verweildauer in einem Zustand eine maximale Frist nicht überschreitet. Auch hier müssen bei der Spezifikation die Wartezeiten als persistent oder volatil angegeben werden. Bei einer vola-

```
PROCESS RT2
VARIABLES
v : type;
now : Real; ! Uhrenvariable
vta : [ptype ↦ Real]; ! Uhrenvariable

CONST clockresolution ≜ cr; ! Maximale Größe f. Zeitschritt

INIT ≜ vip(v, now) ∧
       now = 0 ∧
       vta = [ p ∈ ptype ↦ 0];
ACTIONS
       Tick( t : Real) ≜ ! Zeitschritt
                         LET ti(p : ptype) ≜ IF (aenab(v,now,p) ∧ p ∈ e_a)
                                             THEN vta[p] + t - now
                                             ELSE 0
                         IN
                         t > now ∧
                         t ≤ now + clockresolution ∧
                         (∀ p ∈ ptype :: ti(p) < mt) ∧
                         vta′ = [p ∈ ptype ↦ ti(p)] ∧
                         now′ = t ∧
                         v′ = v;

       a( p : ptype) ≜
           aenab(v, now, p) ∧ ! Schaltbedingung f. aenab
           v′ = anext(v, now, p) ∧ ! Neuer Wert für Zustandsvariable v
           now′ = now ∧
           vta′ = [vta; Except p ↦ 0]; ! Rücksetzen des Timers

       act( p : ptype) ≜
           actenab(v, now, p) ∧ ! Schaltbedingung f. actenab
           v′ = actnext(v, now, p) ∧   ! Neuer Wert für Zustandsvariable v
           UNCHANGED ⟨now, vta⟩;! now und vta stottern

    SF : Tick; ! starke Fairness für Tick
    SF : a; ! starke Fairness für a
END
```

Abbildung 6.6: Der Prozesstyp `RT2` als Beispiel für volatile, maximale Wartezeiten

tilen, maximalen Wartezeit *mt* wird folgendes Konstrukt (vgl. [HK97a, HK97b]) für die Aktion *a* verwendet:

V MAX TIME a : mt

In Abbildung 6.6 wird die Übersetzung des Konstruktes in flaches cTLA angegeben. Die Aktionen *act* und *a* sind unverändert. Um die Aktion *Tick* übersichtlich zu halten, führen wir einen aktionslokalen Bezeichner *ti(p)* ein, der den Wert enthält, auf den der Zeitgeber *vta[p]* in *Tick* gesetzt wird. Durch die Bedingung $\forall p \in ptype :: ti(p) < mt$ wird sichergestellt, dass nur Zeitschritte auftreten nach denen *vta* einen Wert besitzt, der kleiner als *mt* ist. Somit ist die maximal erlaubte Zeit, die die Aktion *a* schaltbereit ist ohne geschaltet zu haben, immer kleiner als *mt*. Durch die starke Fairness der Aktion *a* ist sichergestellt, dass *a(p)* tatsächlich schaltet, was zum Ausschluss von Zeno-Verhalten beiträgt.

Eine persistente, maximale Wartezeit *mt* für die Aktion *a* wird mit folgendem Konstrukt spezifiziert:

P MAX TIME a : mt

Für dieses Schlüsselwort gilt, dass die Summe aller Zeitperiodendauern, in denen *a* enabled ist, aber zwischenzeitlich nicht feuerte, kleiner *mt* ist.

```
Tick( t : Real) ≙ ! Zeitschritt
                  LET ti(p : ptype) ≙ IF (aenab(v,now,p) ∧ p ∈ e_a)
                                      THEN pta[p] + t - now
                                      ELSE pta[p]
                  IN
                  t > now ∧
                  t ≤ now + clockresolution ∧
                  (∀ p ∈ ptype :: ti(p) < mt) ∧
                  pta' = [p ∈ ptype ↦ ti(p)] ∧
                  now' = t ∧
                  v' = v;
```

Abbildung 6.7: Die Aktion `Tick` für persistente, maximale Wartezeiten

Um die persistente, maximale Wartezeit zu realisieren, muss wiederum nur die Aktion *Tick* modifiziert werden, die in Abbildung 6.7 angegeben ist. Mit dem Konstrukt *P MAX TIME* wird starke Fairness für *a* festgelegt.

6.3.6 Unmittelbar schaltende Aktionen

Mit dem Schlüsselwort **immediate** wird angegeben, dass eine Aktion unverzüglich ohne eine Wartezeit schaltet. Damit ist der Ausdruck **immediate a** mit **P MAX TIME a : 0** gleichbedeutend.

6.3.7 Konsistenzbedingungen an realzeitbehaftete cTLA-Spezifikationen

Bei der Erstellung von Realzeitspezifikationen in cTLA sind Konsistenzbedingungen für wohlgeformte cTLA-Realzeitspezifikationen zu beachten, die sich darauf beziehen, ob widersprüchliche minimale und maximale Wartezeiten für eine einzelne Aktionen *a* spezifiziert worden sind:

- In einer cTLA-Spezifikation dürfen für die Aktion *a* nicht zugleich persistente und volatile Wartezeiten spezifiziert werden.
- Ist für die Aktion *a* ein MIN TIME-Konstrukt mit der Wartezeit t_{min} und ein MAX TIME-Konstrukt mit der Wartezeit t_{max} spezifiziert worden, so muss $t_{min} < t_{max}$ sein.
- Die Aktion *a* ist immer nur für eine endliche Anzahl von Parameterbelegungen *p* schaltbar.

6.4 Vererbung von cTLA-Spezifikationen mit Extends

Durch das Schlüsselwort **Extends** wird ein Konstrukt eingeführt, das Vererbung aus der Objektorientierung auf die Vererbung von Prozesstypen überträgt [RK03]. Das Schlüsselwort wird direkt nach dem Prozessnamen verwendet, um anzugeben, dass ein Prozesstyp von einem oder mehreren anderen Prozesstypen erbt. Beispielsweise sei die folgende Spezifikation eines Prozesstypen gezeigt:

```
PROCESS concreteController EXTENDS
        abstractController
```

Diese gibt an, dass der Prozesstyp *concreteController* vom Prozesstyp *abstractController* erbt. Bei der Vererbung von Prozesstypen erbt ein Unterprozesstyp alle Zustandsvariablen eines Oberprozesstyps. Treten Namenskonflikte durch bereits existierende Zustandsvariablen im Oberprozesstyp auf, werden diese durch eine resultierende Zustandsvariable aufgelöst. Dies ist nur zulässig, wenn die Zustandsvariable des Oberprozesstyps den gleichen Datentyp wie die entsprechende Zustandsvariable des Unterprozesstyps besitzt. Der Unterprozesstyp besitzt eine resultierende Menge von Aktionen, die durch die Vereinigung der Menge der Aktionen mit der Menge des Oberprozesstyps gebildet wird. Das Konstrukt unterstützt das Überladen von Aktionen durch Vereinigung der Mengen der Aktionenparameter und Konjunktion der Vorbedingungen und Effekte einer Aktion, wovon aber in dieser Arbeit kein Gebrauch gemacht wird. Zusätzlich darf ein Prozesstyp interne Aktionen besitzen, die nicht weitervererbt werden. Bzgl. der Sicherheitseigenschaften ist jede Instanz des Unterprozesstyps eine korrekte Instanz des Oberprozesstyps. Die Vererbung von Fairness- und Realzeitkonstrukten wird in [RK03] explizit ausgeschlossen. Diese müssen in dieser Arbeit redefiniert werden, um Konflikte auszuschließen. In dieser Arbeit sollen Fairness und Realzeiteigenschaften stattdessen in eigenen Prozesstypen angegeben werden. Das resultierende Initialisierungsprädikat des Unterprozesstyps wird durch die Konjunktion der Initialisierungsprädikate von Ober- und Unterprozesstyp gebildet.

Kapitel 7

Muster für das konkrete Softwaremodell

In diesem Kapitel werden Entwurfsmuster für das konkrete Softwaremodell vorgestellt. Die hier angegebenen Entwurfsmuster dienen dem Entwurf von Steuerungssoftware und verfeinern die Muster von abstrakten Softwaremodellen. Zunächst werden nacheinander die beteiligten Domänen für den Entwurf von Steuerungssoftware vorgestellt. Anschließend wird jedes einzelne Entwurfsmuster im Kontext seiner Domäne – teilweise unter Berücksichtigung von interessanten strukturellen Aspekten und Verhaltensaspekten – vorgestellt. Ein Teil des Entwurfsmuster-Systems ist in [Dra00] vorgestellt worden.

7.1 Struktur des Entwurfsmuster-Systems

Steuerungen können durch verteilte, objektorientierte Realzeitsoftware-Systeme erstellt werden. Verteilte, objektorientierte Realzeitsysteme benötigen deshalb Entwurfsmuster aus den drei Domänen Realzeit-Verarbeitung, verteilte Verarbeitung sowie Fehlertoleranz. Diese Entwurfsmuster können in Form eines Entwurfsmuster-Systems angegeben werden.

Die Begründung für diese Verwendung von Entwurfsmustern ist in den durch die Software gestellten, nicht-funktionalen Anforderungen zu sehen. Selbstverständlich ist klar, dass Sicherheitsanforderungen, die an Steuerungssoftware für technische Anlagen gestellt werden, in geeignete Realzeitanforderungen umgesetzt werden müssen. Hierfür existieren bereits erprobte Entwurfsmuster. Weiterhin ist Steuerungssoftware häufig verteilt, was sich u. a. an dem in diesem Kontext verwendeten Terminus der DCS (Distributed Control System) [Hal08] für Anlagensteuerungen erkennen lässt. Letztlich wird von Steuerungen der Anlagen auch gefordert, dass sie im Fall des Ausfalls von Funktionen bzw. des Auftretens von Fehlfunktionen auch fehlertolerantes Verhalten zeigen, was bedeutet, dass auftretende Fehler abgefangen werden müssen. Im Folgenden sollen die drei Domänen jeweils einzeln vorgestellt werden, wobei zunächst die Eigenschaften sowie das Spektrum der vorhandenen Entwurfsmuster für jede Domäne aufgezeigt wird. Anschließend werden die einzelnen Entwurfsmuster mit ihren untereinander vorhandenen Beziehungen vorgestellt.

Außerdem ist es hier bemerkenswert, dass es Entwurfsmuster gibt, die mehreren Domänen – in der Regel zwei – angehören. Dies sind speziell Entwurfsmuster, die den Entwurf von verteilter, objektorientierter Realzeit-Software erlauben. Auch dieser Sachverhalt soll berücksichtigt werden.

7.2 Verteilte Verarbeitung mit Entwurfsmustern

In diesem Abschnitt werden Entwurfsmuster zur verteilten Verarbeitung vorgestellt. Zunächst werden Anforderungen und Eigenschaften der verteilten Verarbeitung angegeben, bevor in Tabelle 7.1 eine Übersicht der Entwurfsmuster zur verteilten Verarbeitung angegeben wird.

Ein erweiterter Katalog von Eigenschaften eines verteilten Systems wird in [PSW96] angegeben, der u. a. Entferntheit, Nebenläufigkeit, lokale Zustandsbetrachtung, partiellen Systemausfall, Asynchronität und Autonomie, nennt. Einige wichtige Begriffe sollen hier kurz angeführt werden.

Die Entferntheit gibt an, dass Komponenten eines verteilten Systems räumlich voneinander getrennt sind, was bedeutet, dass Wechselwirkungen entweder entfernt oder lokal auftreten können. Dabei sind Fragen zur Verzögerung und zur Zuverlässigkeit zu berücksichtigen. In der Beispielanlage aus Abschnitt 2.5 sind eine Anlagen- und eine Teilanlagensteuerung räumlich voneinander entfernt.

Die Nebenläufigkeit gibt an, dass eine Komponente parallel zu einer anderen ausgeführt werden kann. Dafür müssen die Komponenten nicht notwendigerweise räumlich voneinander getrennt sein, sondern zur Verarbeitung kann auch ein Mehrkernprozessor an einem Ort verwendet werden. Bei der Beispielanlage aus Abschnitt 2.5 laufen der Regler und die Einzelgerätesteuerungen für den Sensor und den Aktor parallel an verschiedenen Orten ab.

Die Autonomie gibt an, dass Funktionalitäten zum Management und zur Steuerung eines verteilten Systems auf autonome Komponenten (Autoritäten) verteilt sein dürfen, wobei keine Autorität eine übergeordnete Gesamtkontrolle besitzen darf. Inkonsistenzen zwischen den einzelnen Einheiten müssen vermieden werden.

Die Autoren [PSW96] geben außerdem an, dass die offene, verteilte Verarbeitung darauf ausgerichtet ist Systeme zu modellieren und zu entwickeln, die Offenheit, Integration, Modularität, Sicherstellung von Dienstqualität, Sicherheit und Transparenz als Eigenschaften besitzen. Auch hier sollen einige wichtige Eigenschaften kurz beschrieben werden.

Die Offenheit befasst sich einerseits mit der Portierbarkeit von Komponenten auf verschiedene Ausführungsknoten, was bedeutet, dass diese Komponenten auf diesen Knoten ohne Modifikation ausgeführt werden können.

Die Integration befasst sich mit dem Zusammenschluss verschiedener Komponenten und Systeme zu einem Ganzen. Hier wird außerdem die Integration von Teilsystemen mit verschiedenen Architekturen unterstützt.

Die Modularität ist eine Eigenschaft aus dem Bereich des Software-Engineering, die Teilsysteme autonom, jedoch als in Beziehung stehend, modelliert. Jede einzelne Steuerung aus Abschnitt 2.5 repräsentiert ein Modul.

Die Sicherstellung der Dienstqualität bezeichnet die Einhaltung einer Menge von Dienstanforderungen an das Systemverhalten. Diese Eigenschaft umfasst beispielsweise die Realisierung der Timeliness (also Rechtzeitigkeit) von Ereignissen, die Verfügbarkeit und Zuverlässigkeit entfernter Ressourcen und Wechselwirkungen sowie die Bereitstellung einer gewissen Fehlertoleranz beim Ausfall von Teilsystemen, die insbesondere bei sehr großen verteilten Systemen von Bedeutung ist. Bei der Kommunikation von Steuerungen, wie sie in Abschnitt 2.5 behandelt werden, muss eine bestimmte Dienstqualität eingehalten werden.

Sicherheitsanforderungen entstehen insbesondere durch die Entferntheit von Prozessen und Wechselwirkungen sowie durch die Mobilität eines Systems.

Transparenz ist eine zentrale Anforderung, um die Erstellung von verteilten Systemen zu erleichtern. Transparenz umfasst u. a. das Verbergen von Implementierungsdetails, d. h. bei der Integration von neuen Systemen und Ressourcen sollte das ggf. auftretende Abweichen von Systemarchitekturen und -parametern dem Anwender verborgen bleiben und die Verteilungstransparenz, d. h. technische Details sollen bei der Realisierung von Verteiltheit möglichst ebenso verborgen bleiben. Transparenz kann auf viele Funktionalitäten bezogen werden, wie Zugriffstranspararenz, Ortstransparenz und Abarbeitungstransparenz. Für die Anlagensteuerung aus Abschnitt 2.5 ist die Verarbeitung der Gruppen- und Einzelgerätesteuerungen transparent.

Nun soll eine kurze Übersicht der Entwurfsmuster aus Tabelle 7.1 gegeben werden, bevor diese jeweils in einem eigenen Abschnitt vorgestellt werden.

Die Entferntheit in einem verteilten System wird durch das Proxy- und das Broker-Muster unterstützt. Das Proxy-Muster erlaubt es, einen lokalen Stellvertreter für ein Objekt in einem entfernten Adressraum festzulegen, welcher mit dem entfernten Objekt kommuniziert. Das Broker-Muster ermöglicht die Weiterleitung von Objektadressen zur Laufzeit, auch wenn die beteiligten Objekte an einem Kommunikationsvorgang vorher nicht bekannt waren. Außerdem unterstüt-

Entwurfsmustername	Entwurfsmusterbeschreibung	Beziehung zu anderen Domänen
Proxy	Bildet Stellvertreter eines Objektes in fremden Adressräumen.	Keine
Broker	Dient als Vermittler von Objektadressen zur Laufzeit.	Realzeit-Broker [CS02]
Rendezvous	Dient der Synchronization von Aufrufen eines Objektes.	Keine
Distributed Observer	Beobachtet Ereignisquellen.	Fehlertolerante Verarbeitung
Verteilte Transaktion	Verwendet Transaktionsmechanismen, wie z. B. das Zwei-Phasen-Commit.	Keine
Asynchronous Command	Unterstützt asynchron und entfernt abgesetzte Kommandos.	Keine

Tabelle 7.1: Die Entwurfsmuster zur verteilten Verarbeitung

zen Broker die föderative Namensverwaltung und die Evolution in einem verteilten System. Das Distributed Observer-Muster erlaubt die Beobachtung von unterschiedlichen und auf mehrere Knoten verteilte Signalquellen. Somit werden die Entferntheit und die Verteilungstransparenz unterstützt. Bei der verteilten Verarbeitung müssen aufgrund der Nebenläufigkeit auch Synchronisationsaspekte berücksichtigt werden, was mit dem Rendezvous-Muster – auch Semaphormuster genannt – geschieht. Die Autonomie wird durch Transaktionen gewährleistet, welche ACID-Eigenschaften auch für verteilte Systeme sicherstellen und damit auch die Transparenz bewahren. ACID ist eine Abkürzung, die die Atomarität, Konsistenz, Isolation und Persistenz für Transaktionen umfasst. Die Asynchronität von Befehlen für QoS-Angaben wird durch das Asynchronous-Command-Muster unterstützt.

7.2.1 Proxy-Muster mit Schwerpunkt auf dem Remote-Proxy

Die grundsätzliche Idee des Proxy-Musters besteht darin, einem tatsächlichen Objekt ein Stellvertreter-Objekt vorzulagern. Das Stellvertreter-Objekt nimmt die Aufrufe von Methoden entgegen und führt je nach Verwendungszweck des Proxys eine eigenständige Verarbeitung durch. Anschließend wird der Methodenaufruf an das eigentliche Objekt weitergeleitet, damit dies die erforderliche Verarbeitung durchführen kann. Für das Proxy-Muster existieren zahlreiche Varianten, die auch kombiniert – z. B. in modernen Application Servern wie WebSphere von IBM – eingesetzt werden. Eine wichtige Variante unterstützt das Caching, um Daten vorzuhalten, die bereits vom tatsächlichen Objekt bereitgestellt worden sind. Außerdem wird häufig die Protokollierung von Aufrufen vorgenommen. Ist die Sicherheit des tatsächlichen Objektes von Bedeutung, wird das Proxy-Objekt dazu verwendet, das eigentliche Objekt vor unberechtigten Zugriffen zu schützen. Weiterhin werden Proxy-Objekte zur Synchronisation von Zugriffen auf das tatsächliche Objekt verwendet, um Synchronisationsprobleme zu vermeiden. In dieser Arbeit ist speziell das so genannte Remote-Proxy von Interesse, das die Kommunikation zwischen Objekten über die Grenzen von Adressräumen hinweg unterstützt.

Kontext: Bei der Kommunikation von Objekten aus unterschiedlichen Adressräumen treten vielfältige Problemstellungen auf.

Problem: Ein direkter Zugriff ist zwar häufig möglich, neigt aber bei der Implementierung zu Leistungs-, Sicherheits-, Transparenz- und Flexibiltätsproblemen, weil die Lösung zu kompliziert ist. Von einer Lösung erwartet man Folgendes:

- Ein Klient kann in einer vertretbaren Laufzeit und sicher auf einen entfernten Server zugreifen. Weiterhin soll der Klient Kenntnisse über die Zugriffskosten und die Transportzeiten beim Zugriff auf einen Server besitzen.
- Der Klient soll transparent und möglichst einfach auf den Dienst eines entfernten Servers

zugreifen können.

Lösung: Das Remote-Proxy ist ein bewährtes Kernkonzept des verteilten Aufrufes von Methoden und wird beim Remote Procedure Call (RPC) oder bei der Remote Method Invocation (RMI) in Java verwendet. Das Proxy-Muster ist anwendbar, wenn eine normale Referenz oder Zeiger nicht angebracht oder unmöglich sind, wenn etwa ein Objekt in einem anderen Thread oder Prozessor untergebracht ist. Dieses Entwurfsmuster entkoppelt einen Klienten von seinen Servern, indem ein lokales Proxy als Stellvertreter für den entfernt zugreifbaren Server verwendet wird. Dadurch entsteht die geforderte Transparenz eines Aufrufes von einem Klient an einen Server. Die Entkopplung von Klient und Server trägt dazu bei, dass Klient und Server in unterschiedlichen Adressräumen untergebracht sein können. Im Proxy wird, wenn nötig geprüft, ob der Klient die Berechtigung für einen Zugriff besitzt. In Abbildung 7.1 wird das zugehörige Sequenzdiagramm gezeigt. Es enthält ein Objekt in der Rolle eines Klienten und ein Objekt in der Rolle eines Servers. Alle Aufrufe eines Dienstes durch den Klienten werden vom lokalen Proxy an den Server weitergeleitet. Das Proxy kann dann Aufrufe an den ursprünglichen Server weiterleiten oder es kann einer asynchronen Strategie folgen, die periodisch oder in Episoden aktualisierte Werte vom Server abruft. In Abbildung 7.1 wird gezeigt wie der aufgerufene Dienst durch Weiterleitung der Service-Nachricht weitergegeben wird.

Beispiel: In der Beispielanlage aus Abbildung 2.4 werden Remote-Proxys zur Kommunikation für alle nicht an einem Ort befindlichen Steuerungen verwendet. Speziell werden zwei Remote-Proxys zur Kommunikation des Reglers mit seiner Aktor- und seiner Sensorsteuerung verwendet. In der OPC UA wird das Proxy-Muster verwendet, um Objektknoten zu verbinden [MLD09].

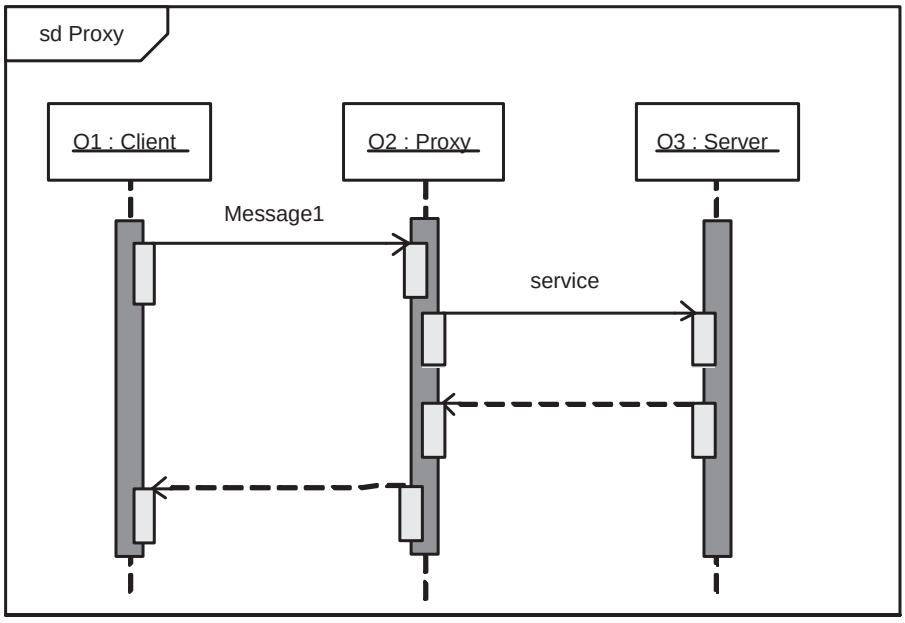


Abbildung 7.1: Ein typisches Sequenzdiagramm des Proxys

Der primäre Nachteil des Proxy-Musters besteht darin, dass ein potentieller Leistungsverlust auftreten kann. Eine andere Schwäche besteht in der Kopplung zwischen den Proxy-Objekten und dem Server. Das Proxy muss wissen, wie man den Server findet, daher muss es nicht nur die operationelle Syntax für den Datenaufruf, sondern auch den Platz des Servers kennen sowie in der Lage sein, Aufrufe über ein zugrunde liegendes Kommunikationsprotokoll weiterzuleiten. Das Proxy-Muster kann durch Verwendung des Broker-Musters ausgearbeitet werden, um eine stärkere Entkopplung von möglichen Servern zu erzielen.

7.2.2 Broker-Muster

In diesem Abschnitt wird das Broker-Muster vorgestellt.

Kontext: Ein verteiltes und unter Umständen heterogenes System, in dem voneinander unabhängige Komponenten zusammenarbeiten, ist die Ausgangssituation.

Problem: Ein verteiltes Software-System besteht aus einer Menge von zusammenwirkenden, voneinander entkoppelten Komponenten, die sich an unterschiedlichen Orten befinden können. Ein derartiges System kann über eine hohe Komplexität verfügen, trotzdem verfügt es über eine hohe Flexibilität, Wart- und Änderbarkeit. Da die verteilten Komponenten miteinander kommunizieren müssen, ist ein grundlegender Kommunikationsmechanismus notwendig (z. B. RMI). Weiterhin müssen die Klienten wissen, wo sich die Server befinden. Dienste für das Hinzufügen, Auswechseln, Aktivieren und Suchen von Komponenten sind erforderlich. Anwendungen, die auf diese Dienste zugreifen, sollten nicht von systemspezifischen Details abhängen.

Lösung: Das Broker-Muster ist ein um weitere Funktionalitäten ausgearbeitetes Proxy-Muster, welches die Entkopplung der Klienten von den Servern unternimmt. Ein Objekt-Broker hat die Aufgabe des Wissensvermittlers über den Aufenthaltsort anderer Objekte. Der Objekt-Broker kann dieses Wissen a priori (zur Übersetzungszeit) besitzen oder es dynamisch zur Laufzeit erwerben, wenn sich Objekte registrieren, oder aber es kann eine Kombination aus Compile- und Laufzeitinformationen vorliegen. Der Hauptvorteil des Broker-Musters besteht darin, dass es möglich ist ein Proxy-Muster zu verwenden, wobei der Aufenthaltsort zur Übersetzungszeit nicht bekannt ist. Dies macht es besonders für Mehrprozessor-Systeme interessant.

Die architektonischen Komponenten aus denen das Broker-Muster zusammengesetzt ist, sind der Objekt-Broker, der Klient und die Server-Subsysteme. Es passiert selten, dass ein Broker einen isolierten Klienten und Server hat. Wenn das Broker-Muster benutzt wird, ist es als eine Entwurfsentscheidung von großer Bedeutung und sehr nützlich.

In fehlertoleranten und sicherheitskritischen Systemen kann der Broker eine große Bedeutung zur Bewahrung der Systemintegrität spielen. So kann der Broker z. B. einen Watchdog enthalten, der Verbindungen mit allen sicherheitsrelevanten Subsystemen aufrecht erhält. Wenn ein Subsystem instabil wird oder abstürzt, kann der Objekt-Broker Sicherheitsaktionen initiieren, wie eine Anlage herunterfahren oder das Hereinfahren von Steuerstäben in den Kern eines Reaktors vornehmen. Derzeit befinden sich Realzeit-Broker für Java [CS02, Sch02, KSKC03], die auf dem Real-Time Corba Standard der OMG beruhen, in der Entwicklung.

7.2.3 Recoverable Distributed Observer Muster

Dieses Entwurtsmuster ist eine Variante des Observer Musters für Software mit Verteilungs- und Fehlertoleranzanforderungen und wird in [ID96] vorgestellt. Das Entwurfsmuster ist bei IBM für den Entwurf des Octopus Schedulers verwendet worden. Weiterhin ist es beim Entwurf des Distributed Lock Mangers für das Calysto File System zur Anwendung gekommen.

Kontext: In einem verteilten System kooperieren Anwendungen, die auf unterschiedlichen Knoten bzw. auf verschiedenen Prozessoren ausgeführt werden. Die Konsistenz der Daten ist von besonderem Interesse. Hierbei sind Fehlertoleranz- und Performanceanforderungen zu berücksichtigen.

Problem: Jede auf einem Knoten ausgeführte Anwendung besitzt einen Teilzustand. Der Gesamtzustand wird durch die einzelnen Teilzustände auf den Knoten gebildet. Dieser muss permanent neu ermittelt und an die einzelnen Knoten propagiert werden. Dabei muss berücksichtigt werden, dass Knoten fehlerbedingt ausfallen können. Außerdem muss berücksichtigt werden, mit welcher Strategie die einzelnen Anwendungen über Zustandsänderungen benachrichtigt werden.

Lösung: Das Entwurfsmuster bietet Entwurfsalternativen bzgl. der Gewährleistung der Datenkonsistenz und der Fehlertoleranz. Auf jedem Knoten gibt es einen lokalen Zustandsmanager (LSM), der den lokalen Zustand verwaltet. Auf einem ausgezeichnetem Knoten befindet sich der globale Zustandsmanager (GSM). Die lokalen und der globale Zustandsmanager kommunizieren miteinander, um sich gegenseitig über Zustandsänderungen zu informieren. Weiterhin gibt es auf jedem Knoten eine Komponente, die der Fehlerbehandlung dient. Diese wird als Local Failure Handler (LFH) bezeichnet. Zusätzlich gibt es eine zentrale Komponente zur Fehlerbehandlung, die Global Failure Handler (GFH) genannt wird. Der GFH prüft in regelmäßigen Abständen mittels

der *checkMembers*-Nachricht, ob die LFH auf den einzelnen Knoten noch arbeiten. Darauf antwortet jeder funktionstüchtige LFH mit der *isAlive*-Nachricht an den GFH. Erfolgt diese Antwort nicht nimmt der GFH an, dass der entsprechende LSM bzw. GSM ausgefallen ist. Daraufhin wird eine Rekonfiguration des verteilten Programms eingeleitet, bei dem ein neuer globaler Zustand ermittelt wird. Für den Fall, dass der GFH selbst ausfällt, muss ein neuer GFH auf einem noch funktionsbereitem Knoten erzeugt werden. Hierbei wählen die LFH einen GFH aus.

Bei dem Entwurfsmuster existieren Variationsmöglichkeiten bzgl. starker und schwacher Konsistenz. Der Unterschied zwischen starker und schwacher Konsistenz wirkt sich darauf aus, wie bei einer Zustandsänderung, die durch eine Anwendung ausgelöst wird, die Datenkonsistenz sichergestellt wird. Im Falle von schwacher Konsistenz wird zuerst ein LSM aktualisiert und die Zustandsänderung wird erst anschließend an den GSM propagiert. Bei starker Konsistenz wird jede Zustandsänderung zunächst an den GSM weitergeleitet und dieser benachrichtigt anschließend jeden LSM über die Zustandsänderung.

Im Sequenzdiagramm aus Abbildung 7.2 wird speziell die Interaktion der Komponenten zur Sicherstellung der Datenkonsistenz bei schwacher Konsistenz betrachtet. Tritt durch eine Anwendung eine Veränderung auf, wird der LSM des zugehörigen Knotens informiert. Der LSM informiert den GSM über die Änderung des Datenzustandes durch die *notifyGSM*-Nachricht. Auf dem GSM wird anschließend die *update*-Nachricht aufgerufen, die den globalen Zustand aktualisiert. Anschließend werden die LSM über die vorangegangenen Zustandsänderungen bzw. den neuen globalen Zustand informiert. Dafür wird auf dem GSM die Operation *notifyLSM* aufgerufen, die anschließend die *set*-Nachrichten an die lokalen LSM versendet.

Die Leistungsaspekte des Musters werden durch das Caching des globalen Zustandes im LSM und die Frequenz, mit der die Operation *notifyGSM* vom LSM aufgerufen wird, vorgegeben. Für strenge Konsistenz wird die Operation *notifyGSM* synchron für Zustandsmodifikationen jedes einzelnen LSM aufgerufen. Bei schwacher Konsistenz wird die Operation *notifyGSM* erst mit einiger Verzögerung aufgerufen.

Das Recoverable Distributed Observer Muster hat die folgenden Vorteile:

- Es nimmt Interaktionen zwischen den Komponenten eines verteilten Systems, die einen gemeinsamen Zustand besitzen, unter Berücksichtigung der Performance vor.
- Es verbessert die Wiederverwendbarkeit und Modularität von verteilten, fehlertoleranten Komponenten, die einen gemeinsamen Zustand besitzen.

Beispiel: Bei der Beispielanlage aus Abbildung 2.4 kann das Recoverable Distributed Observer Pattern zur Kommunikation der Gruppensteuerung zum Abpumpen mit der zugehörigen Verriegelungssteuerung verwendet werden.

7.2.4 Transaktionsmuster

Kontext: In verteilten Systemen kann es vorkommen, dass das Ergebnis einer Verarbeitung mehr als einen Knoten betrifft, bzw. dass durch die Verarbeitung Daten auf mehr als einem Knoten betroffen sind.

Problem: Häufig ist es so, dass Anforderungen an die Konsistenz der Verarbeitung gestellt werden. So muss etwa bei einer verteilten Verarbeitung auf jedem Knoten ein bestimmter Schritt ausgeführt werden, um die Konsistenz der gesamten Verarbeitung zu gewährleisten. Sollte auf einem Knoten der entsprechende Schritt nicht durchgeführt werden können, darf die gesamte Verarbeitung nicht stattfinden. Auch die Nebenläufigkeit zu anderen stattfindenden Verarbeitungen muss berücksichtigt werden. Weiterhin muss die Dauerhaftigkeit der durch die Verarbeitung ausgelösten Änderungen sichergestellt werden.

Lösung: Hierzu wird zunächst der Begriff der Transaktion behandelt. Eine Transaktion ist eine Sequenz von Operationen, die den Zustand eines Objektes oder einer Sammlung von Objekten auf eine wohl definierte Weise ändert. Transaktionen sind deshalb nützlich, weil sie Aussagen darüber machen, wie der Zustand eines Objektes vor, während und nach einer Verarbeitung durch ein Objekt auszusehen hat. Manchmal gibt es auch Aussagen, die nicht in Bezug zu den Objekten einer

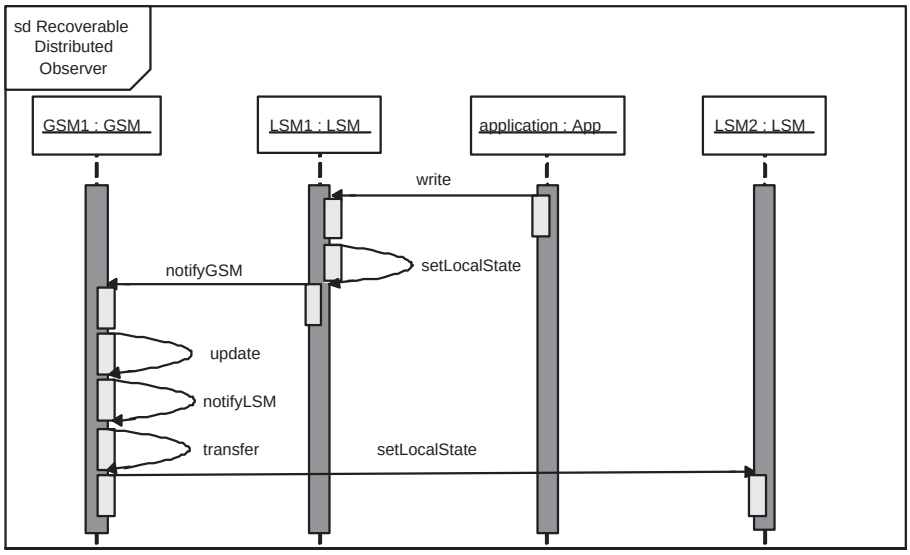


Abbildung 7.2: Das Sequenzdiagramm des Recoverable Distributor Observer Musters

Transaktion stehen. Entwurfsmuster für Transaktionen treten in unterschiedlichen Erscheinungsformen auf. Diese sind nach [Gra99a, MK04] ACID, Composite, Zwei-Phasen-Commit und Audit Trail. Eine ACID-Transaktion – also eine Transaktion mit ACID-Eigenschaften – stellt sicher, dass eine Transaktion niemals eine unerwartete oder inkonsistente Ausgabe erzeugt. Grundlegend ist der Gedanke, dass eine Transaktion die sog. ACID-Eigenschaften erhalten muss. Diese sind Atomarität, Konsistenz, Isolation und Persistenz (Durability), die hier kurz vorgestellt werden:

- Atomarität stellt sicher, dass die Zustandsänderungen, welche an einen Objekt erfolgen, durch eine Transaktion atomar sind, was bedeutet, dass sie entweder genau einmal oder gar nicht geschehen.
- Konsistenz gewährleistet, dass eine Transaktion eine korrekte Transformation des Objektzustandes ist. Bei Beginn und nach einer Transaktion erfüllen die Objekte, auf denen eine Transaktion gearbeitet hat, sämtliche Integritätsbedingungen.
- Isolation stellt sicher, dass die Ausführungsreihenfolge dieser anderen Transaktionen völlig egal ist, obwohl eine Transaktion nebenläufig zu anderen Transaktionen ausgeführt werden kann.
- Persistenz legt die Zustandsänderungen einer Transaktion, die erfolgreich beendet wird, dauerhaft fest.

In komplexen Anwendungen ist es oftmals erforderlich, Transaktionen aus einfacheren ACID-Transaktionen zusammenzusetzen, die als eigenständige Komponenten dienen und für die vollständige Transaktion bereitstehen. Um die Atomarität zu gewährleisten, ist hier ein verteiltes Commit-Protokoll erforderlich. Ein sehr bekanntes Verfahren ist das so genannte Zwei-Phasen-Commit (2PC) [LKK93, CDK05]. Bei diesem Verfahren wird in der ersten Phase des Protokolls, die auch als Abstimmungsphase bezeichnet wird, durch einen Koordinator die Zustimmung oder Ablehnung zur Festschreibung der Datenänderungen sämtlicher beteiligter Prozesse eingeholt. Der Koordinator ist in der Regel der Prozess, der eine Festschreibung einleitet. Damit der Koordinator *Commit* entscheiden darf, muss die Voraussetzung, dass alle Teilnehmer zustimmen, erfüllt sein. Sollte dies nicht der Fall sein, d. h. mindestens ein Teilnehmer stimmt nicht zu, muss der Koordinator auf *Abort* (Zurücksetzen) entscheiden. Auf der Grundlage dieses gemeinsamen Ergebnisses werden entweder alle Teiltransaktionen zum erfolgreichen Abschluss geführt, bei dem die zwischenzeitlich gesperrten Ressourcen auch wieder freigegeben werden, oder die gesamte Transaktion wird zurückgesetzt.

Im Folgenden soll das Transaktionsmuster für das Zwei-Phasen-Commit durch zwei Sequenzdiagramme, die in den Abbildungen 7.3 und 7.4 gezeigt sind, näher beschrieben werden. Das Sequenzdiagramm des Entwurfsmusters besteht aus den anschließend beschriebenen Objekten:

- Die übergeordnete Transaktionslogik, bei der eine Klasse auf der obersten Ebene die gesamte Logik einer Transaktion übernimmt. Diese wird durch *O1* repräsentiert.
- Der Transaktionskoordinator nimmt als Objekt – hier *O2* – die Koordination der einzelnen Teiltransaktionen einer Gesamttransaktion vor. Dieser ermittelt, ob alle Teiltransaktionen korrekt abgeschlossen worden sind, wodurch ein Commit ausgelöst wird, oder ob infolge einer Ablehnung einer Teiltransaktion ein Abort auftritt.
- Die Teiltransaktionen werden selbständig auf jedem Knoten ausgeführt. Jede Instanz dieser Klasse ist verantwortlich für die Ausführung einer lokalen Teiltransaktion auf einem Knoten eines verteilten Systems. Häufig ist es jedoch so, dass die Teiltransaktionen nicht so entworfen wurden, dass sie mit dem Zwei-Phasen-Commit arbeiten können, sodass Wrapper eingesetzt werden müssen. Der Wrapper findet Anwendung, wenn eine existierende Klasse verwendet werden soll, deren Schnittstelle nicht der benötigten Schnittstelle entspricht. Wrapper sind Hüllklassen und kapseln ein bestimmtes Verhalten ein. Die Transaktionswrapper ermöglichen es, dass Transaktionsobjekte mit dem Zwei-Phasen-Commit-Protokoll arbeiten, indem sie die Logik, die dieses Protokoll erfordert, bereitstellen. In dem Sequenzdiagramm sind *O3* und *O5* Wrapperobjekte für die einzelnen Teiltransaktionen, die jeweils eine lokale Kapsel für eine Teiltransaktion bilden.

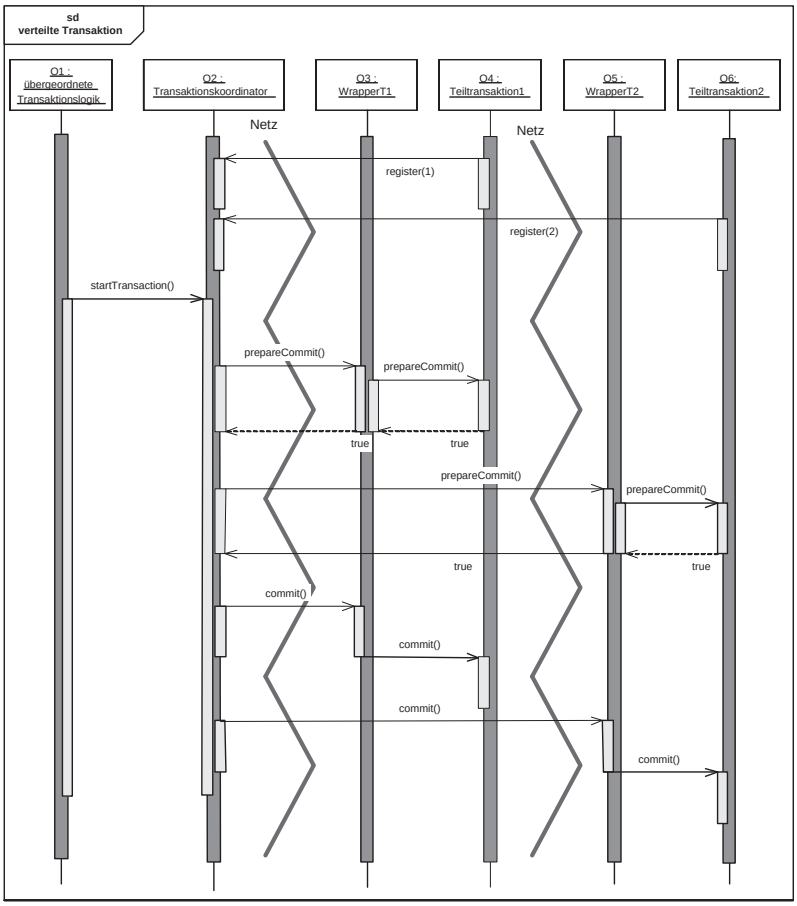


Abbildung 7.3: Ein beispielhaftes Sequenzdiagramm des Transaktionsmusters (Commit)

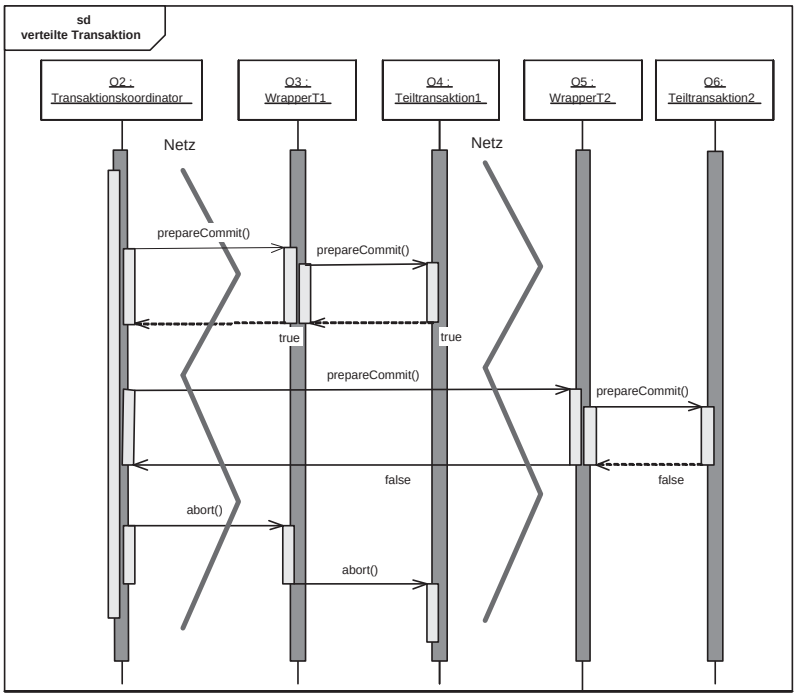


Abbildung 7.4: Ein beispielhaftes Sequenzdiagramm des Transaktionsmusters (Abort)

- Die Objekte für die beiden lokalen Teiltransaktionen auf den Knoten werden hier mit *O4* und *O6* bezeichnet. Die Teiltransaktionen können auch als Komponententransaktionen angesehen werden.

Die Teiltransaktionen registrieren sich beim Koordinatorobjekt mit asynchronen *register*-Aufrufen aus unterschiedlichen Threads. Die Objekte, die die *register*-Methode des Koordinatorobjektes aufrufen, laufen unabhängig weiter.

Durch die übergeordnete Transaktionslogik wird ein *startTransaction*-Aufruf an das Koordinatorobjekt abgesetzt, um eine Transaktion zu beginnen. Das Koordinatorobjekt setzt seinerseits *prepareCommit*-Aufrufe an die Transaktionswrapper der Teiltransaktionen ab. Die Wrapper leiten die *prepareCommit*-Aufrufe an die Teiltransktionen weiter. Die Aufrufe an die Teiltransaktionen erfolgen – wenn möglich – nebenläufig, sonst erfolgen die Aufrufe sequentiell. Jede Teiltransaktion antwortet mit einem Return, ob sie bereit ist an der Transaktion teilzunehmen oder nicht. Stimmt eine Teiltransaktion dem *prepareCommit* zu, muss sie sich für einen *commit*-Aufruf bereithalten. Für den Fall, dass alle Teiltransaktionen an der gesamten Transaktion teilnehmen (s. Abbildung 7.3), wird durch das Koordinatorobjekt ein *commit*-Aufruf an alle Transaktionswrapper abgesetzt, welcher an die jeweiligen Teiltransaktionen weitergeleitet wird.

Falls eine Teiltransaktion nicht bereit ist an der Transaktion teilzunehmen, wird – wie in Abbildung 7.4 für die erste Teiltransaktion gezeigt – wird ein Abbruch der gesamten Transaktion eingeleitet. Dafür sendet der Transaktionskoordinator jeweils eine *abort*-Nachricht an die Objekte der Transaktionswrapper, die diese an die Teiltransaktionen weiterleiten. Die Teiltransaktionen nehmen dann ihren lokalen Abbruch vor.

Das Transaktionsmuster muss eine Kompensation des Ausfalls eines Koordinatorobjektes unterstützen. Dafür wird der Status jeder in Bearbeitung befindlichen Transaktion in einer Datei abgelegt. Dieser wird beim erneuten Start des Koordinatorobjektes zur Wiederherstellung des vorherigen Zustandes verwendet.

Beispiel: Um in der Beispielanlage aus Abbildung 2.4 die nicht-funktionalen Anforderungen zur transaktionsgesicherten Kommunikation zwischen dem Bild und der Teilanlagen- bzw. der Anlagensteuerung zu realisieren, wird dieses Entwurfsmuster verwendet. Die Verwendung von verteilten

Transaktionen mit einem Zwei-Phasen-Commit-Protokoll für die Steuerung von Feldgeräten wird in [SSS09] vorgestellt.

7.2.5 Rendezvous-Muster

Die Synchronisation ist ein zentrales Problem bei der Koordination sowie von nebenläufigen als auch verteilten Komponenten [CDK05]. Es gibt zahlreiche bekannte Ansätze zur Behandlung der Synchronisation aus dem Bereich der Kommunikation von sowie für Threads Sperren zu organisieren [FB04], Semaphore, Monitore, Mutexe etc. Sperren blockieren Threads, da diese in einen Wartezustand versetzt werden. Algorithmen, die ohne Blockierungen arbeiten, werden als lock-free bezeichnet und sind in Mehrprozessorsystemen von besonderer Bedeutung [HS08]. In diesem Abschnitt wird wegen seiner allgemeinen Anwendbarkeit das Rendezvous-Muster vorgestellt [Dou99, Dou04]. Ein Rendezvous-Muster kann sowohl lokal, d. h. im selben Adressraum – etwa bei einem Mehrprozessorsystem – als auch verteilt angewendet werden [Arj07].

Kontext: In verteilten Systemen arbeiten zahlreiche Komponenten in unterschiedlichen Kontrollflüssen nebenläufig miteinander zusammen und greifen auf gemeinsame Daten zu.

Problem: Oft müssen Ergebnisse zwischen Komponenten synchronisiert werden, bevor eine weitere Bearbeitung erfolgen kann, weil Verarbeitungsschritte voneinander abhängen oder zum richtigen Zeitpunkt auf Daten zugriffen werden muss. Es existieren verschiedene Strategien, das Zusammenwirken zu koordinieren. Eine Komponente kann beliebig lange auf eine andere warten. Die Wartezeit kann aber auch begrenzt sein, sodass nach Ablauf einer Frist, die Verarbeitung fortgesetzt wird. Weiterhin ist es möglich, dass die Komponente ihre Verarbeitung sofort fortsetzt, falls eine andere nicht augenblicklich zur Verfügung steht.

Lösung: Im Bereich der Betriebssysteme sind die klassischen Lösungen hierfür Semaphore und Rendezvous-Objekte. Synchronisationsmaßnahmen können sowohl im ASM als auch im KSM als Entwurfsmuster angewendet werden. Rendezvous-Objekte finden, z. B. bei der Synchronization von Realzeit-Tasks Verwendung.

Ein Rendezvous bezieht sich auf die Synchronisation von nebenläufigen Tasks. Die üblichere Benutzung des Begriffs Rendezvous bezieht sich auf die Synchronization von mehreren Tasks. Das Rendezvous von zwei Tasks wird bilaterales Rendezvous genannt. Formal gesehen ist ein Rendezvous ein Mittel, um die Vorbedingungen der partizipierenden Tasks zu erzwingen. Dieses Entwurfsmuster besteht aus einem koordinierenden, passiven Objekt (Rendezvous-Objekt genannt), Klienten, die sich synchronisieren müssen, und beidseitigen Sperrsemaphoren. Sperrsemaphore veranlassen die Threads zum Warten, bis sämtliche ihrer Vorbedingungen erfüllt sind. Dies wird durch das Rendezvous-Objekt in Zusammenarbeit mit einem Policy-Objekt geprüft. Durch ein Policy-Objekt wird festgelegt, welche Strategie für das Rendezvous verwendet wird. Eine bekannte Strategie ist das zeitbehaftete Rendezvous, bei dem ein Thread nach Ablauf einer Frist selbstständig weiterläuft. Bei der Thread Barrier Strategie [OW04] werden die beteiligten Threads solange blockiert, bis eine bestimmte Anzahl von ihnen registriert ist, danach wird die Blockade aufgehoben. Die Cyclic Barrier Strategie ist eine Variante des Thread Barriers, bei der die Barriere (hier das Rendezvous-Objekt) beliebig oft wiederverwendet werden kann. Durch Bedingungsvariablen ist es möglich Datenflüsse zu synchronisieren, indem geprüft wird, ob eine definierte Bedingung eingetreten ist.

Das Entwurfsmuster besteht aus den folgenden Objekten:

- Ein Thread-Objekt des Entwurfsmusters ist ein normales Objekt, das in einem asynchronen Thread operiert. Es ruft die *wait()*-Methode des Rendezvous-Objekts auf und bleibt solange gesperrt, bis das zugehörige Sperrobjekt (ein Objekt der Klasse Lock), eine Antwort auf den *wait()*-Aufruf die Kontrolle zurück an das Thread-Objekt gibt, damit dieses seine Verarbeitung fortsetzen kann.

- Das Rendezvous-Objekt koordiniert den Synchronisationsprozess und verschmilzt die beidseitigen Semaphore zu Klienten-Threadobjekten. Jedes Rendezvous-Objekt handhabt typischerweise einen einzelnen Synchronisationspunkt. Jeder an dem Rendezvous teilnehmende Thread hat eine Assoziation mit der Rendezvous-Klasse, sodass deren *wait()*-Operation aufgerufen werden kann. Das Rendezvous-Objekt behandelt die Koordination der Sperrobjekte

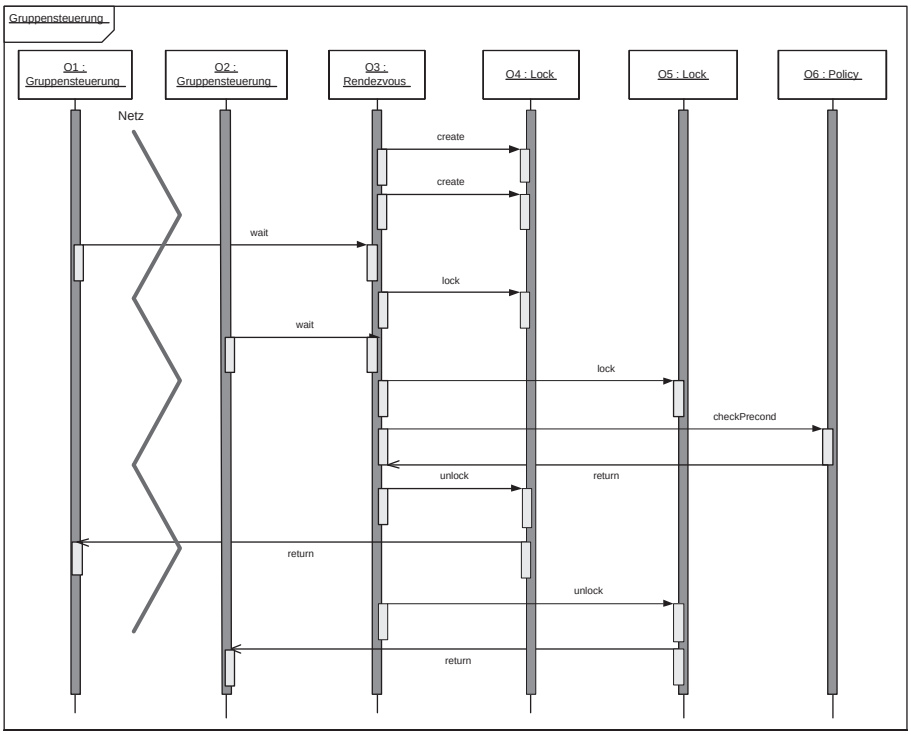


Abbildung 7.5: Ein typisches Sequenzdiagramm des Rendezvous-Musters

und stellt mit Hilfe eines Policy-Objektes, das die ausgewählte Sperrstrategie enthält, fest, ob alle Sperren im Sperrzustand sind. Wenn dies eintritt, wird allen Sperrobjekten signalisiert, dass sie die von ihnen blockierten Threads freigeben und in den *nicht gesperrt* Zustand übergehen.

- Ein Sperrobjekt hat zwei Zustände: *gesperrt* und *nicht gesperrt* (der initiale Zustand). Wenn ein Thread Objekt die *wait* Operation auf dem Rendezvous aufruft, wird die ObjektID verwendet, um das passende Sperrobjekt auszuwählen und seinen Zustand zu ändern. Jeder Klient hat sein eigenes Sperrobjekt, das erzeugt wird, wenn das Rendezvous initial angelegt wird. Sperren können auf zahlreiche Weisen implementiert werden. Die gebräuchlichsten sind das *busy-wait* und das *sleep-wait*. Beim *busy-wait* Ansatz befindet sich die *lock*-Operation in einer Schleife bis die Synchronisationsbedingung (alle am Rendezvous beteiligten Threads warten) erfüllt ist. Die Schlafe-Warten Lösung setzt den Klient Thread schlafend in Abhängigkeit vom Betriebssystem Ereignis. Dieses Betriebssystem Ereignis wird durch das Betriebssystem angefragt, wenn die Vorbedingungen erfüllt sind.

In Abbildung 7.5 ein Sequenzdiagramm angegeben, das beispielhaft die Anwendung des Rendezvous-Musters zeigt. Das Objekt *O1* ist eine Gruppensteuerung zum Befüllen eines Behälters, die in einem eigenen Thread abläuft. Durch das Objekt *O2* wird eine Gruppensteuerung zum Abpumpen eines Behälters modelliert. Zwischen diesen beiden Objekten ist eine Synchronisation erforderlich. Hierfür wird das Rendezvous Objekt *O3* verwendet, das mit dem Policy-Objekt *O6* zusammenarbeitet, um festzustellen, ob die Voraussetzungen für ein Rendezvous erfüllt sind. Die Objekte *O1* und *O2* rufen jeweils die *wait*-Operation auf dem Objekt *O3* auf. Die Sperrobjekte *O4* und *O5*, welche von *O3* erzeugt werden, unterstützen das Rendezvous-Objekt, um die Sperrung von *O1* und *O2* aufrechtzuerhalten, die durch zwei *lock()*-Aufrufe von *O3* erzeugt wurde. Sind die Voraussetzungen für das Rendezvous erfüllt, ruft *O3* auf jedem Sperrobjekt *unlock* auf und die Sperrobjekte heben durch Antworten auf den *wait*-Aufruf die Blockierung der Objekte aus den Threads auf.

Begriff	Definition
Task-Execution-Time	Die Task-Execution-Time ist die CPU-Zeit, die benötigt wird, um ein Modul oder eine Task auszuführen. Die Ausführungszeit für den schlechtesten Fall wird als Worst Case Execution Time (WCET) bezeichnet [Mar07].
Taskperiode	Die Taskperiode ist das Zeitintervall zwischen zwei Taskaufrufen. Wenn man sagt, dass die Task t eine Taskperiode von 10 ms hat, bedeutet dies, dass die Task alle 10ms initiiert und abgeschlossen sein muss.
Task-Deadline	Die Task besitzt einen Beendigungszeitpunkt D. Ein derartiger Beendigungszeitpunkt D heißt Task-Deadline. Der voreingestellte Wert für eine Task-Deadline ist das Ende einer Taskperiode.
Slack-Time oder Laxity	Für den Fall, dass die Task-Execution-Time vor der Task-Deadline abgeschlossen ist, verbleibt eine Zeitdifferenz zwischen dem Endpunkt der Task-Execution-Time und der Task-Deadline. Diese wird als Slack-Time oder Laxity [Mar07] bezeichnet.
Taskrate	Die Taskrate ist der reziproke Wert der Taskperiode. In einem festem Zeitintervall t ist die Ausführungsrate einer Task t/T, wobei T die Taskperiode ist.
Jitter	Der Jitter beschreibt das Zeitintervall, in dem eine Task zu jedem Zeitpunkt gestartet werden kann, aber vor dem Ende der Deadline beendet sein muss. Damit ist die Task-Deadline Taskausführungszeit plus erlaubtem Jitter.
Transport-Lag	Der Transport-Lag ist ein Begriff aus der Systemtheorie bzw. Regelungstechnik und bezeichnet die verstrichene Zeit zwischen der aktuellen Messung eines Wertes durch einen Sensor und dem Empfang eines Wertes am Eingang eines Aktors. Der Transport-Lag ist für die Stabilität eines Reglers von Bedeutung.

Tabelle 7.2: Die Begriffe aus dem Bereich der Realzeit-Verarbeitung

7.3 Realzeit-Verarbeitung mit Entwurfsmustern

In diesem Abschnitt werden zunächst grundlegende Begriffe der Realzeitverarbeitung eingeführt. Weiterhin dient er der Vorstellung von Entwurfsmustern, die die Erstellung von Realzeitsoftware ermöglichen.

Realzeitverarbeitung ist eng verwoben mit dem Begriff der Task, die eine Aufgabe ausführt und der dafür eine genau definierte Zeit vorgegeben wird, aber die Aufgabe auch in dieser Zeit versehen muss.

Für den Bereich der Realzeit-Software scheint es zunächst sinnvoll zu sein, eine Definition des Begriffes Realzeit-System nach dem Oxford Dictionary of Computing anzugeben:

> „Any System [software] in which time at which output is produced is significant. This is usually because the input corresponds to some movement in the physical world and the output has to relate to the same movement. The lag from input time to output time must be sufficiently small for acceptable timeliness. “

Diese Definition gibt uns ein intuitives Verständnis dieser Art von Software bei der Zeit eine kritische Ressource ist. Weiterhin bleibt festzuhalten, dass zwischen Soft- und Hard Realtime-Systems unterschieden werden kann. Der Unterschied besteht in der Art, wie sich das Auftreten eines Fehlers, der eine Zeitschranke verletzt, äußert. Bei Hard Real-Time Systems kann ein Fehler, der eine Zeitschranke verletzt, katastrophale Auswirkungen nach sich ziehen und ist damit nicht

Entwurfsmustername	Entwurfsmusterbeschreibung	Beziehung zu anderen Domänen
Task-Muster	Legt als grundlegendes Muster für Realzeit-Software die periodische oder spontane Ausführung einer Verarbeitung fest und regelt die dafür bereitgestellten Ressourcen.	Keine
Asynchronous Commands	s. Beschreibung in Abschnitt 7.2	Verteilte Verarbeitung

Tabelle 7.3: Die Entwurfsmuster zur Realzeit-Verarbeitung

tolerierbar. Im Gegensatz hierzu bleibt bei Soft Real-Time Systems ein solcher Fehler ohne ernste Auswirkungen und kann toleriert werden. Hiernach lassen sich auch Anforderungen an Realzeit-Software klassifizieren. Die Masse der Anforderungen ist aber eher „weich", sodass oft die Hard-Realtime-Anforderungen nur einen kleineren Teil ausmachen. In der Tabelle 7.2 sind einige weitere Definitionen für wichtige Begriffe aus der Domäne der Realzeitverarbeitung angegeben worden.

In Tabelle 7.3 sind Entwurfsmuster für die Realzeitverarbeitung aufgeführt. Die für die Realzeit-Verarbeitung grundlegende Task wird durch das Task-Muster repräsentiert, das periodische und spontane Ausführung zulässt, für die Task-Raten und Deadlines festgelegt werden können.

7.3.1 Task-Muster

Dieser Abschnitt gibt eine Übersicht zum Begriff der Task, der für Realzeitsysteme von großer Bedeutung ist. Tasks werden häufig in Realzeit-Software angewendet.

Kontext: In Realzeitsystemen existieren nach [Dou99] zahlreiche Kontrollflüsse, die nebenläufig ausgeführt werden. Ein Thread setzt sich aus einer Menge sequentiell ausgeführter Aktionen, die eine zusammenhängende Funktionalität und eine bestimmte Priorität besitzt, zusammen.

Problem: Die Aktionen einer Task können durchaus zu vielen Objekten gehören. Es ist also eine Einheit zu finden, die sich aus mehreren Bestandteilen zusammensetzen kann. Dabei sind die zu den einzelnen Bestandteilen gehörenden strukturellen und verhaltensorientierten Aspekte zu berücksichtigen, die mit der UML selbstverständlich auf unterschiedliche Arten dargestellt werden können.

Lösung: Die gesamte Umgebung[1] eines solchen Threads wird als Task bezeichnet. In einer einzelnen Task kollaborieren in der Regel zahlreiche Objekte miteinander. Die Darstellung der strukturellen Aspekte einer Task werden im Klassendiagramm aus Abbildung 7.6 vorgestellt.

Weiterhin muss auch der Synchronisation von Tasks Aufmerksamkeit geschenkt werden. Dafür werden auch so genannte Taskdiagramme [Dou99] eingeführt, die ein Klassendiagramm angeben, welches vollständig aus aktiven Klassen besteht. In Abbildung 7.7 wird zunächst als Beispiel ein später noch erläutertes Klassendiagramm für den verteilten, entfernten Regler gezeigt, dessen Task-Diagramm in Abbildung 7.8 angegeben wird. Das Task-Diagramm besteht aus den aktiven Klassen *periodicTask*, *Sensor* und *Actor*. Anforderungen, die die Dienstqualität beschreiben – wie Scheduling-Policy, Taskperiode und Deadlines – können in diesem Diagramm durch OCL-Constraints dargestellt werden.

Betrachtet man die verhaltensorientierten Aspekte einer Task, ist es selbstverständlich, dass State- und Activity-Charts verwendet werden können. Ein Statechart kann das Verhalten des zu einer Task gehörenden Threads beschreiben, dabei muss wiederum die Synchronisation berücksichtigt werden.

Wie sieht nun die Task-Struktur einer verteilten Realzeitanwendung aus? Zunächst müssen Threads für Tasks definiert werden. Anschließend werden jedem Thread einer Task entsprechende Objekte hinzugefügt. Die Art und Weise, wie die Threads definiert und Objekte hinzugefügt werden, hängt sehr stark von den konkreten Anforderungen bzw. Randbedingungen der Software ab.

[1] Die gesamte Umgebung kann aus mehreren Objekten bestehen.

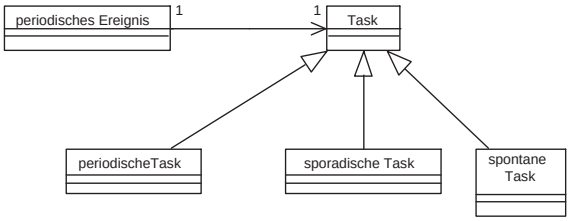


Abbildung 7.6: Das Klassendiagramm einer Task

Am Beispiel des entfernten, verteilten Reglers aus Abbildung 7.7 soll dieses Vorgehen erläutert werden. Die Klasse *periodicTask* modelliert den Regler. Die Klasse *SensorProxy* repräsentiert das Proxy eines Sensors. Diese Klasse entspricht dem Proxy-Muster aus [GHJV93, BMR+96]. Ebenso modelliert die Klasse *ActorProxy* das Proxy eines Aktors. Instanzen dieser drei Klassen werden am Ort des Reglers erzeugt. Die Klasse *Sensor* modelliert die Software eines Sensors, der eine physikalische Größe misst. Dementsprechend modelliert die Klasse *Actor* die Software des Aktors, der Änderungen an der Stellgröße vornimmt. Die Klassen *SensorAdapter* und *ActorAdapter* modellieren die Adapterobjekte für den Sensor und den Aktor. Diese Klassen lassen sich auf das Adapter-Muster aus [GHJV93, BMR+96] zurückführen. Sie befinden sich jeweils am Ort des Sensors bzw. des Aktors. Die Klassen werden im Klassendiagramm durch gerichtete Assoziationen im Klassendiagramm verbunden.

Man identifiziert bei dem entfernten, verteilten Regler die drei aktiven Klassen: *periodicTask*, *Sensor* und *Actor*, wie im Taskdiagramm in Abbildung 7.8 angegeben. Die Proxyobjekte gehören jeweils zum Thread der *periodicTask*. Die Adapterobjekte werden jeweils dem Sensor- bzw. Actorthread zugeordnet.

Weiterhin können Tasks für Ereignisquellen oder Senken definiert werden. Hierbei werden entweder Ereignisse gruppiert, die einer gemeinsamen Quelle entstammen, oder aber der Zugriff auf eine Schnittstelle durch eine Task bearbeitet. Wenn kein sehr komplexes System vorliegt, können Einzelereignisse lose verarbeitet werden. So kann für jedes interne oder externe Ereignis eine eigene Task definiert werden. Wenn eine Reihe von Schritten sequentiell durchgeführt wird, können die eigenen Aktionen in einem eigenen, einzelnen Thread einer Task zusammengefasst werden. Bei dem Beispiel des entfernten verteilten Reglers ist die Ereignisquelle der Sensor und die Ereignissenke der Aktor für die jeweils eine eigenständige Task vorgesehen ist.

Außerdem macht es häufig Sinn intensive Verarbeitungsvorgänge in einer Task zu kapseln. Das kann beispielsweise das Einfügen, das Entfernen oder die Filterung von Daten sein. Hierfür kann eine eigenständige Task, die etwa im Hintergrund arbeitet, verwendet werden. Ein weiteres wichtiges Kriterium ist die Kohärenz von Informationen einer Task. Diese Gruppierung kann angewendet werden, wenn die zusammenhängenden Daten in der Domäne des Benutzers verwendet werden. Hierzu werden die korrespondierenden Objekte in die Task eingefügt. Weiterhin sind bei der Konstruktion von Tasks natürlich zeitliche Charakteristiken von Bedeutung. Wenn Daten mit

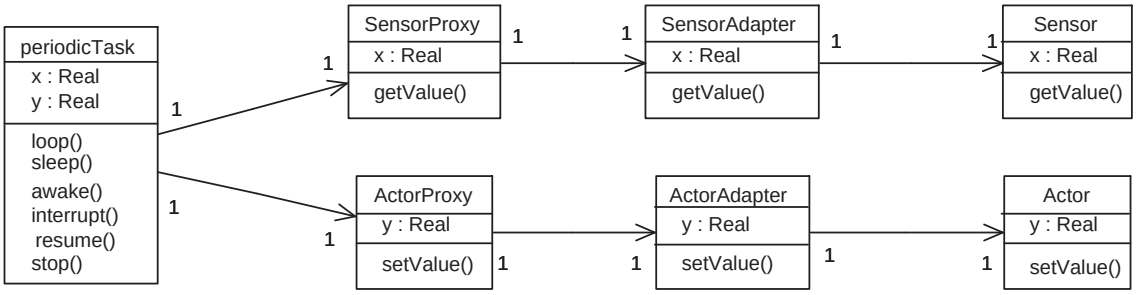


Abbildung 7.7: Das Klassendiagramm des objektorientierten Entwurfes eines *verteilten, entfernten Reglers*

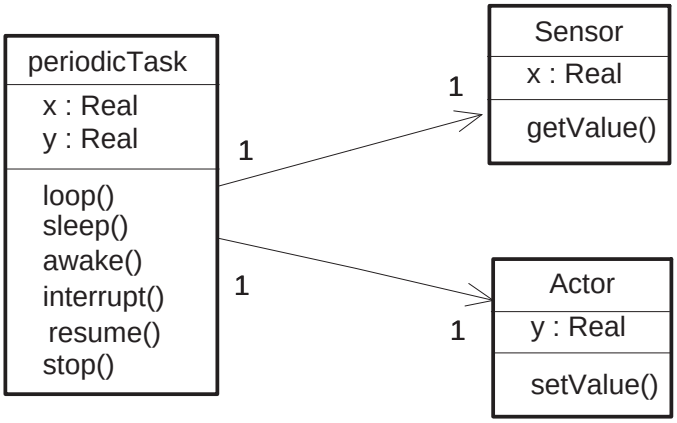


Abbildung 7.8: Das Taskdiagramm des objektorientierten Entwurfes eines *verteilten, entfernten Reglers*

einer bestimmten Rate ankommen, kann eine periodische Task diese aufnehmen und weiterleiten, falls erforderlich. Aperiodische Ereignisse können durch einen Interrupthandler verarbeitet und an entsprechende Objekte weitergeleitet werden.

Das letzte Kriterium für die Bildung von sinnvollen Tasks liegt für den Fall von Fehlertoleranz und Sicherheitsanforderungen vor. So ist es, sinnvoll innerhalb von Tasks, Sicherheitsprüfungen, bzw. sicherheitsrelevante Informationen zu verarbeiten als eigenständig festzulegen. Ein Zusammenlegung von verarbeitungs- und sicherheitsrelevantem Code sollte auf jeden Fall vermieden werden. Tasks, die sicherheitsrelevante Informationen verarbeiten, sind etwa Watchdogs, aber auch Alarme sollten in selbstständigen Tasks verarbeitet werden.

Zum gegenwärtigen Zeitpunkt sind mehrere Industrie-Standards verfügbar, die sich mit objektorientierter Realzeit-Software auseinandersetzen. Beispiele hierfür sind Realzeit-Java von Newmonics und RT-Java von der RT-Java Working Group. Jeder dieser Standards kennt Klassen, die dem Begriff der Task folgen. RT-Java kennt weiterhin die Realzeitaktivität, die aus einer oder mehreren Realzeit-Tasks zusammengesetzt wird. Zwischen diesen Tasks bestehen typische Beziehungen, die durch das Anwendungsgebiet vorgegeben werden. Beispiele für Realzeit-Tasks sind eine Videokonferenzsitzung, ein interaktives Videospiel, ein Fax-Modem Gerätetreiber oder ein Roboter-

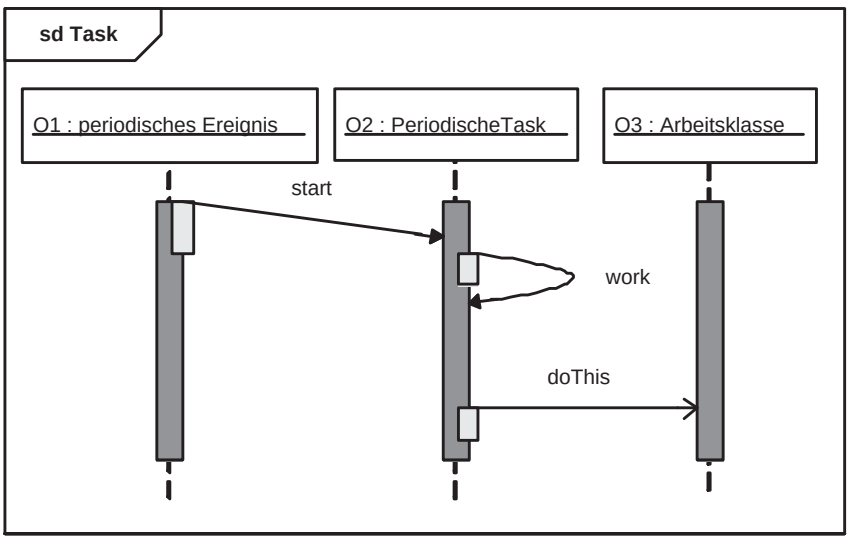


Abbildung 7.9: Sequenzdiagramm einer Task

Controller. Die Tasks für einen Roboter-Controller können beispielsweise aus jeweils einer Task für einen Roboter-Arm etc. bestehen.

Weil Realzeit-Java Programme unter den unterschiedlichsten Bedingungen ausführbar sein sollen, ist es erforderlich jede Aktivität vor ihrer Ausführung zu konfigurieren. Der Konfigurationsprozess besteht darin zu bestimmen, wie viel CPU-Zeit und Speicher für eine vorgegebene Arbeitslast auf einer bestimmten Plattform benötigt werden. Das Ergebnis einer Konfiguration besteht aus der Darstellung der Ressourcenanforderungen einer Aktivität, d. h. welcher Anteil an CPU-Zeit, der benötigte Speicherbedarf und die Rate mit der Speicher alloziert bzw. recycled wird. Diese werden jeweils als gewünscht bzw. notwendig eingestuft. Prinzipiell besteht die Anforderung, dass die Rate, mit der Speicher von Tasks einer Aktivität verbraucht wird, von den anderen Tasks einer Aktivität wieder freigegeben werden muss.

Die Konfiguration einer Aktivität führt zu einer Ressourcenverhandlung mit dem Realzeitkern des zugrundeliegenden Betriebssystems in deren Verlauf ein Ressourcenbudget ausgehandelt wird. Die Aktivität antwortet auf das vorgeschlagene Budget mit einer Annahme oder einer Ablehnung, was sich daran orientiert, ob ihr das Budget akzeptierbar erscheint.

Wenn neue Aktivitäten in das System eingeführt bzw. existierende gelöscht werden, muss die systemweite Ressourcenzuordnung möglicherweise revidiert werden. Außerdem kann jede Aktivität zur Laufzeit eine Veränderung ihres Budgets anstoßen, genauso wie der Kern des Betriebssystems eine solche Neuverhandlung auslösen kann.

Eine RT-Java-Task besitzt nach [J C00] die nachfolgend aufgeführten Operationen:

- Die Operation *currentTask* liefert eine Referenz auf die gegenwärtig ausgeführte Task.
- Operationen zur Festlegung von Prioritäten.
- Die Operation *CoreTask.abort* unterbricht die gegenwärtig laufende *work()* Operation.
- Die Operation *CoreTask.join* blockiert die gegenwärtig laufende Task, bis die einzufügende Task beendet ist.
- Die Operation *CoreTask.resume* kann eine gegenwärtig suspendierte Task durch Erhöhen ihrer Priorität wieder zum Leben erwecken.
- Die Operation *CoreTask.setPriority* ermöglicht die Einstellung der Priorität einer gegenwärtig laufenden Task.
- Die Operation *CoreTask.sleep* legt eine Task für eine minimale Zeit schlafen. Diese minimale Zeit wird als Parameter übergeben.
- Die Operation *CoreTask.start* startet die *work*-Operation einer Task.
- Die Operation *CoreTask.stop* erlaubt es, dass eine gerade ausgeführte Task–ähnlich wie bei *abort* – angehalten wird. Außerdem können noch einige Nachbehandlungen bei der Task vorgenommen werden.
- Die Operation *CoreTask.suspend* setzt die Priorität der gegenwärtig laufenden Task möglichst niedrig.
- Die Operation *CoreTask.work* enthält die von der Task zu entrichtende Arbeit bzw. deren Algorithmus.

Die Abbildung 7.9 zeigt in einem Sequenzdiagramm exemplarisch, wie die Interaktionen einer Task aussehen. Die Task wird durch ein periodisches Ereignis (hier *O1*) gestartet. Anschließend beginnt die Verarbeitung in der Task, was durch den *work* Operationsaufruf auf das Object *O2*, das hier eine periodische Task ist, beschrieben ist. Mit dem *doThis* Aufruf kann *O2* Verarbeitungsschritte an ein Objekt *O3* auslagern.

Realzeit-Tasks können in unterschiedliche Typen, die für verschiedene Anwendungen benötigt werden, klassifiziert werden, die im Klassendiagramm aus Abbildung 7.6 angegeben werden:

- Periodische Tasks werden regelmäßig für eine bestimmte Ausführungszeit wieder aufgerufen. Ausführungszeit und Periode werden durch den Anwendungsprogrammierer und anschließender Verhandlung mit dem Kern des Realzeit-Betriebssystems festgelegt. In RT-Java wird eine periodische Task durch ein periodic Event Objekt gestartet.

- Sporadische Tasks werden durch externe Ereignisse aufgerufen, die in einem bestimmten Zeitraum bearbeitet werden müssen. Für sporadische Tasks werden vorab Ressourcen reserviert. Der Ursprung eines Ereignisses, das die Task auslöst, ist bei Realzeit-Java und RT-Java softwarebasiert.

- Spontane Tasks werden als Ergebnis einer dynamischen Bedingung aufgerufen. Der Ursprung für eine spontane Task muss durch Software ausgelöst werden. Für spontane Tasks werden vorab im Gegensatz zu sporadischen Tasks keine Ressourcen reserviert, d. h. der Kern bestimmt während der Ausführungszeit, ob genügend Ressourcen zur Verfügung stehen, um eine Anfrage zu bedienen. Ist dies der Fall, wird die Task zur gegenwärtigen Arbeitslast hinzugefügt, ansonsten wird sie abgewiesen.

Beispiel: Zur Umsetzung der Realzeitanforderungen der Beispielanlage aus Abbildung 2.4 wird jede separate Steuerung durch Tasks realisiert. Tasks werden auch im Rahmen des Informationsmodells der IEC 61131-3 [Tec09] verwendet, um Steuerungssoftware zu realisieren.

7.4 Entwurfsmuster für die fehlertolerante Verarbeitung

In diesem Abschnitt sollen Grundlagen der Fehlertoleranz kurz umrissen werden, bevor Entwurfsmuster, die die fehlertolerante Verarbeitung ermöglichen, angegeben werden. Die Definitionen sind aus [Ech90] und [LT90] entnommen und wurden zur Beschreibung von Softwareeigenschaften angepasst.

In Tabelle 7.4 werden einige für diese Arbeit wichtige Begriffe aus dem Gebiet der Fehlertoleranz angegeben. Grundlegend ist der Begriff des Fehlers, welcher das Betreten eines nicht erlaubten Zustandes bzw. die Nichtverfügbarkeit einer Funktion angibt. Es gibt eine Vielzahl von Fehlern. In dieser Arbeit sind insbesondere Spezifikations- und Betriebsfehler von Interesse. Fehler in der Spezifikation und deren Umsetzung sollen durch Einsatz von cTLA zur Verifikation von Software vermieden werden, während Betriebsfehler sinnvollerweise in der Software spezifiziert werden müssen, was durch Angabe bestimmter Zustände, die im Fehlerfall eingenommen und durch Einleitung bestimmter Maßnahmen wieder verlassen werden können, vorgenommen werden kann. Insgesamt sollen durch diese Maßnahmen zur Korrektheitssicherung Fehler beim Entwurf vermieden werden. Dies führt dazu, dass ein Softwaresystem fehlertolerant wird, was bedeutet, dass auch beim Ausfall von Komponenten die Funktionalität eines Systems erhalten bleibt.

Die Tabelle 7.5 gibt eine Übersicht von Entwurfsmustern an, die bei der fehlertoleranten Verarbeitung zum Einsatz kommen. Für das Aufspüren von Betriebsfehlern können das Watchdog- und das Master-Slave-Muster eingesetzt werden. Das Watchdog-Muster erlaubt eine Prüfung von Systemfunktionalitäten bzw. Systemausgaben in periodisch wiederkehrenden Zeiträumen und ermöglicht die Entdeckung bzw. das Aufspüren von Betriebsfehlern. Sollte ein Fehler vorliegen können entsprechende Maßnahmen zur Fehlerbehebung bzw. zum Recovery eingeleitet werden. Durch das Master-Slave-Muster hingegen können Betriebsfehler durch die Verwendung von Redundanz entdeckt werden. Die Muster für Vor- und Rückwärtsbehebung gestatten es, Betriebsfehler, die während der Bearbeitung eingetreten sind, zu beheben. Die vertikale Fehlereingrenzung ist ein Architekturmuster, das es erlaubt, Fehler zwischen Schichten einzugrenzen.

7.4.1 Master-Slave-Muster

Kontext: Ein Entwurfsmuster, das sich mit Fehlertoleranz befasst und auch im Hardwarebereich zum Einsatz kommt, ist das Master-Slave Muster. Es kann unter anderem dazu eingesetzt werden, um den Entwurf von fehlertoleranten Systemen zu ermöglichen Andere Anwendungsfelder liegen im Bereich der Unterstützung von Parallelverarbeitung.

Bezeichnung	Definition
Fehler	Fehlzustände, die einen unzulässigen Zustand einer Komponente bezeichnen oder Funktionsausfällen, die eine unzulässige bzw. aussetzende Funktion einer Komponente beschreiben. Es gibt unterschiedliche Ursachen von Fehlern, welche im Entwurf (Spezifikation, Implementierung, Dokumentation), der Herstellung und dem Betrieb (Störung, Verschleiß, zufällige physikalische Fehler, Bedienung, Wartung) liegen.
Spezifikationsfehler	Unvollständigkeiten, Abweichungen oder Widersprüche bei der Überführung von Benutzeranforderungen in eine formale, verbale Software-Spezifikation.
Fehlertoleranz	Fähigkeit eines Systems – auch mit einer begrenzten Anzahl fehlerhafter Komponenten – seine spezifizierte Funktion zu erfüllen.
Betriebsfehler	Diese erzeugen während der Nutzungsphase eines Softwaresystems einen fehlerhaften Zustand in einem vormals fehlerfreien System. Ein solcher Fehler tritt erst bei der Inbetriebnahme eines Systems auf.
Fehlervermeidung	Verbesserung der Zuverlässigkeit durch Perfektionierung der konstruktiven Maßnahmen, um das Auftreten von Fehlern von vornherein zu vermeiden. Dies lässt sich durch sorgfältigeren Entwurf, eine erhöhte Anzahl von Tests und Verbesserungen vor Inbetriebnahme, Verwendung von geeigneten Materialien und verbesserten Herstellungstechniken erreichen.
Fehlermodell	Strukturierung eines Systems in Komponenten und Angabe der Fehlermöglichkeiten der einzelnen Komponenten, zumindest in qualitativer Form.
Menge zu tolerierender Fehler	Die Menge zu tolerierender Fehler gibt an, welche und wie viele Fehler des Fehlermodells zu tolerieren sind. Auch der Fall, in dem kein Fehler auftritt, wird selbstverständlich toleriert.

Tabelle 7.4: Die Definitionen für das Gebiet der fehlertoleranten Verarbeitung

Problem: Kann eine Aufgabe in mehrere Teilaufgaben zerlegt werden, deren Verarbeitung voneinander unabhängig erfolgt, so kann dieses Muster verwendet werden. Anschließend müssen die Lösungen der Teilaufgaben wieder zusammengeführt werden. Bei der Konstruktion eines solchen Systems ist das softwaretechnische Prinzip Separation of Concerns zu beachten. Weiterhin ist der korrekten Koordination der Teilergebnisse Beachtung zu schenken, die entsprechend kombiniert werden müssen, um eine Lösung zu erhalten.

Lösung: Im Folgenden soll eine Lösung dieses Problems betrachtet werden. Ein Master erhält von einem Klienten einen Auftrag zur Ausführung eines Dienstes. Er delegiert die Ausführung dieses Auftrages einfach an eine feste Zahl von Komponenten, die eine identische Funktionalität

Entwurfsmustername	Entwurfsmusterbeschreibung	Beziehung zu Mustern anderer Domänen
Watchdog	Nimmt die Prüfung der Systemfunktionalität in regelmäßigen Zyklen vor.	Keine
Master-Slave	Redundante Verarbeitung, um Fehler zu entdecken.	Keine
Recoverable Distributed Observer	fehlertolerante Sicherstellung der Datenkonsistenz	Verteilte Verarbeitung

Tabelle 7.5: Entwurfsmuster zur fehlertoleranten Verarbeitung

aufweisen und den Auftrag unabhängig voneinander bearbeiten. Durch jede dieser Komponenten wird ein Slave repräsentiert. Einem potentiellen Klienten, welcher den Master um die Bearbeitung einer Aufgabe bittet, liefert der Master das Ergebnis des Klienten zurück, der als erster ein Ergebnis liefert. Dies sichert die Fehlertoleranz, da der Klient ein gültiges Ergebnis erhält, wenn wenigstens ein Slave ein Ergebnis für seine Aufgabe ermittelt hat. Falls kein Slave ein Ergebnis liefert, wird durch den Slave eine Fehlerbehandlung eingeleitet, die unter Umständen dem Klienten durch einen Fehlerwert angezeigt wird. Um zu ermitteln, ob ein Klient sein Ergebnis zu einem gültigen Zeitpunkt abgeliefert hat, können Zeitschranken eingesetzt werden. Besonderes Augenmerk ist der Stabilität des Masters selbst zu widmen, der die zentrale Komponente repräsentiert, deren Ausfall schwer zu kompensieren ist. Zur Garantie der Stabilität dieses Entwurfsmusters kann das Watchdog-Muster verwendet werden.

7.4.2 Watchdog-Muster

Ein Watchdog ist ein häufig verwendetes Konzept in eingebetteten Realzeitsystemen [Dou99]. Es ist ein Subsystem, das Nachrichten von anderen Subsystemen auf einer periodischen oder schlüsselbasierten Grundlage untersucht. Wenn ein Dienst des Watchdogs zu spät oder außerhalb der Reihenfolge auftritt, führt der Watchdog einige Korrekturaktionen, wie das Rücksetzen, Herunterfahren, Alarmieren oder Initiieren eines ausgefeilteren Recoverymechanismus des Systems, durch. Watchdogs sind sehr einfache Softwarekomponenten und werden oft durch Hardware realisiert, um sie vor Softwarefehlern zu schützen. Manchmal führen Software Watchdogs weitergehende Aktivitäten durch. Solche Watchdogs werden periodisch aufgerufen und führen eine eingebaute Sammlung von Tests durch, wie z. B. CRC Prüfungen über ausführbarem Code, Prüfungen des Arbeitsspeichers, Überprüfung von Anzeichen auf Stackoverflows. Obwohl dies keine Watchdogs im klassischen Sinne sind, sind sie einfach zu implementieren und werden in vielen sicherheitskritischen Umgebungen benötigt. Der Vorteil der Verwendung von Watchdogs liegt in der billigen und einfachen Implementierung, die keine komplizierte Hardware- oder Softwareunterstützung erfordert. Dennoch haben viele Systeme keine einfache Antwort auf die Entdeckung eines Systemfehlers. Der Hauptnachteil dieses Entwurfsmusters besteht darin, dass es zu einfach strukturiert ist, um eine komplexe Fehlerbehandlung bzw. ein Fault-Recovery herzustellen.

Beispiel: In der Beipielanlage aus Abbildung 2.4 wird das Watchdog-Muster verwendet, um sicherzustellen, dass die Verriegelungssteuerung gewährleisten kann, dass die Membranpumpe nicht beschädigt wird.

Kapitel 8

Analysemuster für das abstrakte Softwaremodell

Bei der Konstruktion von Steuerungssoftware ist es naheliegend, sich an etablierten Prozessmodellen – wie etwa dem RUP (Rational Unified Process) – für die Softwareentwicklung zu orientieren. Diese beginnen in der Regel mit der Entwicklungsphase der Anforderungsanalyse, in der Anforderungen an ein Softwaresystem in abstrakter Form festgehalten werden. Anforderungen können in funktionaler und nicht-funktionaler Form vorliegen. Mögliche nicht-funktionale Anforderungen sind Beschreibungen zur geforderten Antwortzeit, zur erwarteten Performanz, zur gewünschten Verteilung und zur erforderlichen Fehlertoleranz. An die Entwicklungsphase der Anforderungsanalyse schließt sich der Entwurf an, in dem eine Architektur, die sämtliche Anforderungen erfüllt, entwickelt wird. Objektorientierte Modellierungssprachen ermöglichen es, für beide Phasen Modelle zu entwickeln, die auf den gleichen Konzepten beruhen und die Wiederverwendung – z. B. durch Verwendung von Mustern – unterstützen.

Erfahrungen zeigen, dass sich die Modelle der Anforderungsanalyse und des Entwurfs auf einem anderen Abstraktionsniveau bewegen. Im abstrakten Softwaremodell (ASM), das sich auf einem logischen bzw. abstraktem Niveau bewegt, sollen die funktionalen Anforderungen und die Realzeit-Anforderungen an eine Steuerungssoftware spezifiziert werden. Im konkreten Software Modell (KSM) hingegen, das sich auf konkretem technischen Niveau befindet, sollen zusätzlich die nicht-funktionalen Anforderungen der Steuerungssoftware berücksichtigt werden.

In diesem Kapitel liegt der Schwerpunkt auf dem Entwurf eines Analysemuster-System von Software für das Fachgebiet der Steuerungstechnik, das den Modellierer bei der Erstellung des ASM durch Bereitstellung eines Mustersystems von Analysemustern unterstützt. Ein Analysemuster-System für eine verteilte Fertigungszelle ist in [Mis99] behandelt worden. Dazu wird der Begriff des Analysemuster-Systems kurz vorgestellt. Anschließend wird gezeigt, wie man ein Analysemuster-System für Steuerungssoftware aus dem erörterten Architekturmodell ableitet. Im Folgenden wird die Struktur dieses Analysemuster-Systems mit den Beziehungen der Analysemuster vorgestellt und anschließend werden die einzelnen Muster erörtert.

8.1 Analysemuster-System für Prozesssteuerungs-Software

Im Gegensatz zur existierenden Literatur wird im Folgenden der Begriff des Mustersystems auch auf Analysemuster und deren Beziehungen ausgedehnt. Dies wird als Analysemuster-System bezeichnet. Es ist aus Kapitel 4 bekannt, dass bei einem Analysemuster [RZ96] die Form durch die Begriffe und die Konzepte einer Anwendungsdomäne beschrieben wird.

Bisher in der Literatur existierende Ansätze, die Mustersysteme beschreiben [BMR+96], beziehen sich auf Mustersysteme, die aus Entwurfsmustern bestehen. Hierfür soll statt dessen in dieser Arbeit der Begriff des Entwurfsmuster-Systems verwendet werden. Entwurfsmuster-Systeme werden im folgenden Kapitel behandelt.

Das wichtigste Ziel für die Entwicklung eines Analysemuster-Systems sollte darin bestehen, eine problemnahe, abstrakte und logische Darstellung von Problemlösungen für den Modellierer bereitzustellen, wie sie bei der Anforderungsanalyse erforderlich sind. Man erreicht dieses Ziel, indem man die Struktur und das Verhalten der logischen und problembezogenen Vorgabe durch das Muster festhält. Analysemuster-Systeme erlauben es, Analysewissen aufzunehmen und zwischen Modellierern zu propagieren. Ein Analysemuster-System besteht aus einer Sammlung von Analysemustern und legt außerdem die Beziehungen der Analysemuster untereinander fest. Sowohl die Analysemuster als auch deren Beziehungen müssen geeignet dokumentiert werden, wobei die Dokumentationskonzepte des Kapitels 4 Verwendung finden. Es können durchaus unterschiedliche Arten von Beziehungen auftreten, wie z. B. Benutzt-Beziehungen sowie Beziehungen, die die Kombinierbarkeit von Analysemustern angeben. Ein typisches Beispiel für ein Analysemuster aus der Domäne der Steuerungssoftware ist das Reglermuster, das – wie bereits in Kapitel 3 beschrieben wurde – aus einem Regler, einem Aktor und einem Sensor besteht. Bei der Analyse der Domäne für Steuerungssoftware ergibt sich, dass dieses Muster häufig in Beziehung zu anderen Steuerungen steht.

Die Struktur der Lösung, die ein Analysemuster anbietet, wird durch ein Klassendiagramm spezifiziert, während das im zeitlichen Ablauf auftretende Geschehen durch Interaktions- und Verhaltensmuster festgehalten wird. Interaktionsmuster spezifizieren die Interaktion von Objekten, die Instanzen aus dem Klassendiagramm sind und deren Verhalten durch Verhaltensmuster beschrieben wird. Bei der Modellierung von komplexeren Problemen können Interaktionsmuster kombiniert werden, wobei auch Überlappungen zwischen den Interaktionsmustern auftreten können.

8.2 Analysemuster-System und Domänenmodell

Im Folgenden soll zunächst eine an einer Top-Down-Vorgehensweise angelehnte Struktur für Steuerungssoftware angegeben werden, die sich an den Ebenen des Kapitels 2 über die Architektur von Steuerungssoftware orientiert. Das Ziel bestand darin, ein abstraktes Analysemuster-System für die Architektur von Steuerungssoftware zu finden. Aus den funktionalen Anforderungen und der Architektur von Systemen zur Prozesssteuerung, wie sie im Kapitel 2 vorgestellt wurden, werden Analysemuster identifiziert, deren Form durch die Domäne der Steuerungstechnik vorgegeben wird und diese in möglichst weitem Umfang überdecken. Dies sollte nach folgenden Prinzipien geschehen:

- Eine Konzentration auf die systembezogenen Funktionalitäten soll vorgenommen werden, welche sich an der Struktur des Domänenmodells orientiert.
- Die Umsetzung der Elemente des Domänenmodells in ein Analysemuster-System soll unterstützt werden.
- Die Systembildung durch Musterkombination soll unterstützt werden.

Abbildung 8.1 zeigt ein Paketdiagramm, das angibt, wie die Ebenen des Architekturmodells durch Pakete strukturiert werden können. Pakete werden zur Funktions- bzw. Informationskohäsion verwendet. Bei der funktionalen Kohäsion werden alle Pakete, die einem ähnlichen Zweck dienen, also funktional kohäsiv sind, zusammengefasst. Alle Bestandteile (Entitäten bzw. Klassen) des Architekturmodells, die untereinander stark und nach außen hin schwach gekoppelt sind, können durch ein Paket zusammengefasst werden. Das Paketdiagramm in Abbildung 8.1 gibt an, dass die Pakete der Feldebene, der Prozesssicherung, der Prozessstabilisierung und der Prozessleitung jeweils als ein Modell mit dem Stereotyp <<model>> spezifiziert werden. Dieses UML-Stereotyp gibt an, dass die im zugehörigen Paket enthaltenen Modelle auf einem hohen Abstraktionsniveau modelliert sind.

Die Entitäten der einzelnen Architekturebenen werden anschließend in Steuerungsobjekte umgesetzt, die in den Paketen der einzelnen Ebenen modelliert werden. Zwischen den Paketen werden entsprechend ihrer Position in der Hierarchie der Architektur Abhängigkeiten angegeben, die mit dem <<access>>-Stereotyp versehen werden. Mit dem <<access>>-Stereotyp gekennzeichnete

Abhängigkeiten zwischen Paketen ermöglichen es zu beschreiben, dass ein Paket auf ein anderes zugreifen kann, aber dessen Inhalt nicht importiert. Mit einer <<access>>-Abhängigkeit kann aber ausgedrückt werden, dass Elemente des zugreifenden Pakets jede Art von Beziehung mit den Bestandteilen des zugegriffenen Paketes eingehen können. Abschließend gibt es ein Paket, das ebenenübergreifende Entitäten und Funktionalitäten besitzt, die keiner Ebene eindeutig zugeordnet werden können, aber für jede Ebene wertvolle Unterstützung liefern. Alle übrigen Pakete, die den einzelnen Ebenen zugeordnet werden, greifen auf dieses Paket zu. Die Definition der Pakete und deren Abhängigkeiten unterstützen die Bildung eines Analysemuster-Systems.

Zur Unterstützung der Systembildung durch Musterkombination sollen weitere Beziehungen eingeführt werden. Dafür soll die Architektur nach Beziehungen zwischen Mustern untersucht werden. Auch diese Beziehungen sollen anschließend im Analysemuster-System dokumentiert werden. Die Beziehungen zwischen den Mustern eines Systems können durchaus unterschiedlicher Natur sein und beispielsweise beschreiben, wie Muster gekoppelt bzw. kombiniert werden. Für die Kopplung von Analysemustern sind, wie aus dem Kapitel 4 bekannt, die beiden Möglichkeiten der losen Kopplung und Überlappung von Mustern vorhanden. Die Kopplungsproblematik ist besonders in Diagrammen, die die Interaktion von Mustern beschreiben, von Bedeutung. In der vorgestellten Architektur können durchaus Beziehungen zwischen Steuerungsobjekten unterschiedlicher Architekturebenen auftreten, was speziell an den Schnittstellen zwischen den Ebenen geschieht.

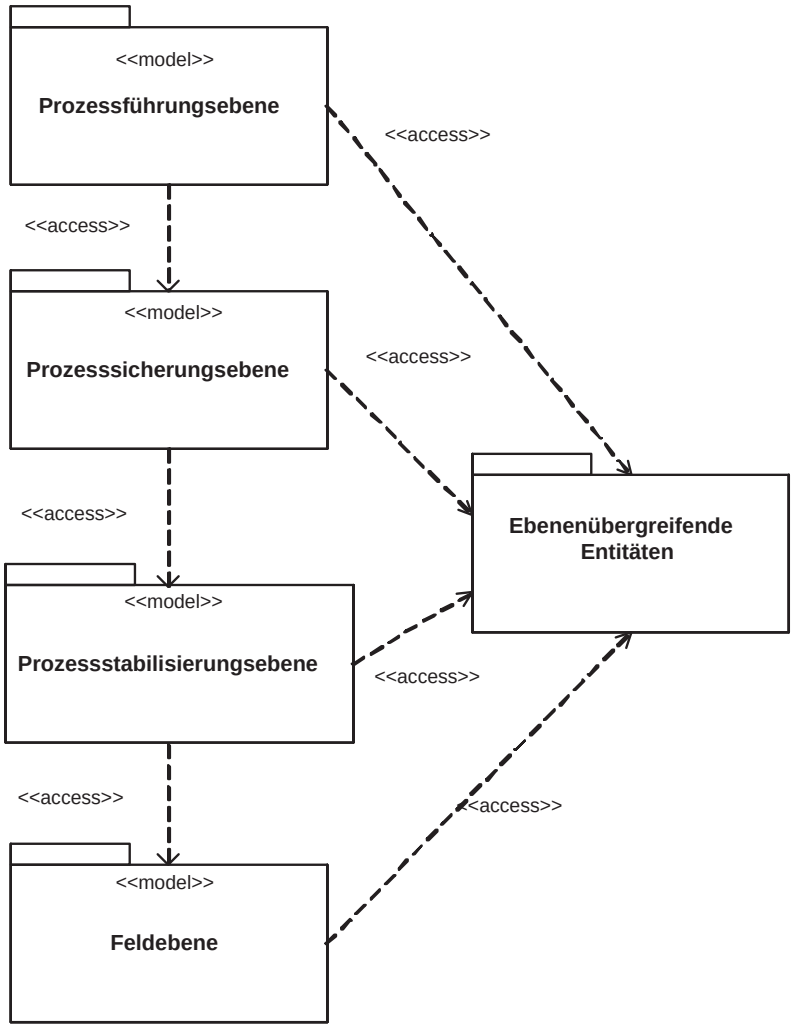


Abbildung 8.1: Ein Paketdiagramm zur Strukturierung der Ebenen

8.3 Struktur des Analysemuster-Systems

Um die Struktur des Analysemuster-Systems zu beschreiben, ist es notwendig weitere Abstraktionsebenen in das Paketdiagramm aus Abbildung 8.1 einzuziehen, die sich aus einer Bottom-Up orientierten Vorgehensweise ergeben, wie es in Abbildung 8.2 geschehen ist. Pakete, die Analyse-

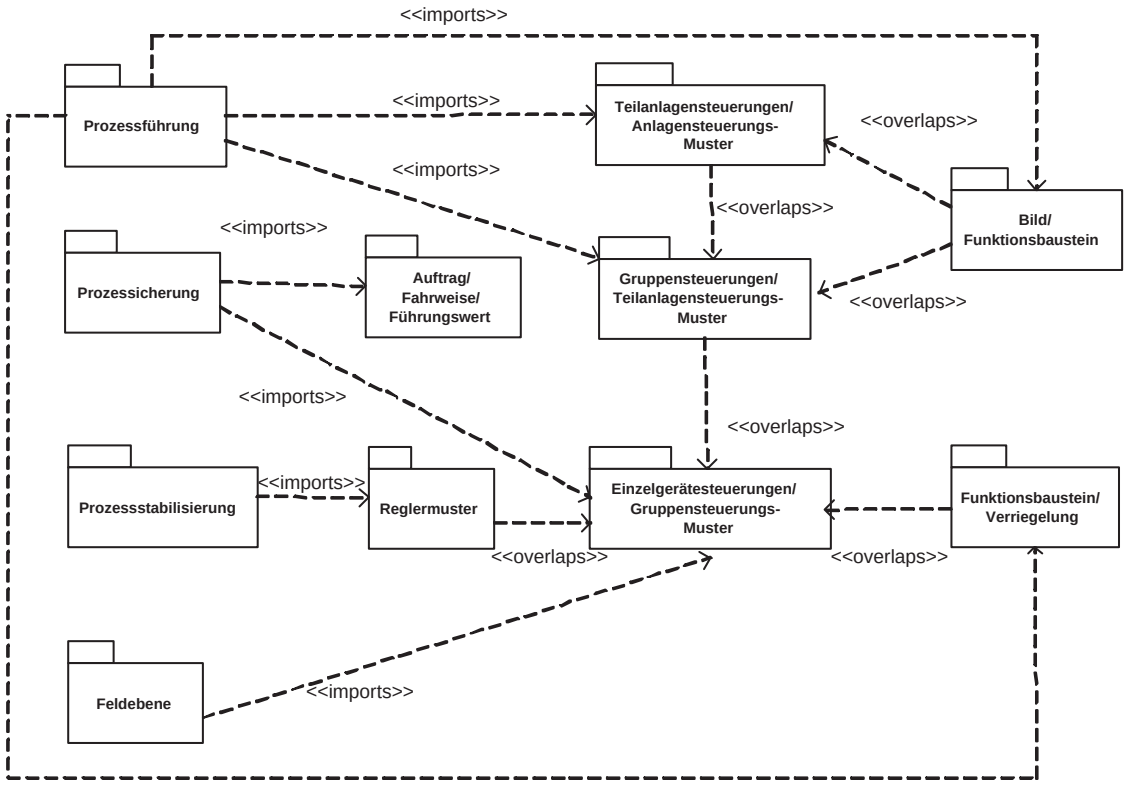


Abbildung 8.2: Das Analysemuster-System mit seinen Mustern und deren Beziehungen

muster enthalten, werden mit dem <<analysis pattern>> Stereotypen versehen und entsprechen somit einem Muster des Analysemuster-Systems. Es werden zahlreiche Analysemuster, die die wichtigsten Aspekte der Analyse von Steuerungssoftware abdecken, angegeben. Zur Erfassung der Beziehungen innerhalb des Analysemuster-Systems und der Analysemuster sowie der Architekturebenen werden die folgenden Abhängigkeiten aufgeführt:

- Wird ein Analysemuster mit seinen Klassen in das Paket einer Architekturebene übernommen, so wird dies durch eine Abhängigkeit durch das <<import>>-Stereotyp angegeben.
- Überlappen sich die Funktionalitäten verschiedener Analysemuster, weil in beiden gleiche Funktionalitäten durch dasselbe Objekt bereitgestellt werden, wird das <<overlaps>>-Stereotyp verwendet.

Die <<imports>>-Abhängigkeiten erweitern die Semantik von <<access>>-Abhängigkeiten so, dass damit auch der Import von Elementen aus einem fremden Paket in das Paket, von dem die Abhängigkeit ausgeht, möglich ist. Für Pakete kann die Sichtbarkeit ihres Inhalts nach außen angegeben werden. Dies geschieht durch Kennzeichnung des Klassennamens mit einem Präfix + für öffentlich, - für privat und # für protected. Damit können Bestandteile des Inhalts als öffentlich bzw. nicht öffentlich angegeben werden. Hat ein Paket eine <<access>>- bzw. <<imports>>-Abhängigkeit zu einem anderen Paket, dann sind alle öffentlich sichtbaren Elemente dieses anderen Paketes innerhalb des zugreifenden Paketes sichtbar. Beziehungen zwischen den einzelnen Mustern werden durch Abhängigkeiten angegeben, die mit Stereotypen versehen werden. Transitivität wird bei <<imports>>-Abhängigkeiten nicht unterstützt.

Orientiert man sich an [QC99], der die Kombinationsmöglichkeiten von Mustern diskutiert, können sich Muster sowohl überlappen als auch lose miteinander kombiniert werden. Daher wird das <<overlaps>>-Stereotyp neu eingeführt. Dieser bezieht sich nicht bloß auf den Zugriff, Im- bzw. Export von Entitäten aus anderen Paketen, sondern auch dieses Stereotyp zieht Anforderungen an den Aufbau von Interaktionsmustern nach sich. Besteht eine derartige Abhängigkeitsbeziehung zwischen zwei Paketen, so bedeutet das, dass zwischen den beiden zugehörigen Mustern Überlappungen bestehen können, indem beispielsweise Interaktionsdiagramme über Objekte der gleichen Klassen verfügen bzw. Objekte sogar identisch sind, d. h., dass in den zugehörigen Interaktionsmustern Objekte mit demselben Verhalten, die möglicherweise sogar dem gleichen Verhaltensmuster unterliegen (bei gleicher Identität als identischer Objektbezeichner), auftreten können. Es bietet sich an solche Objekte mit dem <<global>>-Stereotyp zu versehen, das angibt, dass die Objekte eines Sequenzdiagramms auch in anderen Sequenzdiagrammen Verwendung finden können.

Nachdem die Struktur der Pakete und deren Beziehungen vorgestellt worden sind, sollen im Folgenden die einzelnen Muster mit ihren Beziehungen in der Abbildung 8.2 behandelt werden. In der Abbildung werden aus Gründen der Übersichtlichkeit nur die wichtigsten Überlappungsmöglichkeiten aufgeführt. Zur Beschreibung der Ebenen des Architekturmodells werden die Pakete aus dem Architekturmodell verwendet, die die Namen der entsprechenden Ebenen enthalten. Wenn ein Muster eindeutig einer Ebene zugeordnet werden kann, wird das zugehörige Paket durch eine <<import>>-Abhängigkeit mit dem entsprechenden Paket versehen.

8.4 Analysemuster für die Domäne

In diesem Abschnitt werden die einzelnen Muster des Analysemuster-Systems vorgestellt. Zunächst wird eine Übersicht aller Analysemuster angegeben. Jedes Analysemuster wird dann in einem separaten Abschnitt betrachtet. Dabei enthält das Schema zur Musterbeschreibung neben den in Kapitel 4 genannten Bestandteilen *Kontext*, *Problem*, *Lösung* zusätzlich die optionalen Komponenten *Beispiel* und *Überlappung*. Das Beispiel gibt eine mögliche Verwendung im Rahmen der Steuerungssoftware aus der Beispielanlage des Kapitels 2 an. Falls bei dem Analysemuster Überlappungen zu anderen Analysemuster existieren, werden diese unter *Überlappung* angegeben. Im Folgenden werden die Analysemuster kurz aufgeführt bevor, jedes in einem eigenen Abschnitt ausführlich erläutert wird:

- Das Einzelgerätesteuerungen/Gruppensteuerungs-Muster, das innerhalb der Prozesssicherungsebene Verwendung findet, behandelt das Zusammenspiel von Einzelgerätesteuerungen und deren übergeordneten Gruppensteuerungen.
- Das Auftrag/Fahrweise/Führungswert-Muster hält das Zusammenwirken von Auftrag, zugeordneten Fahrweisen und Führungswerten fest.
- Das Gruppensteuerungen/Teilanlagensteuerungs-Muster beschreibt die Kooperation von Gruppensteuerungen und Teilanlagensteuerungen, die auf der Prozessführungsebene zur Anwendung kommen.
- Das Teilanlagensteuerungen/Anlagensteuerungs-Muster, das ebenfalls auf der Prozessführungsebene Anwendung findet, modelliert, wie Anlagensteuerungen mit Teilanlagensteuerungen zusammenwirken.
- Das Funktionsbaustein/Verriegelungs-Muster spezifiziert die Zusammenarbeit von Funktionsbausteinen und Verriegelungen, die die Sicherheit in Anlagensteuerungen gewährleisten. Dieses Muster findet Verwendung in der Prozessstabilisierungsebene.
- Das Reglermuster, welches zur Prozessstabilisierungsebene gehört, stabilisiert die Zustände von Teilen der Feldebene unter Verwendung unterschiedlicher Regelstrategien (z. B. als PID- oder Fuzzy-Regler) stabilisiert. Das Muster kann sich mit dem Einzelgerätesteuerungsmuster überlappen.

- Das Bild/Funktionsbaustein-Muster, das der Prozessführungsebene zugeordnet wird, nimmt die Anzeige von Zuständen bzw. Störungen von Funktionsbausteinen in Bedienpulten, Steuerständen und Konsolen vor.
- Das Auftrag/Befehl/Funktionsbaustein-Muster hält das Zusammenwirken von Aufträgen und Befehlen in Funktionsbausteinen fest.
- Differenzierer und Integrierer [Unb92] sind klassische Bausteine aus der Regelungs- und Anlagensteuerungstechnik, für die hier ebenfalls Analysemuster angegeben werden sollen.

8.4.1 Auftrag/Fahrweise/Führungswert-Muster

Dieser Abschnitt dient der Vorstellung des Auftrag/Fahrweise/Führungswert-Musters. Bei der Kommunikation zwischen Funktionsbausteinen müssen häufig Informationen bezüglich bestimmter Fahrweisen an die Steuerungen von Anlagenteilen, die auch Führungswerte enthalten, übermittelt werden.

Kontext: In Funktionsbausteinen werden Aufträge durch Befehle gestartet. Diese Aufträge sind für die weitere Verarbeitung durch Funktionsbausteine von Bedeutung.

Problem: Wie können Aufträge, die aus Fahrweisen und Führungswerten zusammengesetzt sind, modelliert werden?

Lösung: Da Führungswerte mit den Fahrweisen in einem Auftrag zusammengefasst werden, bietet sich die Komposition der Klasse *Fahrweise* und der Klasse *Führungswert* durch die Klasse *Auftrag* an (s. Abbildung 8.3). Für jeden Auftrag sind sowohl die Klasse *Führungswert* als auch die Klasse *Führungswert* mit geeigneten Werten parametrisiert. Die Klasse *Auftrag* besitzt die Attribute *Name*, *Datum* und *Auslöser*, um einen Auftrag zu identifizieren. Somit kann jede Fahrweise mit ihren entsprechenden Führungswerten in Verbindung gebracht werden, die auch als Parameter zur Verfügung gestellt werden können. Das Muster konzentriert sich auf die strukturelle Beschreibung von Daten, sodass hier kein Sequenzdiagramm angegeben wird.

Beispiel: Wenn in der Beispielanlage aus Abbildung 2.4 etwa ein Auftrag zum Trocknen bzw. Mahlen gestellt wird, müssen Führungswerte für die Körnung bzw. Trockentemperatur bereitgestellt werden.

Überlappung: keine

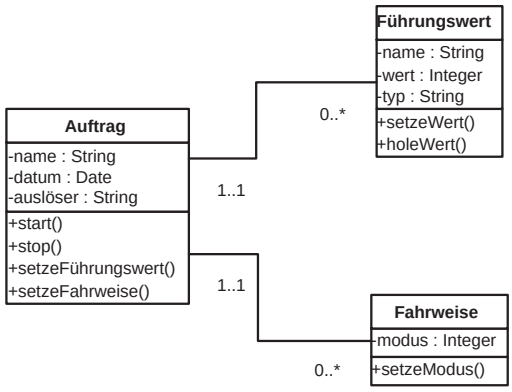


Abbildung 8.3: Das Klassendiagramm des Auftrag/Fahrweise/Führungswert-Musters

8.4.2 Teilanlagensteuerungen/Anlagensteuerungs-Muster

Dieser Abschnitt behandelt das Teilanlagensteuerung/Anlagensteuerungs-Muster.

Kontext: Anlagensteuerungen unterschiedlicher, technischer Ebenen arbeiten ebenenübergreifend zusammen.

Problem: Die Steuerung einer Anlage steht mit den ihr zugeordneten Teilanlagensteuerungen in Beziehung, um die geforderte Funktionalität bereitzustellen. Teilanlagensteuerungen werden von ihren Anlagensteuerungen mit Führungswerten und Fahrweisen versorgt, während Teilanlagensteuerungen Informationen bzgl. ihres Bearbeitungszustands an die übergeordnete Teilanlagensteuerung liefern. Zustandsinformationen können entsprechend für Teilanlagensteuerungen aufgearbeitet werden. Auch Teilanlagensteuerungen erben die Eigenschaften von Funktionsbausteinen.

Lösung: Die Lösung hat eine große Ähnlichkeit mit dem Gruppensteuerungen/Teilanlagensteuerungs-Muster und wird daher hier nicht weiter betrachtet (s. Abbildung 8.4).

Beispiel: Bei der Beispielanlage, die in Abbildung 2.4 gezeigt wird, ist das Zusammenwirken der Anlagensteuerung zur Herstellung von Gummibärchen mit der Teilanlagensteuerung für die Extraktion der Gelatine ein Beispiel zur Anwendung des Teilanlagensteuerungen/Anlagensteuerungs-Muster.

Überlappung: Diese besteht wegen gemeinsamer Teilanlagensteuerungen zum Gruppensteuerungen/Teilanlagensteuerungs-Muster. Da dieses Muster wichtige Informationen bzgl. der Verarbeitung in einer Anlage bereitstellen kann, besteht eine Überlappung zum Bild/Funktionsbaustein-Muster.

8.4.3 Gruppensteuerungen/Teilanlagensteuerungs-Muster

In diesem Abschnitt wird das Gruppensteuerungen/Teilanlagensteuerungs-Muster vorgestellt.

Kontext: Gruppensteuerungen arbeiten ebenenübergreifend zusammen. Zwischen Gruppen- und Teilanlagensteuerungen werden Informationen ausgetauscht.

Problem: Die Steuerung einer Teilanlage steht mit den ihr zugeordneten Gruppensteuerungen in Beziehung, um die geforderte Funktionalität bereitzustellen. Dazu werden Gruppensteuerungen von ihren Teilanlagensteuerungen mit Führungswerten und Fahrweisen versorgt, während Gruppensteuerungen Informationen bzgl. ihres Bearbeitungszustands an die übergeordnete Teilanlagensteuerung liefern. Zustandsinformationen können von den Gruppensteuerungen für die Teilanlagensteuerungen bereitgestellt werden.

Lösung: Funktionsbausteine und Gruppensteuerungen besitzen den gleichen zuvor beim Einzelgerätesteuerungen/Gruppensteuerungs-Muster beschriebenen Aufbau und die gleichen Beziehungen. Klassen für Teilanlagensteuerungen sind Unterklassen der abstrakten Klasse Funktionsbaustein. Auch Klassen für Teilanlagensteuerungen verfügen über Schwellwerte und Attribute sowie über entsprechende Operationen, um auf diese Attribute zuzugreifen (s. Abbildung 8.5). Zwischen der Klasse Teilanlagensteuerung und der Klasse Gruppensteuerung wird eine Assoziation mit dem Namen *istUntergeordnet* modelliert. Die Multiplizitäten dieser Assoziation erlauben, dass beliebig viele Gruppensteuerungen mit einer Teilanlagensteuerung in Beziehung stehen. Gruppensteuerungen

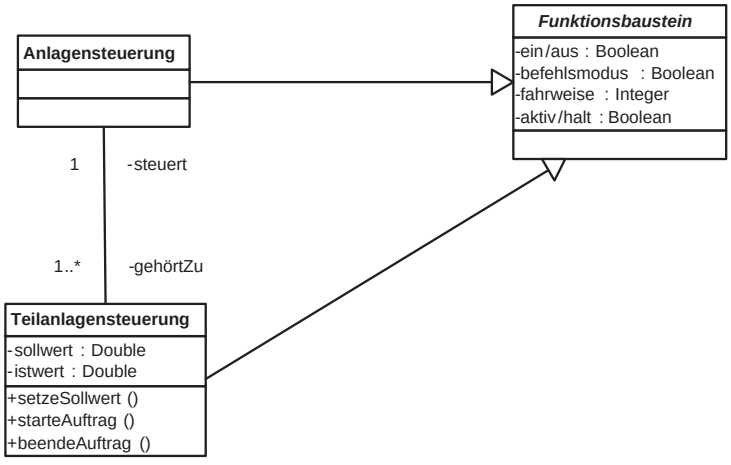


Abbildung 8.4: Das Teilanlagensteuerungen/Anlagensteuerungs-Muster

nehmen an dieser Assoziation in der Rolle *istUntergeordnet* und Teilanlagensteuerungen in der Rolle *führt* teil.

Beispiel: Die Beispielanlage in Abbildung 2.4 enthält eine Teilanlagensteuerung für die Extraktion der Gelatine, die mit der Gruppensteuerung zum Abpumpen zusammenarbeitet. Dabei tauschen Teilanlagensteuerung und Gruppensteuerung Informationen untereinander aus. Dies ist ein Beispiel für die Anwendung des Gruppensteuerungen/Teilanlagensteuerungs-Musters.

Überlappung: Bei einer gemeinsamen Gruppensteuerung existiert eine Überlappung mit dem Einzelgerätesteuerungen/Gruppensteuerungs-Muster. Da dieses Muster wichtige Informationen bzgl. des Zustandes der Verarbeitung in einer Anlage bereitstellen kann, besteht hier auch eine Überlappung zum Bild/Funktionsbaustein-Muster.

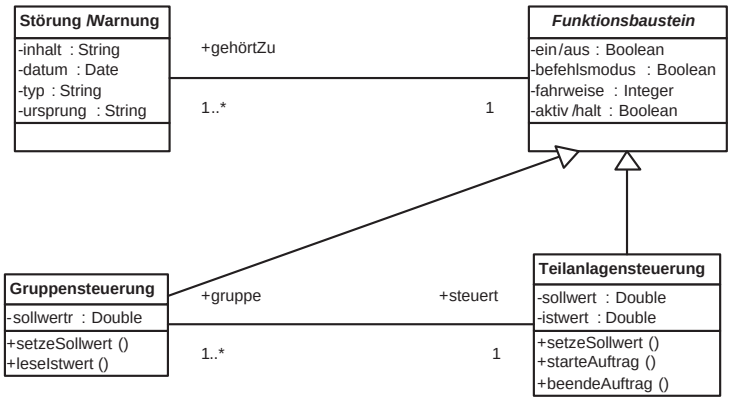


Abbildung 8.5: Das Gruppensteuerungen/Teilanlagensteuerungs-Muster

8.4.4 Einzelgerätesteuerungen/Gruppensteuerungs-Muster

Dieser Abschnitt behandelt das Einzelgerätesteuerungen/Gruppensteuerungs-Muster.

Kontext: Anlagensteuerungen unterschiedlicher technischer Ebenen arbeiten ebenenübergreifend zusammen. Zwischen Feldebene und Prozesssicherungs- bzw. Prozesssteuerungsebene werden Informationen ausgetauscht.

Problem: Jede Einzelgerätesteuerung kann mit Führungswerten durch die ihr zugeordnete Gruppensteuerung versorgt werden. Sensoren liefern insbesondere Messergebnisse zur Verarbeitung an die ihnen übergeordneten Gruppensteuerungen und ist es möglich, dass eine Gruppensteuerung Messergebnisse bei den ihr untergeordneten Sensoren erfragt. Diese Messergebnisse werden in den Gruppensteuerungen zur weiteren Verarbeitung verwendet. Die Verarbeitung innerhalb der Gruppensteuerung wird durch ihre Fahrweisen vorgegeben. Als Ergebnis der Verarbeitung werden neue Zustände von den Gruppensteuerungen eingenommen und es wird auf die Einstellung von Aktoren, die der Gruppensteuerung untergeordnet sind, eingewirkt. Es ist selbstverständlich möglich, dass Einzelgeräte Störungen bzw. Fehler an die ihnen übergeordneten Gruppensteuerungen melden.

Lösung: Das Muster ist im Klassendiagramm in Abbildung 8.6 dargestellt. Einzelgeräte- und Gruppensteuerungen sind Instanzen der abstrakten Klasse Funktionsbaustein. Die Klasse *Funktionsbaustein* besitzt die folgenden Attribute:

- *Ein/Aus*, um den Betriebszustand eines Funktionsbausteins zu modellieren.
- *Befehlsmodus*, um den Modus für die Eingabe von Befehlen an einen Funktionsbaustein zu spezifizieren.
- *Fahrweise*, das eine Referenz auf die aktuell angewählte Fahrweise darstellt.
- *Aktiv/Halt* gibt an, ob der Funktionsbaustein aktiviert ist oder angehalten wurde.

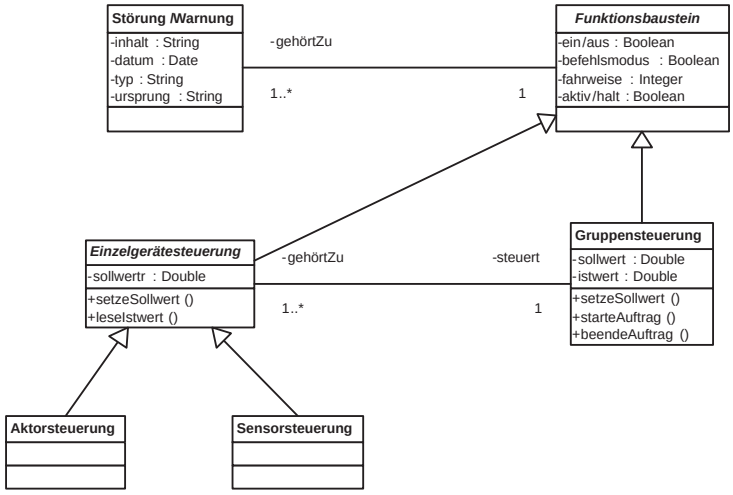


Abbildung 8.6: Das Einzelgerätesteuerungen/Gruppensteuerungs-Muster

Weiterhin besitzt die Klasse *Funktionsbaustein* die Operation *meldeStörung/Warnung*, durch die Störungen bzw. Warnungen an einen Funktionsbaustein gemeldet werden können. Einzelgerätesteuerungen und Gruppensteuerungen sind Unterklassen der Klasse Funktionsbaustein. Einzelgerätesteuerungen sind wiederum abstrakte Klassen, während Gruppensteuerungen bereits konkrete Klassen sind. Gruppensteuerungen besitzen Schwellwerte und Zustandsvariablen als Attribute sowie Operationen, um auf diese Schwellwerte und Zustandsvariablen lesend zuzugreifen, bzw. um sie mit Werten zu schreiben.

Einzelgerätesteuerungen sind ähnlich aufgebaut mit der Ausnahme, dass sie in dieser Arbeit keine Schwellwerte besitzen sollen. Abstrakte Klassen für Einzelgerätesteuerungen sind Oberklassen von Aktorsteuerungen und Sensorsteuerungen, die konkrete Klassen sind. Zwischen der Klasse *Einzelgerätesteuerung* und der Klasse *Gruppensteuerung* wird eine bidirektionale Assoziationen mit dem Namen *SteuerungFeld* modelliert. Einzelgerätesteuerungen besitzen im Rahmen dieser Assoziation die Rolle *istUntergeordnet*, während Objekte der Klasse Gruppensteuerung in der Rolle *steuert* an der Assoziation teilnehmen.

Beispiel: In der Beispielanlage aus Abbildung 2.4 ist die Gruppensteuerung, die für das Abpumpen des Gelatine-Gels verwendet wird und mit der Pumpensteuerung zusammenwirkt, ein Beispiel für die Anwendung des Einzelgerätesteuerungen/Gruppensteuerungs-Musters. Die Gruppensteuerung für das Abpumpen arbeitet sehr eng mit ihren Einzelgeräten zusammen. Auch in werden [MLD09] werden Lüftersteuerungen zu Einzelgerätesteuerungen gruppiert. Im Informationsmodell der IEC 61131-3 [Tec09] werden abstrakte Funktionsbausteine als sog. *CtrlFunctionBlocks* verwendet, die jeweils eine spezielle Funktionalität bereitstellen. Ein *CtrlFunctionBlock* wird ebenfalls als eine abstrakte Klasse modelliert.

Überlappung: Da Verriegelungen häufig auf der Ebene von Einzelgeräte- bzw. Gruppensteuerungen arbeiten, sind Überlappungen mit dem Funktionsbaustein/Verriegelungs-Muster möglich. Die Voraussetzung hierfür ist die Existenz von sinnvollen Verriegelungsbedingungen.

8.4.5 Funktionsbaustein/Verriegelungs-Muster

In diesem Abschnitt wird das Funktionsbaustein/Verriegelungs-Muster vorgestellt.

Kontext: Um die Sicherheit von Anlagen zu gewährleisten, müssen bestimmte Vorkehrungen getroffen werden, bei Gefährdungen bestimmte Anlagenteile außer Betrieb zu nehmen.

Problem: Für Funktionsbausteine werden durch Prozesse Situationen festgelegt, in denen Sicherheitsbedingungen verletzt werden, z. B. beim Erreichen bzw. Überschreiten von Schwell- bzw. Verrieglungswerten. Verriegelungen können unter diesen Bedingungen Abschaltungen durchführen.

Lösung: Jede Instanz der abstrakten Klasse *Funktionsbaustein* kann mit Verriegelungsobjekten

in Beziehung stehen. Verriegelungen werden jedoch häufig in der Feldebene und in der Prozessstabilisierungsebene eingesetzt, sodass bei diesem Muster Einzelgerätesteuerungen und Gruppensteuerungen Unterklassen der abstrakten Klasse Funktionsbaustein sind. Verriegelungsobjekte werden durch Instanzen von Einzelgerätesteuerungen über Veränderungen von Werten informiert und überprüfen durch Operationen, ob durch Verletzung von Schwellwerten, die in Attributen der Assoziationsklasse *Schwellwert* festgehalten werden, Verriegelungsbedingungen eingetreten sind. Dadurch können Verriegelungsaktionen ausgelöst werden, die die Abschaltung bzw. Deaktivierung von technischen Funktionseinheiten vornehmen und diese in einen sicheren Zustand überführen, indem sie den Zustand von Funktionsbausteinen (z. B. Gruppensteuerungen) beeinflussen.

Beispiel: In der Beispielanlage aus Abbildung 2.4 wird die Steuerung für die Membranpumpe verriegelt, falls die Membranpumpe aus einem gefährlichen Zustand, wie etwa plötzliche Abkühlung, angefahren wird.

Überlappung: Wie in Abschnitt 8.4.4 beschrieben, können diese mit dem Einzelgerätesteuerungen/Gruppensteuerungs-Muster auftreten.

8.4.6 Reglermuster

In diesem Abschnitt erfolgt die Vorstellung des Regler-Musters.

Kontext: In technischen Systemen wird die stabile Einhaltung bestimmter Zustände benötigt, um Stabilität, Sicherheit und Effektivität einer Anlage zu gewährleisten.

Problem: Istwerte eines technischen Systems müssen mit vorgegebenen Sollwerten verglichen und korrigiert werden, um bestimmte Zustände einzuhalten. Zur Aufrechterhaltung der Stabilität bzw. von bestimmten voreingestellten Sollzuständen in einem technischen System werden Regler verwendet. Es gibt unterschiedliche Regelstrategien, um die Regeldifferenz zwischen Soll- und Istwert möglichst klein zu halten. Die bekannteste ist der bewährte PID-Regler, der sich an der klassischen Regelungstheorie orientiert. In den letzten Jahren sind aber auch verstärkt Regler, die Fuzzy-Logik verwenden, zur Anwendung gekommen.

Lösung: Zwischen Reglerobjekten, die Instanzen der abstrakten Klasse *Funktionsbaustein* sind, und Einzelgerätesteuerungen, die für Sensoren und Aktoren verwendet werden, gibt es entsprechende Assoziationen.

Beispiel: Der Regler, der in der Beispielanlage aus Abbildung 2.4 zur Stabilisierung der Temperatur für die Extraktion verwendet wird, zeigt beispielhaft die Verwendung des Regler-Musters. Ein weiteres Beispiel wird in [MLD09] durch eine Regelung für einen Lüfter abgegeben.

Überlappung: keine

8.4.7 Bild/Funktionsbaustein-Muster

In diesem Abschnitt wird das Bild/Funktionsbaustein-Muster behandelt.

Kontext: In technischen Anlagen muss der Zustand von einzelnen Komponenten visualisiert werden.

Problem: In Leitwarten und Steuerständen wird der Bearbeitungszustand von Teilanlagensteuerungen, Anlagensteuerungen und auch Gruppensteuerungen durch Bilder visualisiert. Die Visualisierung umfasst neben dem aktuellen Bearbeitungszustand die Anzeige von aufgetretenen Störungen bzw. Warnungen. Diese Informationen werden von Anlagensteuerern zur Führung der Anlage verwendet.

Lösung: Das Klassendiagramm in Abbildung 8.7 skizziert die Lösung. Die Klasse *Bild* dient der Anzeige von Warnungen/Störungen. Die Klasse Bearbeitungszustand gibt Informationen zum gegenwärtigen Bearbeitungszustand eines Funktionsbausteines. Beim Betrieb von Funktionsbausteinen – speziell von Anlagen und Teilanlagensteuerungen – können Störungen bzw. Warnungen, die den Bearbeitungszustand betreffen, auftreten. Sollten beim Betrieb von Instanzen der Klasse *Funktionsbaustein* Störungen bzw. Warnungen erzeugt werden, so werden diese mit ihren zugehörigen Bildern und Funktionsbausteinen assoziiert. Dies ermöglicht die Anzeige von Störungs-/Warnungsinhalten im zugehörigen Bild des Anlagenbestandteils. Ein Bild visualisiert also den Bearbeitungszustand eines Funktionsbaustein, wie etwa speziell einer Teilanalagensteuerung.

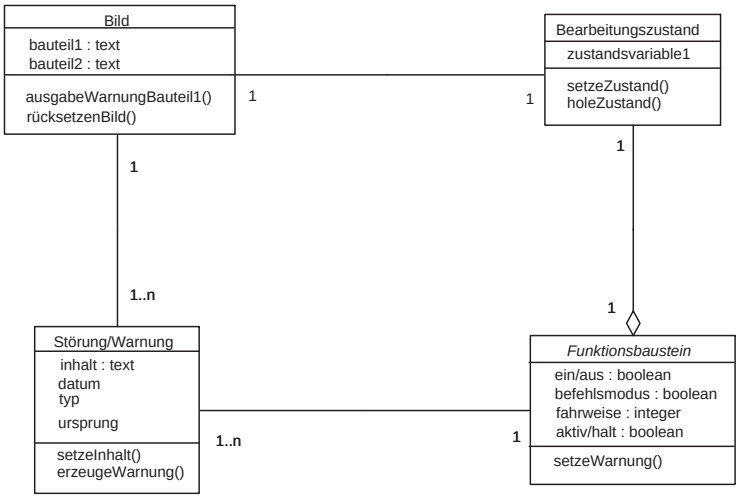


Abbildung 8.7: Das Bild/Funktionsbaustein-Muster

Im Allgemeinen identifiziert Jaffe [JLHM91, Lev95] drei Fragen in der Anforderungsspezifikation für jeden Datenbestandteil eines Bildes, das für einen Menschen angezeigt werden soll:

- Welches Ereignis verursacht den Eintrag eines Items?
- Kann oder sollte die Anzeige jemals aktualisiert werden, sobald der Datenbestandteil angezeigt worden ist? Wenn dies der Fall ist, welche Ereignisse sollten diese Aktualisierung verursachen? Ereignisse, die Updates auslösen, können sein: Externe Beobachtungen, Verstreichen von Zeit, Aktionen, die durch den beobachtenden Operator vorgenommen und Aktionen, die durch andere Operatoren ausgelöst werden.
- Welche Vorgänge führen zum Verschwinden eines Ereignisses? Zusätzlich zu den Daten kann der Computer die Labels (sowie Menüs oder Software gestützte Labels oder Knöpfe), die mit den Operatoraktionen assoziiert werden, kontrollieren.

Beispiel: In der Beispielanlage aus Abbildung 2.4 kann der Anlagenbediener die Temperatur und andere physikalische Größen, die bei der Extraktion der Gelatine von Bedeutung sind, beobachten. Dies ist ein Beispiel für die Verwendung des Bild/Funktionsbaustein-Musters.

Überlappung: Diese bestehen zu Analysemustern, die geeignete Informationen für Anlagenbediener bereitstellen können. Hierfür kommen das Gruppensteuerungen/Teilanlagensteuerungs-Muster und das Teilanlagensteuerungen/Anlagensteuerungs-Muster in Betracht.

8.4.8 Auftrag/Befehl/Funktionsbaustein-Muster

Wenn die in Abbildung 2.4 gezeigte Anlagensteuerung einen Auftrag bzw. einen Befehl an die ihr untergeordnete Teilanlagensteuerung zur Extraktion absetzt, findet das Auftrag/Befehl/Funktionsbaustein-Muster Verwendung.

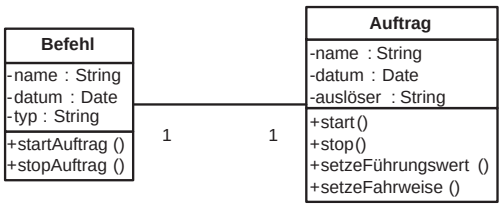


Abbildung 8.8: Das Auftrag/Befehl/Funktionsbaustein-Muster

Kontext: Miteinander in Verbindung stehende Funktionsbausteine tauschen Befehle untereinander aus.

Problem: Übergeordnete Funktionseinheiten oder Anlagenbediener verursachen durch Befehle die Abarbeitung von Aufträgen in Funktionsbausteinen. Befehle haben einen Namen, einen Typ, ein Startdatum und einen Ziel-Funktionsbaustein. Befehle werden in Aufträge umgewandelt, die anschließend an den Zielfunktionsbaustein weitergereicht werden.

Lösung Es werden zwei Klassen für Befehle und Aufträge eingeführt (s. Abbildung 8.8). Diese beiden Klassen werden durch eine Assoziation verbunden. Die Klasse *Befehl* besitzt die oben beschriebenen Attribute. Die Klasse *Auftrag* übernimmt die Attributwerte der Klasse *Befehl* und erhält im Attribut Auslöser zusätzlich eine Referenz auf den Befehl.

Beispiel: Ein Beispiel für einen Befehl mit einem Auftrag ist etwa ein Aufruf zum Starten des Extraktionsvorgangs zur Gelatine Herstellung.

Überlappung: keine

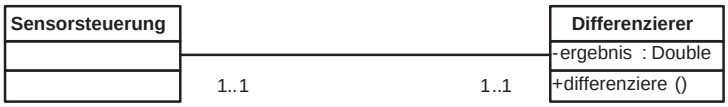


Abbildung 8.9: Das Sensor/Differenzierer-Muster

8.4.9 Sensor/Differenzierer-Muster

Der Abschnitt dient der Vorstellung des *Sensor-Differenzierer-Musters*.

Kontext: Signale müssen regelungstechnisch verarbeitet werden.

Problem: Bestimmte eingehende Signale müssen differenziert werden. In der Regelungstechnik wird hierfür das so genannte D-Glied [Unb92] verwendet. Ein D-Glied ermittelt als Ausgangssignal die Steigung eines Eingangssignals. Häufig wird das Ausgangssignal um einen bestimmten Faktor verstärkt; dann spricht man vom PD-Glied. Praktische Problemstellungen hierfür sind Filter, die den Anstieg eines Signals erkennen müssen.

Lösung Es wird eine Klasse *Differenzierer* (s. Abbildung 8.9) eingeführt, die eine Methode enthält, die als Argument das Eingangssignal entgegennimmt und als Rückgabewert das Ausgangssignal liefert. Die Datentypen des Eingangs- und des Ausgangssignals sind reellwertig.

Beispiel: In der Beispielanlage existiert keine sinnvolle Verwendung für dieses Analysemuster.

Überlappung: keine

8.4.10 Sensor/Integrierer-Muster

In diesem Abschnitt wird das *Sensor-Integrierer-Muster* vorgestellt.

Kontext: Signale eines Sensors müssen regelungstechnisch verarbeitet werden.

Problem: Bestimmte eingehende Signale eines Sensors müssen integriert werden. Hierfür wird in der Regelungstechnik das sog. Integral bzw. I-Glied [Unb92] bereitgestellt. Auch hier beim I-Glied wird häufig eine Verstärkung um einen proportionalen Faktor vorgenommen. Ein praktisches Problem hierfür ist die Summierung eines Eingangssignals über einen bestimmten Zeitraum.

Lösung Eine Klasse *Integrierer* (s. Abbildung 8.10) wird eingeführt, die eine Methode enthält, die als Argument das Eingangssignal entgegennimmt und als Rückgabewert das Ausgangssignal liefert. Die Datentypen des Eingangs- und des Ausgangssignals sind Real.

Beispiel: In der Beispielanlage gibt es keine sinnvolle Verwendung für dieses Analysemuster.

Überlappung: keine

8.5 Verhaltensmuster für Analysemuster

Bei der Erforschung von Mustern standen bisher strukturelle Beziehungen zwischen einzelnen Klassen im Vordergrund. Einige Wissenschaftler [BCS00, Dou99, Fow99] haben bisher auf Verhaltens-

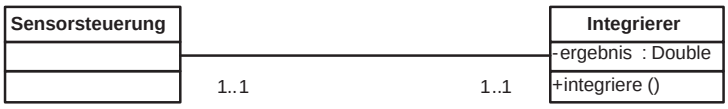


Abbildung 8.10: Das Integrierer-Muster

templates bzw. auf die Existenz von wiederkehrenden Strukturen in Bezug auf Verhalten bei objektorientierter Software hingewiesen. Verhaltensmuster beschreiben wiederverwendbare Schablonen für das Verhalten von Klassen. Bei der Analyse von Steuerungssoftware für die chemietechnische Beispielanlage aus Abschnitt 2.5 werden die in diesem Abschnitt aufgeführten Verhaltensmuster, die typisch für das Verhalten von Steuerungssoftware sind, vorgestellt. Ähnlich wie bei OPC-UA-Programmen [MLD09] werden Statechart-Diagramme zur Modellierung verwendet.

8.5.1 Verhalten des abstrakten Funktionsbausteins

In diesem Abschnitt wird ein Verhaltensmuster für die abstrakte Klasse Funktionsbaustein vorgestellt. Funktionsbausteine nehmen in Abhängigkeit zum aktuellen Bearbeitungszustand bestimmte Zustände ein. Diese ergeben sich aus dem Funktionsbaustein-Konzept und müssen bei Bedarf in Anwendungen dieses Konzeptes entsprechend verfeinert werden. Das Muster besteht, wie in Abbildung 8.11 durch ein Statechart-Diagramm gezeigt, aus einem Hauptzustand, in dem jede Verarbeitung eines Funktionsbausteines festgehalten wird. Im Folgenden soll jeder Zustand kurz erklärt werden. Der Zustand *Bereit* wird eingenommen, sobald ein Funktionbaustein fertig konfiguriert und betriebsbereit ist. Der Zustand *Laufend* wird eingenommen, wenn das Ereignis *starteVerarbeitung* auftritt. In diesem Zustand findet die normale Verarbeitung des Funktionsbausteines in der gerade gewählten Fahrweise statt. Deswegen kann dieser Zustand beliebig um weitere Unterzustände verfeinert werden. Typischerweise besteht dieser Unterzustand selbst aus den zwei Unterzuständen *Hand* und *Automatik* zwischen denen durch das Ereignis *um* gewechselt werden kann. Im Zustand Automatik wird die gegenwärtig ausgewählte Fahrweise automatisch abgefahren, während sie im Zustand *Hand* per manueller Steuerung abläuft. Weiterhin kann es beim Betrieb eines Funktionsbausteines sein, dass bestimmte Störungssituationen auftreten, wie Fehlermeldungen oder Alarme. In diesem Fall wird aus dem Zustand *Laufend* in den Zustand *Abgebrochen* übergegangen, sobald das Ereignis *abbrechen* auftritt. In diesem Zustand kann eine weitere Fehlerbehandlung vorgenommen werden. Sobald diese abgeschlossen ist, kann mit dem Ereignis *fortsetzen* die ursprüngliche Verarbeitung fortgeführt werden. Natürlich ist es aufgrund beliebiger Umstände, wie z. B. Gefahren oder anderer Beeinträchtigungen des Betriebsablaufs, manchmal notwendig die Verarbeitung des Betriebsablaufs im Funktionsbaustein anzuhalten. Tritt das Ereignis *anhalten* im Bearbeitungszustand *Laufend* ein, so wird dieser Zustand betreten. Im günstigsten Falle wird die Bearbeitung eines Funktionsbausteins durch nichts, also weder externe noch interne Einflüsse, beeinträchtigt. In diesem Fall wird nach der Fertigstellung der Verarbeitung durch das Ereignis *beenden* in den Zustand *Fertig* übergegangen, wonach die Zustandsmaschine auch automatisch terminiert wird. In der Beispielanlage zur Herstellung von Gelatine aus Abschnitt 2.5 sind die Anlagen-, die Teilanlagen- und die Gruppensteuerungen Funktionsbausteine, die eine gewisse Komplexität aufweisen. Das Verhaltensmuster aus Abbildung 8.11 ist daher in jeder dieser Steuerungen anwendbar. Natürlich müssen für jede Steuerung entsprechende Anpassungen vorgenommen werden.

Dieses abstrakte Verhaltensmuster ist bei allen Anlagen-, Teilanlagen-, Gruppen- und Einzelgerätesteuerungen anwendbar. Speziell die Teilanlagensteuerung der Teilanlage zur Extraktion kann sowohl im Hand- bzw. Automatikbetrieb arbeiten. Natürlich kann der Ablauf der Teilanlagensteuerung angehalten, durch Fehler abgebrochen und erfolgreich beendet (im Sinne einer korrekten Fertigstellung) werden.

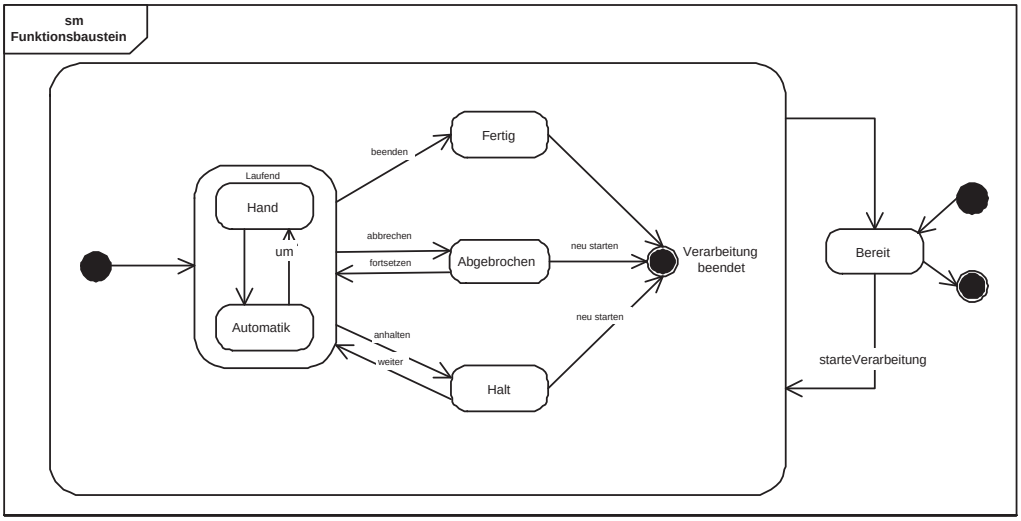


Abbildung 8.11: Das Verhaltensmuster der Klasse Funktionsbaustein

8.5.2 Zustandsmuster der zyklischen Fahrweise

Für die Unterzustände der Zustände *Laufend.Automatik* bzw. *Laufend.Hand* können beliebige Fahrweisen gewählt werden. Einige typische Fahrweisen werden in den folgenden Abschnitten kurz vorgestellt. Bei der zyklischen Fahrweise werden bestimmte Teilfahrweisen bzw. Teilabläufe in zyklischen Wiederholungen wieder angestoßen. Im Folgenden werden die Unterzustände des Haupzustandes *zyklische Fahrweise* erläutert. Vom Initialzustand wird in den Unterzustand *Fahrweise 1* übergegangen. Manchmal ist es notwendig explizite Initialiserungszustände anzugeben. In Abbildung 8.12 werden die Unterzustände *Fahrweise 1*,..., *Fahrweise M* jeweils beim Auftreten des Ereignisses *weiter* eingenommen. Befindet man sich im Zustand *Fahrweise M* und das Ereignis *weiter* tritt auf, wird der Zyklus ein weiteres Mal durchlaufen. In den Unterzuständen können weitere Aktivitäten bzw. Aktionen modelliert werden. Für den Fall, dass es notwendig ist, explizite Terminierungszustände anzugeben, kann der Zyklus an bestimmten Stellen verlassen werden. Hier kann das in einem Unterzustand geschehen.

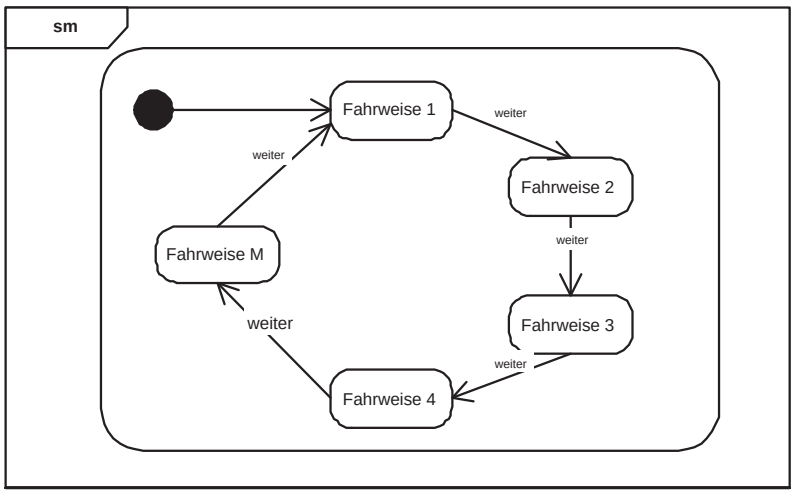


Abbildung 8.12: Das Verhaltensmuster der Klasse zyklische Fahrweise

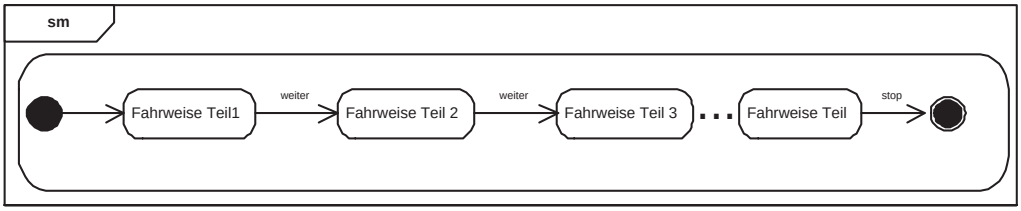


Abbildung 8.13: Eine sequentielle Fahrweise

8.5.3 Zustandsmuster der sequentiellen Fahrweise

Eine weitere Möglichkeit die Teilschritte einer Fahrweise anzugeben, besteht darin diese sequentiell anzuordnen. Auch diese soll hier kurz beschrieben werden. Die Abbildung 8.13 gibt ein Statechart-Diagramm des Musters an. Der Initialzustand gibt den Anfangszustand einer Verarbeitung vor. Die Teilzustände *Fahrweise Teil 1* bis *Fahrweise Teil M* geben die einzelnen Teile einer Fahrweise an, die sequentiell durchlaufen werden. Die Teilzustände können dabei entweder aus einzelnen Schritten oder wiederum aus Teilfahrweisen bestehen. Zustandsübergänge finden immer statt, wenn das Ereignis *weiter* auftritt. Der Endzustand gibt die Beendigung einer Verarbeitung an. Er wird betreten, wenn der letzte Teil der Fahrweise bearbeitet worden ist und das Ereignis *stop* eintritt. Ein Beispiel für die Anwendung dieses Zustandsmuster ist sie Teilanlagensteuerung zur Extraktion der Gelatine aus Abschnitt 2.5. Die Verfahrensbestandteile des Befüllens des Behälters, des Erhitzens mit Temperaturregelung und des Abpumpens müssen in sequentieller Reihenfolge mit geeigneten Parametern ausgeführt werden. Zur Beschreibung des Verhaltens der Teilanlagensteuerung ist das Zustandsmuster der sequentiellen Fahrweise geeignet, da jeder Verfahrensbestandteil einer Teilfahrweise entspricht.

In der Beispielanlage aus Abschnitt 2.5 werden von der Teilanlagensteuerung zur Extraktion sequentiell die Verfahrensschritte des Befüllens des Behälters, dem Erhitzen der Lösung und dem Abpumpen durchlaufen. Die Teilanlagensteuerung zur Extraktion ist somit ein Beispiel zur sequentiellen Fahrweise.

8.5.4 Zustandsmuster der explizit angestoßenen Fahrweise

Weiterhin kann ein Funktionsbaustein in der Lage sein alternativ, eine von mehreren Fahrweisen zu bearbeiten. Dazu muss es möglich sein eine Fahrweise explizit auszuwählen bzw. anzustoßen.

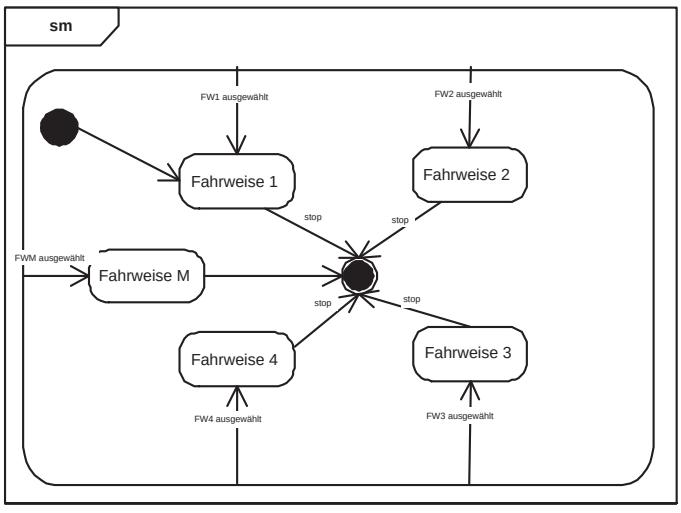


Abbildung 8.14: Eine explizit angestoßene Fahrweise

Bei einem solchen Muster, das in Abbildung 8.14 durch ein Statechart-Diagramm gezeigt ist, liegen Transitionen aus dem Hauptzustand auf die einzelnen Unterzustände vor, die zur Auswahl mit entsprechenden Ereignissen versorgt werden müssen. Der Initialzustand wird am Anfang einer Verarbeitung eingenommen. Die Teilzustände *Fahrweise 1* bis *Fahrweise M* geben alternative Fahrweisen an, die entsprechend ausgewählt werden müssen. Zustandsübergänge finden immer statt, wenn das Ereignis *weiter* auftritt. Der Endzustand gibt die Beendigung einer Verarbeitung an. Er wird betreten, wenn die ausgewählte Fahrweise bearbeitet worden ist und das Ereignis *stop* eintritt.

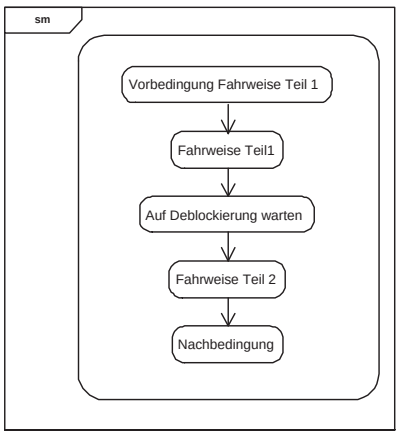


Abbildung 8.15: Das Warten-Zustandsmuster

8.5.5 Warten-Zustandsmuster

Die einzelnen Schritte von Fahrweisen bestehen typischerweise auch aus bestimmten Zustandsmustern. Interessant sind hier das Warten-Zustandsmuster, das in diesem Abschnitt beschrieben wird. Beim Warten-Muster wird ein Wartezustand aus einer Teilfahrweise heraus betreten, der verlassen wird, sobald ein bestimmtes Ergebnis abgeliefert wurde. Hier werden die Zustände dieses Zustandsmusters (s. Abbildung 8.15) aufgeführt: Im Zustand *Vorbedingung FW1* werden mögliche Vorbedingungen für den Teil 1 einer Fahrweise geprüft. Sind die Prüfungen abgeschlossen, wird der Teil 1 einer Fahrweise durchgeführt. Wenn notwendig, werden für die Bearbeitung externe Quellen/Komponenten aufgerufen, die bestimmte Resultate zurückliefern sollen. Werden externe Ergebnisse erwartet, so führt das zur Einnahme eines Blockierungszustandes, in dem gewartet wird. Dieser kann anschließend verlassen werden, um den Teil 2 einer Fahrweise zu bearbeiten. Ist dies geschehen wird eine Nachbedingung geprüft.

Kapitel 9

Musterverfeinerung

In diesem Kapitel wird gezeigt, wie Analysemuster des Analysemuster-Systems beim Entwurf durch Entwurfsmuster aus dem Entwurfsmuster-System verfeinert werden können. Die Analysemuster im ASM spezifizieren funktionale Anforderungen auf abstraktem Niveau. Dies führt zu Musterkombinationen im KSM, die sich aus Analysemustern sowie zur korrekten Verfeinerung verwendeten Entwurfsmustern zusammensetzen. Ein Paar, das durch ein Analysemuster sowie eine solche Entwurfsmuster-Kombination – also einer Menge von Entwurfsmustern – gebildet wird, soll im Folgenden als Verfeinerungsmuster bezeichnet werden. Bei der Anwendung von Verfeinerungsmustern zur Entwicklung von Steuerungssoftware werden bestimmte Ziele verfolgt:

- Die Entwicklung von Steuerungssoftware soll sich an den für diese Domäne gültigen Abstraktionen orientieren und diese einem Entwickler von Steuerungssoftware leicht anwendbar bereitstellen.
- Die Organisation der gemeinsamen Umsetzung von funktionalen und nicht-funktionalen Anforderungen soll unterstützt werden.
- Der korrekheitsgesicherte Entwurf wird unterstützt, indem für jedes Analysemuster eine geeignete Entwurfsmuster-Kombination bereitgestellt wird. Die Anforderungen und bestimmte Annahmen, die für die Analysemuster formuliert werden, müssen durch die Spezifikationen der verwendeten Entwurfsmuster erfüllt werden.
- Es sollen wiederverwendbare Beziehungen zwischen Mustern, die sich auf unterschiedlichen Abstraktionsniveaus befinden, dokumentiert werden.

Zunächst wird der Begriff des Verfeinerungsmusters definiert, der in [Gra99b] eingeführt wurde. Mittlerweile haben auch andere Autoren wie N. Shankar [Sha03], der Verfeinerungsmuster als mittelfristiges Forschungsziel nennt, und C. Pons [Pon06] den Begriff verwendet. Weiterhin werden nicht-funktionale Anforderungen an Analysemuster untersucht und danach sinnvolle Verfeinerungsmuster für die Domäne der Steuerungssoftware vorgestellt. Abschließend wird auf die korrekte Anwendung von Verfeinerungsmustern bei der Softwareentwicklung durch Beschreibung eines Prozessmodells eingegangen.

9.1 Verfeinerungsmuster

In diesem Abschnitt soll der Begriff des Verfeinerungsmusters genauer definiert werden. Um zu einer systematischen Aufzählung von Verfeinerungsmustern zu gelangen, wird danach für jedes Analysemuster des Analysemuster-Systems untersucht, welche konkreten nicht-funktionalen Anforderungen bzgl. Realzeit, Entferntheit und Fehlertoleranz an ein Analysemuster bestehen können, die durch eine Entwurfsmuster-Kombination umgesetzt werden. Abschließend werden in diesem Abschnitt einige Vorschläge für derartige Verfeinerungsmuster in einem Musterkatalog (s. Tabelle 9.1 und 9.2) angegeben, der beschreibt, wie jedes individuelle Verfeinerungsmuster aufgebaut ist. Pons

beschreibt in [Pon06] eine gute Motivation für die Erforschung von Verfeinerungsmustern, nachdem sie die Grundlagen der formalen Verfeinerung vorgestellt hat:

> „This refinement machinery is present in most formal specification languages such as Object-Z [6], [21], B [10], and the refinement calculus [2]. Besides, some restricted forms of programming languages can also be formally refined [4]. But, in the standard specification language UML [14], the refinement machinery has not reached a mature state yet. Being UML a language widely used in software development, any effort made towards increasing the robustness of the UML refinement machinery becomes a valuable task which will also contribute to the improvement of MDD (Model Driven Development). "

Für die Formalisierung von UML-Diagrammen wird in [Pon06] Object-Z verwendet. Der dort beschriebene Ansatz, der von Pons selbst als formal nach informal charakterisiert wird, ist umgekehrt in Bezug auf die Formalisierung zu dem, der in dieser Arbeit verwendet wird, da in dieser Arbeit informale UML Muster auf der Basis von cTLA formalisiert werden. Bei dem Ansatz von Pons werden stattdessen Verfeinerungsmuster aus Object-Z – einer auf objektorientierten Konzepten beruhenden formalen Spezifikationssprache – in sog. UML-Verfeinerungsmuster übertragen. Als Ursache hierfür wird angegeben, dass in UML-Spezifikationen Verfeinerung oft nicht explizit spezifiziert wird, sondern in den Modellen versteckt ist. Dies wird als versteckte Verfeinerung (engl. hidden refinement) bezeichnet. Als Nachteil des Ansatzes von Pons ist allerdings die geringe Verbreitung von Objekt-Z in industriellen Projekten zu nennen. Außerdem erschwert die frühe Formalisierung die Kommunikation mit den Anwendern, die in frühen Projektphasen involviert sind.

Der Begriff des Verfeinerungsmuster wird in [Sha03] wie folgt beschrieben:

> „A halfway house between transformational and posit-and-prove can be envisaged, where certain patterns of model and refinement can be captured and used in the construction of refinements. This is a more pragmatic idea than transformational refinement in that the pattern might not guarantee the correctness of the refinement. Instead M_2 would be constructed from M_1 by application of a pattern and the correctness of the refinement would be proved in the usual posit-and-prove way. Ideally the pattern should provide much of the ancillary properties (e.g., invariants, tactics) required to complete the proof, or at least an indication of what kinds of properties might be needed. The aim of using such patterns is to minimize verification effort when applying refinement. A research goal is to identify such patterns through a range of case studies and supporting the application of the patterns with tools. "

Die Idee dieser Beschreibung wird für die in dieser Arbeit gültige Definition verwendet. Ein Verfeinerungsmuster setzt sich aus mehreren Bestandteilen zusammen:

- Einem Analysemuster, das die funktionalen Anforderungen beschreibt und dessen Klassen Verhaltensmuster besitzen. Dieses Analysemuster entspricht M_1 aus der Definition von Shankar.
- Einer Menge von nicht-funktionalen Anforderungen bzgl. Realzeit, Entferntheit oder Fehlertoleranz an das Analysemuster, die durch UML-Constraints oder UML-Notizen in den Diagrammen des Analysemusters spezifiziert werden.
- Einer verfeinernden Musterkombination, die in Abhängigkeit vom gerade betrachteten Domänenausschnitt das verfeinerte Analysemuster enthalten kann und zusätzlich aus einem oder mehreren Entwurfsmustern zusammengesetzt ist. Diese Musterkombination muss die funktionalen und nicht-funktionalen Anforderungen erfüllen, die an das Analysemuster bestehen. Diese Entwurfsmuster-Kombination entspricht M_2.
- Einer oder mehreren Annahmen, die vorgegebene Sicherheits-, Lebendigkeits- oder Realzeit-Eigenschaften der Entwurfsmuster-Kombination beschreiben und auch in den UML-Diagrammen der Entwurfsmuster-Kombination eines Verfeinerungsmusters angegeben werden.

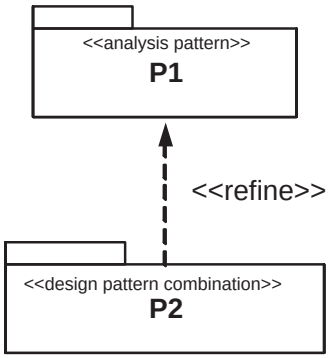


Abbildung 9.1: Die Darstellung eines Verfeinerungsmusters durch Pakete

Als ein Beispiel für Verfeinerungsmuster wird die Verfeinerung des Analysemusters Bild/Funktionsbaustein herangezogen. Es wird angenommen, dass der Funktionsbaustein eine Einzelgerätesteuerung repräsentiert, an den Anforderungen bzgl. der Realzeit und der Fehlertoleranz bestehen. Zur korrekten Verfeinerung wird zur Erfüllung der Realzeit-Anforderungen das Entwurfsmuster der periodischen Task verwendet. Weiterhin wird zur Erfüllung der Anforderungen bzgl. der Fehlertoleranz das Watchdog Muster verwendet. Insgesamt ergibt sich das Verfeinerungsmuster der periodischen, überwachten Einzelgerätesteuerung.

9.2 Verfeinerungsmuster-Katalog

In diesem Abschnitt werden konkrete Vorschläge für Verfeinerungsmuster gemacht und in einem Katalog aufgeführt. Dabei ist zu berücksichtigen, dass durch ein Verfeinerungsmuster häufig Entwurfsprobleme bzgl. unterschiedlicher, nicht-funktionaler Anforderungen gelöst werden müssen, wodurch eine Kombination mehrerer Entwurfsmuster zustande kommt.

Zur Verfeinerung von Klassen werden im OOD nach [Kai99] und [Hawai99] Analyseklassen ersetzt bzw. zusammengefügt. UML stellt leider nur eingeschränkte Mittel zur Verfügung, um Verfeinerungen zwischen Diagrammen zu spezifizieren [Pon06]. So existieren keine standardisierten Konstrukte, um die Semantik einer formalen Verfeinerung zwischen Modellen auf verschiedenen Abstraktionsebenen zu spezifizieren. Speziell gilt dies für die unterschiedlichen Diagrammarten zur Verhaltensspezifikation. Eine Ausnahme bilden Statechart-Diagramme, die es gestatten Zustände aus Klassen zu vererben und damit eine Verfeinerung auszudrücken. Allerdings ist offensichtlich, dass die Verfeinerung unterschiedlicher Diagramme grafisch auch nur kompliziert darzustellen ist. In diesem Abschnitt werden Abhängigkeiten mit dem <<refine>>-Stereotyp verwendet, um Verfeinerungen zwischen dem Klassendiagramm eines Analysemusters und dem Klassendiagramm einer Entwurfsmuster-Kombination darzustellen, wie in Abbildung 9.1 gezeigt. Dazu müssen sich die Klassendiagramme jeweils in einem Paket befinden.

Dabei spezifiziert *P1* ein Paket (Package), das durch das Paket *P2* verfeinert wird. Die Klassendiagramme des Analysemusters und der Entwurfsmuster-Kombination werden jeweils im Rahmen eines Paketes angegeben. Wenn durch das Paket ein Analysemuster spezifiziert wird, erhält es den Stereotyp << analysis pattern>>. Weiterhin wird mit dem Stereotyp << design pattern combination >> ausgedrückt, dass das Paket eine Entwurfsmuster-Kombination enthält. Jetzt werden die folgenden Analysemuster aus *P1* nach Verfeinerungen durch Realzeit-, Verteilungs- und Fehlertoleranzanforderungen untersucht:

- das Reglermuster aus Abschnitt 8.4.6
- das Sensor/Differenzierer-Muster aus Abschnitt 8.4.9
- das Sensor/Integrierer-Muster aus Abschnitt 8.4.10
- das Einzelgerätesteuerungen/Gruppensteuerungs-Muster aus Abschnitt 8.4.4

- das Gruppensteuerungen/Teilanlagensteuerungs-Muster aus Abschnitt 8.4.3
- das Teilanlagensteuerungen/Anlagensteuerungs-Muster aus Abschnitt 8.4.2
- das Gruppensteuerung/Verriegelungs-Muster aus Abschnitt 8.4.5

Zur Präsentation des Verfeinerungsmuster-Katalogs wird jedes so entstehende Verfeinerungsmuster in einer der folgenden Tabellen aufgeführt. Die Tabelle 9.1 enthält Verfeinerungsmuster mit Anforderungen bzgl. der Verteilung und der Realzeit. Im Gegensatz dazu enthält die Tabelle 9.2 Anforderungen bzgl. der Fehlertoleranz und der Realzeit. Für Verfeinerungen, die durch Realzeitanforderungen bedingt sind, wird keine separate Tabelle angegeben. Die Angabe der Verfeinerungsmuster in einer Tabelle erfolgt nach folgendem Schema: Der Name eines Verfeinerungsmusters steht jeweils in der linken Spalte der jeweiligen Tabelle und wird durch Verkettung von beschreibenden Eigenschaften (meistens mit Hilfe von Adjektiven) der verfeinernden Entwurfsmuster gefolgt vom Namen des Analysemuster gebildet, das in der mittleren Spalte steht. In der rechten Spalte werden die einzelnen Muster der Entwurfsmuster-Kombination, die die Entwurfsprobleme lösen, aufgeführt. Die einzelnen Verfeinerungsmuster des Katalogs werden im Folgenden vorgestellt.

Name des Verfeinerungsmusters	Zu verfeinerndes Analysemuster aus dem Analysemuster-System	Zur Verfeinerung verwendete Entwurfsmuster
Verteilter Regler	Regler	Proxys jeweils bei Aktoren und Sensoren, periodische Reglertask
Vermittelte Regelung	Regler	RT-Broker jeweils bei Aktoren, Sensoren und periodischer Reglertask
Verteilte Einzelgerätesteuerungen/Gruppensteuerung	Einzelgerätesteuerungen/ Gruppensteuerung	Proxy, periodische Task
Verteilte Gruppensteuerungen/Teilanlagensteuerung	Gruppensteuerungen/ Teilanlagensteuerung	Proxy, periodische Task
Verteilte Teilanlagensteuerungen/Anlagensteuerung	Teilanlagensteuerungen/ Anlagensteuerung	Proxy, periodische Task
Verteilter bzw. vermittelter Differenzierer mit Einzelgerätesteuerung	Sensor/Differenzierer	Proxy/RT-Broker u. periodische Tasks
Verteilter bzw. vermittelter Integrierer mit Einzelgerätesteuerung	Sensor/Integrierer	Proxy/RT-Broker u. periodische Tasks
Vermittelte Einzelgerätesteuerungen/Gruppensteuerung	Einzelgerätesteuerungen/ Gruppensteuerung	RT-Broker, periodische Task
Vermittelte Gruppensteuerungen/Teilanlagensteuerung	Gruppensteuerungen/ Teilanlagensteuerung	RT-Broker, periodische Task
Vermittelte Teilanlagensteuerungen/Anlagensteuerung	Teilanlagensteuerungen/ Anlagensteuerung	RT-Broker, periodische Task
Periodische, entfernte bzw. vermittelte Beobachtung	Allgemeiner Bild/Funktionsbaustein	Proxy/RT-Broker, periodische Task
Transaktionsgesicherter Bild/ Funktionsbaustein	Bild/Funktionsbaustein	Sporadische Bild-Task mit Transaktionssteuerungsobjekten
Transaktionsgesicherter, verriegelter Bild/Funktionsbaustein	Verriegelter Bild/Funktionsbaustein	Transaktionsmuster, jeweils sporadische Task für Bild und Funktionsbaustein
Periodische Funktionalität	Funktionsbaustein	Periodische Task
Sporadische Funktionalität	Funktionsbaustein	Sporadische Task

Tabelle 9.1: Der Verfeinerungsmuster-Katalog für Realzeit- und Verteilungsanforderungen

Name des Verfeinerungsmusters	Zu verfeinerndes Analysemuster aus dem Analysemuster-System	Zur Verfeinerung verwendete Entwurfsmuster
Periodische, überwachte Einzelgerätesteuerung	Einzelgerätesteuerung/ Bild	Watchdog, periodische Task
Überwachte Gruppensteuerung	Gruppensteuerung/Bild	Watchdog, periodische Task
Periodische, überwachte Gruppensteuerung	Gruppensteuerung/Bild	Watchdog, periodische Task
Sporadische bzw. periodische, überwachte Teilanlagensteuerung	Teilanlagensteuerung/ Bild	Watchdog, periodische oder sporadische Task
Sporadische bzw. periodische, überwachte Anlagensteuerung	Anlagensteuerung/Bild	Watchdog, sporadische bzw. periodische Task
Redundante Einzelgerätesteuerungen/Gruppensteuerung	Einzelgerätesteuerungen/ Gruppensteuerung	Master-Slave Entwurfsmuster und entsprechende Tasks
Redundante Gruppensteuerungen/Teilanlagensteuerung	Gruppensteuerungen/ Teilanlagensteuerung	Master-Slave Entwurfsmuster und entsprechende Tasks
Redundante Teilanlagensteuerungen/Anlagensteuerung	Teilanlagensteuerungen/ Anlagensteuerung	Master-Slave Entwurfsmuster und entsprechende Tasks
Redundante Regelung	Regler	Master-Slave Entwurfsmuster und entsprechende Tasks
Fehlertolerante Gruppensteuerung mit Verriegelung	Gruppensteuerung	periodische Task, Master Slave
Fehlertolerante, verriegelte Einzelgerätesteuerungen/ Gruppensteuerung	Gruppensteuerung/Verriegelung	Periodische Task für GS spontane für Verriegelung

Tabelle 9.2: Der Verfeinerungsmuster-Katalog für Realzeit- und Fehlertoleranzanforderungen

9.2.1 Verfeinerung von Analysemustern mit Realzeitanforderungen

In diesem Abschnitt werden die Verfeinerungen von Analysemustern mit Realzeitanforderungen angegeben. Sie bilden die Basis für spätere Verfeinerungen durch Verteilungs- bzw. Fehlertoleranzanforderungen. Falls beim Entwurf eine Task verwendet wird, wird folgende Konvention für den Typ der Task verwendet:

- Für den Fall, dass eine periodische Task verwendet wird, wird die Abkürzung *pTask* für die Bezeichnung der Klasse verwendet.
- Falls eine sporadische Task zur Verwendung kommt, wird die Klasse mit *sTask* abgekürzt.
- Wenn offen ist, welcher Typ einer Task zur Verwendung kommt, wird die Bezeichnung *Task* verwendet.

Beim Analysemuster des Reglers ist es erforderlich eine periodische Task zur Verfeinerung der Reglerklasse einzusetzen, wie in Abbildung 9.2 gezeigt. Diese liest in regelmäßigen Zeitabständen Istwerte der Sensoren ein und liefert Stellwerte an den Aktor, um die zeitlichen Anforderungen an das Analysemuster zu erfüllen. Generell wird hier festgelegt, dass für eine Reglertask eine periodische Task verwendet wird, um die nicht-funktionalen Anforderungen an einen Regler umzusetzen. Auch die Steuerungen von Aktoren und Sensoren müssen wegen existierender Realzeitanforderungen durch Tasks verfeinert werden. Wie diese Verfeinerung aussieht, hängt stark von den technischen Randbedingungen ab. Es muss die Entwurfsentscheidung getroffen werden, ob eine periodische oder eine sporadische Task verwendet wird.

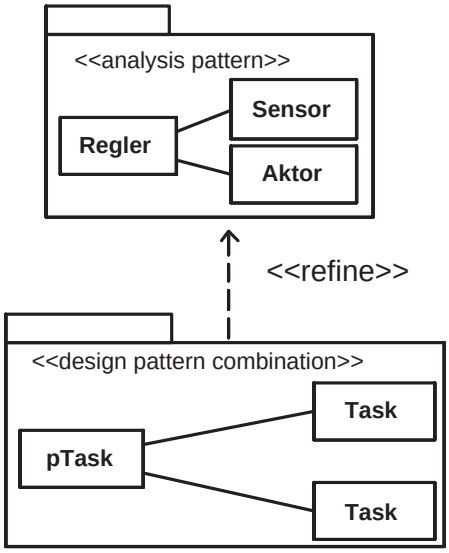


Abbildung 9.2: Die Verfeinerung des Reglers durch Realzeitanforderungen

Für das Analysemuster des Sensor/Differenzierers bzw. des Sensor/Integrierers (siehe Abschnitt 8.4.9 und 8.4.10) ist es erforderlich, dass der Differenzierer bzw. Integrierer durch eine periodische Task verfeinert wird, da entsprechend den Anforderungen eingehende Messwerte in wiederkehrenden Intervallen verarbeitet werden müssen, um die Funktionalität eines Integrierers bzw. Differenzierers zu erbringen. Dies wird in Abbildung 9.3 für beide Analysemuster gezeigt, indem zur Verfeinerung der Differenziererklasse bzw. der Integriererklasse jeweils eine periodische Task verwendet wird, was auch in den folgenden Abschnitten erfolgt. Für Einzelgerätesteuerungen gelten bzgl. der technischen Randbedingungen ähnliche Überlegungen wie beim Regler.

Beim Einzelgerätesteuerungen/Gruppensteuerungs-Muster wird abhängig von den Anforderungen – d. h. im Rahmen einer Entwurfsentscheidung – festgelegt, dass die Gruppensteuerung durch eine periodische Task verfeinert wird, um die Realzeit-Anforderungen zu realisieren. Der linke Teil der Abbildung 9.4 zeigt die Verfeinerung dieses Analysemusters.

Für diese periodischen Tasks sind die Zykluszeiten entsprechend den zeitlichen Anforderungen an das Analysemuster zu wählen. Für Einzelgerätesteuerungen von Aktoren und Sensoren gelten bzgl. der technischen Randbedingungen die o. g. Überlegungen. Es soll aber offen bleiben, welcher Typ einer Task verwendet wird.

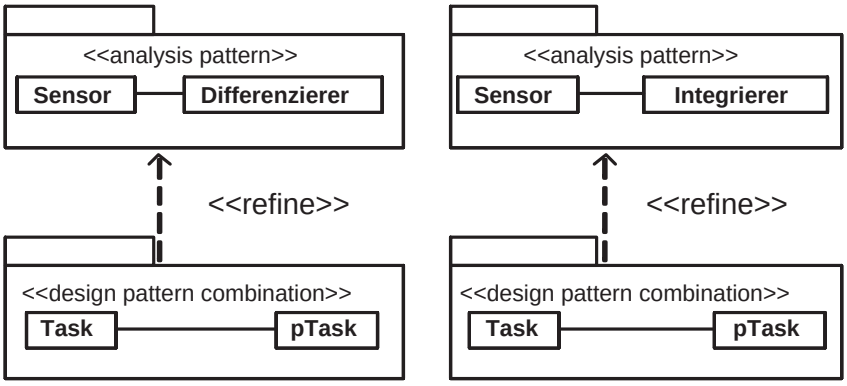


Abbildung 9.3: Die Verfeinerung des Integrierers und des Differenzierers durch Realzeitanforderungen

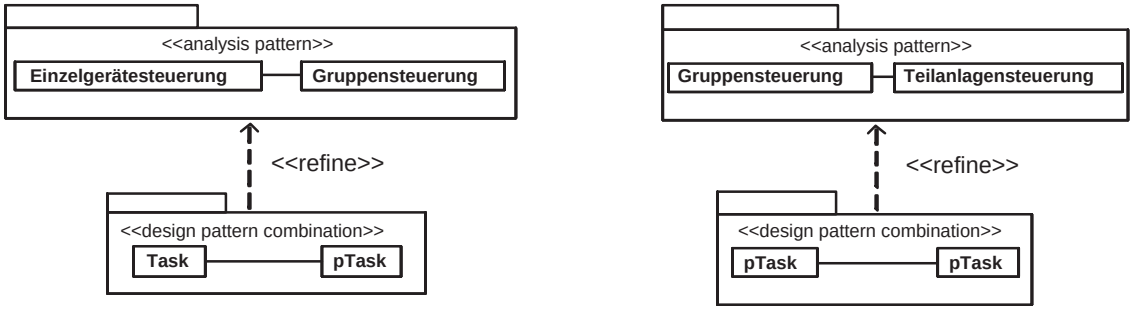


Abbildung 9.4: Die Verfeinerung des Einzelgeräte-/Gruppensteuerungsmusters durch Tasks

Weiterhin muss das Gruppensteuerungen/Teilanlagensteuerungs-Muster durch Tasks verfeinert werden. Dabei werden sowohl die Analyseklassen Teilanlagensteuerung als auch die ihr zugeordneten Gruppensteuerungen wegen der Realzeitanforderungen an das Analysemuster jeweils durch eine Task verfeinert. Beim Entwurf muss entschieden werden, welcher Typ einer Task verwendet wird. So muss die Task einer Teilanlagensteuerung periodisch ihre untergeordneten Gruppensteuerungen aufrufen, während die Gruppensteuerung ihrerseits ihre zugeordneten Einzelgerätesteuerungen überwacht (s. Abbildung 9.5). Daher wird festgelegt, dass Teilanlagensteuerungen durch eine periodische Task realisiert werden. Wie beim Gruppensteuerungen/Teilanlagensteuerungs-Muster wird beim Analysemuster der Teilanlagensteuerungen/Anlagensteuerung vorgegangen, wofür kein Diagramm angegeben wird, wobei Anlagensteuerungen durch Anwendung einer periodischen Task realisiert werden.

In Abbildung 9.5 wird die Verfeinerung des Bild/Funktionsbaustein-Musters gezeigt. Für Bilder wird angenommen, dass sie durch eine sporadische Task verfeinert werden. Für den Funktionsbaustein bleibt es offen, welcher Typ einer Task verwendet wird.

Auch an das Analysemuster des verriegelten Funktionsbausteines bestehen Realzeitanforderungen bzgl. des Verriegelungsobjekts, da eine Verriegelung innerhalb eines bestimmten Zeitraumes vorgenommen werden muss. Deshalb ist es erforderlich, das Verriegelungsobjekt durch eine sporadische Task zu verfeinern, da nur durch die so vorab bereitgestellten Ressourcen eine Garantie für eine zeitgerechte Verriegelung gegeben werden kann. Das Objekt der periodischen Task der Gruppensteuerung ruft bei Werteänderungen das Objekt der Klasse sporadische Task des Verriegelungsobjektes auf, um die Verriegelungsbedingung zu prüfen.

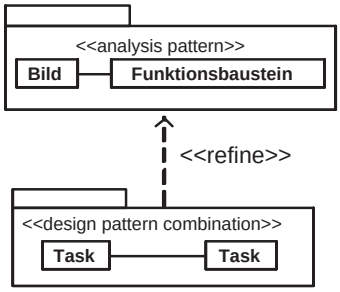


Abbildung 9.5: Die Verfeinerung des Bild/Funktionsbaustein-Musters

9.2.2 Verfeinerung von Analysemustern mit Verteilungsanforderungen

Steuerungssoftware ist in der Regel verteilt. Als Lokationen zur Verteilung von Steuerungssoftware kommen folgende Hardware-Komponenten in Frage:

- Sensoren und Aktoren, die prozessnah sind und in denen Einzelgerätesteuerungen ausgeführt werden.

- Mikro-Controller oder kleinere Leitrechner, die noch prozessnah sind und zur Ausführung von Gruppensteuerungen, von periodischen Regelungen und von Integrierern bzw. von Differenzierern dienen.
- Leitstationen für Teilanlagen- und Anlagensteuerungen, die prozessfern sind.
- Prozessferne Bedienpulte und Leitstände mit Bildern einer Anlage, welche zur Visualisierung des Zustands von Anlagen bzw. deren Bestandteilen verwendet werden.

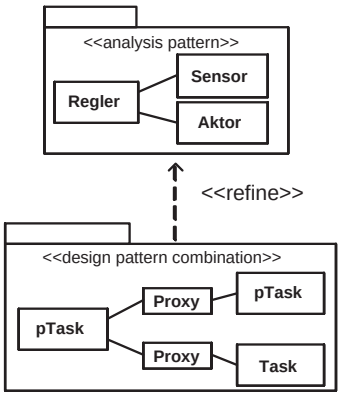


Abbildung 9.6: Das Verfeinerungsmuster verteilter Regler

Im Folgenden soll für eine derartige Verteilung von Steuerungssoftware untersucht werden, welche Analysemuster nicht-funktionale Anforderungen bzgl. der Verteilung besitzen.

Beim Analysemuster des Reglers sind die Einzelgerätesteuerung des Aktors, die Einzelgerätesteuerung des Sensors und die Reglersoftware voneinander entfernt. Sensor und Aktor greifen direkt in den Prozess ein, welcher die Regelstrecke enthält, während der Regler auf einer entfernten Hardware-Komponente installiert ist. Bei der Verfeinerung wird die Entferntheit durch die Verwendung eines Proxy-Musters zur Kommunikation des Reglers mit dem Aktor bzw. mit dem Sensor umgesetzt. Das dadurch entstandene Verfeinerungsmuster wird als verteilter Regler bezeichnet und ist in Abbildung 9.6 gezeigt. Für den Fall, dass verschiedene Kommunikationspartner für Sensor- oder Aktorsteuerungen in Frage kommen, wird das Broker-Muster zur Verfeinerung verwendet. Falls diese Entwurfsentscheidung getroffen wird, soll das dadurch entstehende Verfeinerungsmuster vermittelter Regler genannt werden.

Auch bei den Analysemustern Sensor/Differenzierer bzw. Sensor/Integrierer wird, da die Einzelgerätesteuerung vom Integrierer bzw. dem Differenzierer entfernt ist, ein Proxy-Muster zur Kommunikation zwischen den Komponenten aufgenommen. Dies wird in Abbildung 9.7 gezeigt. Die Einzelgerätesteuerung wird häufig prozessnah eingesetzt, während sich der Integrierer/Differenzierer an einem anderen Ort befindet. Dadurch entstehen zwei Verfeinerungsmuster, die als verteilter Sensor/Differenzierer bzw. verteilter Sensor/Integrierer bezeichnet werden.

Das Gruppensteuerungen/Teilanlagensteuerungs-Muster besitzt Anforderungen bzgl. der Entferntheit, weil Teilanlagensteuerungen prozessfern und Gruppensteuerungen prozessnah eingesetzt werden. Soll eine Teilanlagensteuerung mit einer fest zugewiesenen Gruppensteuerung kommunizieren, wird ein Proxy-Muster zur Verfeinerung eingesetzt, das als lokaler Stellvertreter der Gruppensteuerung dient.

Soll es der Teilanlagensteuerung dagegen erlaubt sein auf Gruppensteuerungen dynamisch zuzugreifen, wird eine Verfeinerung mit dem Broker-Muster vorgenommen. Dieses Verfeinerungsmuster, das als vermitteltes Gruppensteuerungen/Teilanlagensteuerungs-Muster bezeichnet wird, ist in Abbildung 9.8 angegeben. Als zentralen Bestandteil erkennt man den realzeitfähigen RT-Broker [BHS07b], der über Proxy Objekte mit den Tasks, die für die Gruppensteuerung und Anlagensteuerung eingeführt worden sind, kommuniziert.

Beim Einzelgerätesteurungen/Gruppensteuerungs-Muster befinden sich mehrere Einzelgerätesteuerungen und die Gruppensteuerung an unterschiedlichen Orten. Die Einzelgerätesteuerungen

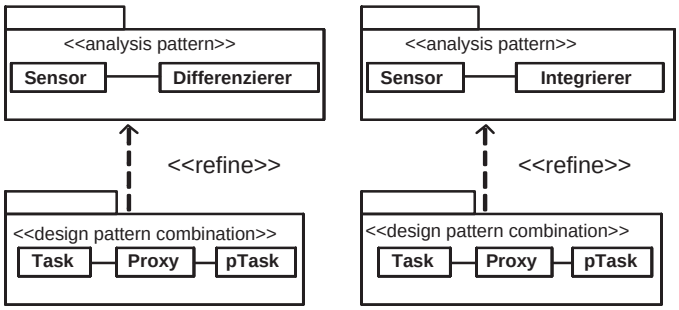


Abbildung 9.7: Die Verfeinerungsmuster verteilter Integrierer bzw. Differenzierer

werden auf prozessnahen Aktoren bzw. Sensoren ausgeführt, während sich die Gruppensteuerung auf einem kleinen Leitrechner befindet. Besteht die nicht-funktionale Anforderung, dass die Gruppensteuerung mit einer ihr fest zugeordneten Einzelgerätesteuerung kommunizieren soll, kann hierfür ein Proxy-Muster zur Verfeinerung verwendet werden, welches zwischen die Gruppensteuerung sowie die Einzelgerätesteuerungen eingefügt wird, was in Abbildung 9.9 gezeigt ist. Das Proxy-Objekt ist passiv und wird im Rahmen der Task der Gruppensteuerung instantiiert und aufgerufen.

Dieses Proxy-Muster dient der Gruppensteuerung als lokaler Stellvertreter für eine Einzelgerätesteuerung. Besteht hingegen die Anforderung, dass die Gruppensteuerung dynamisch auf mehrere Einzelgeräte zugreifen soll, wird statt des Proxy-Musters ein Broker-Muster zur Verfeinerung verwendet, bei dem der Kommunikationspartner dynamisch anhand von bestimmten Eigenschaften ausgewählt wird.

Wie in den beiden zuvor beschriebenen Fällen wird auch beim Teilanlagensteuerungen/Anlagensteuerungs-Muster vorgegangen, wofür kein separates Diagramm angegeben wird.

Das Bild/Funktionsbaustein Muster ermöglicht es, dass die in einem Funktionsbaustein ablaufende Steuerungssoftware und das zur Beobachtung auf einem Leitstand eingesetzte Bild (Abbildung einer technischen Funktionalität mit Zustandsinformationen, wie physikalischen Größen) miteinander kommunizieren. Daher ergeben sich in Abhängigkeit von der exakten Funktionalität des Bildes zusätzliche nicht-funktionale Anforderungen in Bezug auf die Entferntheit.

Besteht die Anforderung, dass ein Bild fest mit einem zugeordneten Funktionsbaustein – z. B. einer Teilanlagensteuerung – zu Kommunikationszwecken verbunden ist, wird ein Proxy-Muster zwischen das Bild und den Funktionsbaustein eingefügt. Auf diese Weise entsteht das Verfeinerungsmuster verteilter Bild/Funktionsbaustein, das in Abbildung 9.10 gezeigt ist. Wenn ein Bild Informationen mit einem oder mehreren Funktionsbausteinen austauschen soll, wird zur Verfeinerung das Broker-Muster zwischen das Bild und den Funktionsbaustein geschaltet. Manchmal besteht die Anforderung, dass Anweisungen oder Befehle des Bildes an mehrere Funktionsbausteine im Rahmen einer Transaktion übertragen werden müssen, um sicherzustellen, dass die Informationen aus einer Anlage auch den Bediener erreichen. Dies bedeutet, dass entweder alle Bilder mit den

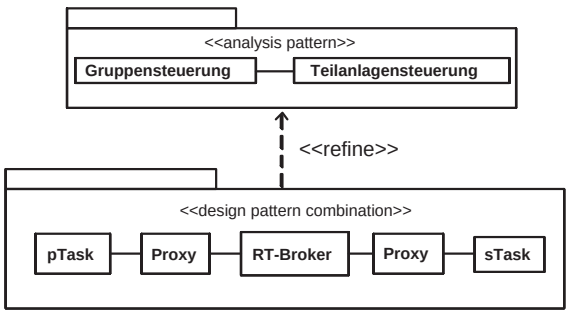


Abbildung 9.8: Das Verfeinerungsmuster vermitteltes Gruppensteuerungen/Teilanlagensteuerungs-Muster

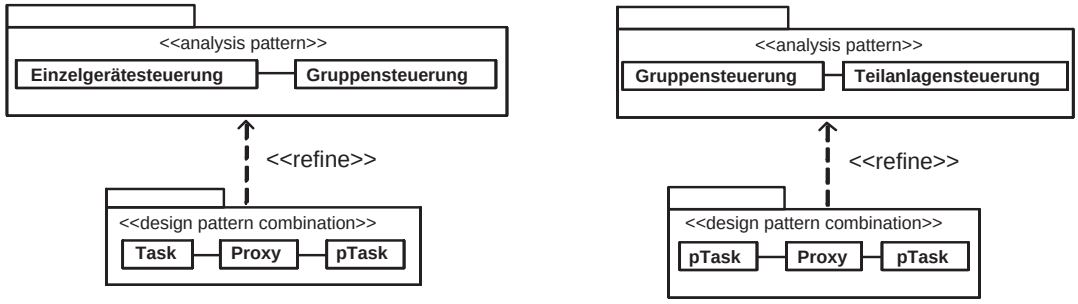


Abbildung 9.9: Die Verfeinerungsmuster für verteilte Einzelgeräte-, Gruppen-, und Teilanlagensteuerungen

passenden Informationen versorgt werden oder gar keines. Weiterhin ist es in diesem Zusammenhang relevant, dass die Informationen in einer Datenbank persistiert werden, um im Fehlerfall den Verursacher ermitteln zu können. Um diese Anforderung zu erfüllen, wird ein Transaktionsmuster zwischen das Bild und den Funktionsbaustein inkludiert. Dadurch entsteht ein Verfeinerungsmuster, das als transaktionsgesicherter Bild/Funktionsbaustein bezeichnet wird.

9.2.3 Verfeinerung von Analysemustern mit Fehlertoleranzanforderungen

Steuerungssoftware besitzt Fehlertoleranzanforderungen, um Ausfälle zu vermeiden bzw. fehlerhaftes Verhalten zu vermeiden. Bei Analysemustern liegen Fehlertoleranzanforderungen in Bezug auf die Stabilität sowie die Redundanz vor. In diesem Abschnitt werden die Analysemuster des Analysemuster-Systems nach Fehlertoleranzanforderungen untersucht. Das Analysemuster des Reglers stellt häufig hohe Anforderungen an dessen Stabilität. Der Regler wird deshalb durch einen Watchdog verfeinert, um dessen Funktion in wiederkehrenden Intervallen zu überprüfen. Fällt der Regler aus, werden entsprechende Recovery-Maßnahmen eingeleitet. Für den Watchdog wird eine eigenständige periodische Task zur zyklischen Prüfung verwendet, da die Prüfung auch bei nicht funktionstüchtiger Regler-Task erfolgen muss. Dieses Verfeinerungsmuster wird gesicherter Regler genannt und ist in Abbildung 9.11 angegeben.

Auch beim Einzelgerätesteuerungen/Gruppensteuerungs-Muster bestehen hohe Anforderungen an die Stabilität der Gruppensteuerung. Ein Watchdog wird zur Verfeinerung des Reglers eingesetzt, um zu wiederkehrenden Zeitpunkten die Funktionalität einer Gruppensteuerung zu überwachen. Natürlich wird auch der Watchdog durch eine Task realisiert, da er immer wieder in zeitlichen Abständen ausgeführt werden muss. Ist die Funktionalität der Gruppensteuerung beeinträchtigt, werden Maßnahmen zum Recovery der Gruppensteuerung eingeleitet. Dieses Verfeinerungsmuster wird in der linken Hälfte der Abbildung 9.13 gezeigt.

Ebenso ist für das Gruppensteuerungen/Teilanlagensteuerungs-Muster die Stabilität der Teil-

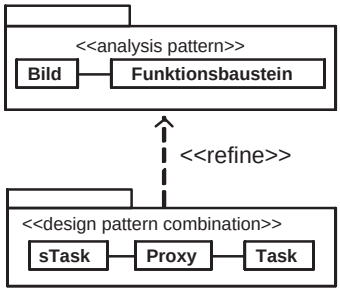


Abbildung 9.10: Das Verfeinerungsmuster verteilter Bild/Funktionsbaustein

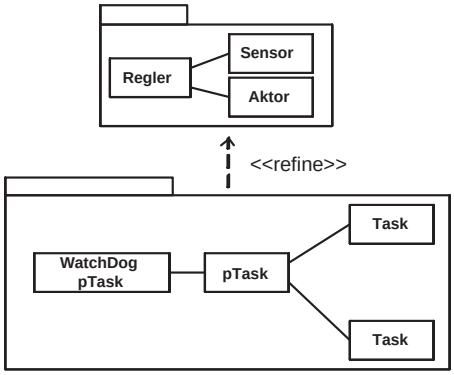


Abbildung 9.11: Das Verfeinerungsmuster gesicherter Regler

anlagensteuerung von hoher Bedeutung, da ihr Ausfall sämtliche zugeordneten Teilanlagensteuerungen betreffen würde. Deswegen wird dieses Muster ebenfalls durch einen Watchdog zur zyklisch wiederkehrenden Funktionsprüfung der Teilanlagensteuerung verfeinert, der im Fehlerfall entsprechende Maßnahmen einleitet. In der rechten Hälfte der Abbildung 9.13 ist dieses Verfeinerungsmuster angegeben.

Gleichermaßen kann das Teilanlagensteuerungen/Anlagensteuerungs-Muster behandelt werden, um fehlertolerante Verarbeitung durchzuführen, wofür keine separate Abbildung angegeben wird. Das dadurch entstehende Verfeinerungsmuster heißt gesicherte Teilanlagensteuerungen/Anlagensteuerung, das hier in keiner Abbildung angeben ist.

Falls für das Bild/Funktionsbausteins-Muster Fehlertoleranzanforderungen vorliegen, ist es möglich die Bild-Klasse durch einen Watchdog zu überwachen. Das dadurch entstandene Verfeinerungsmuster wird als überwachter Bild/Funktionsbaustein bezeichnet und ist in Abbildung 9.12 aufgeführt.

Nun sollen Analysemuster auf Redundanzanforderungen untersucht werden. Beim Einzelgerätesteuerungen/Gruppensteuerungs-Muster wird bei Redundanzanforderungen die Gruppensteuerung durch ein Master-Slave-Muster verfeinert. Zur Realisierung des Masters wird die Task der Gruppensteuerung verwendet. So können mehrere parallel geschaltete Einzelgerätesteuerungen als Slaves Verarbeitungsergebnisse erzeugen, die durch den Master kontrolliert werden. Die Instanzen der Slave Klasse werden im Rahmen der Task für die Gruppensteuerung gebildet und aufgerufen. Auf diese Weise entsteht das Verfeinerungsmuster redundante Einzelgerätesteuerungen/Gruppensteuerung, das in Abbildung 9.14 auf der linken Seite gezeigt ist.

Genauso können beim Teilanlagensteuerungen/Anlagensteuerungs-Muster Redundanzanforderungen bestehen. In diesem Fall wird die Teilanlagensteuerung durch das Master-Slave-Muster verfeinert, wodurch mehrere Gruppensteuerungen parallel geschaltet werden, deren Arbeitsergeb-

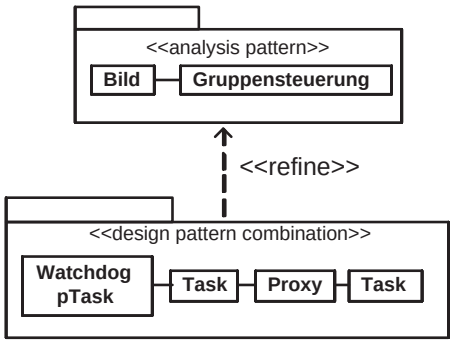


Abbildung 9.12: Das Verfeinerungsmuster überwachtes Bild

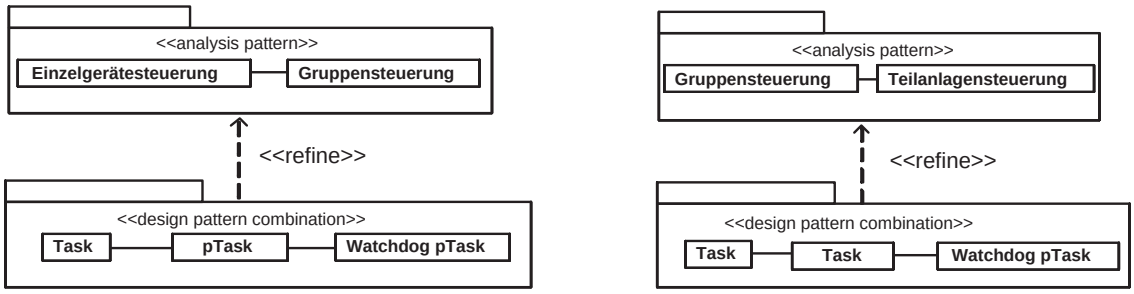


Abbildung 9.13: Die Verfeinerungsmuster zur überwachten Einzelgeräte-, Gruppen- und Teilanlagensteuerung

nisse durch einen neu hinzugenommenen Master überprüft werden. Gleichermaßen kann eine solche Verfeinerung auch für das Teilanlagensteuerungen/Anlagensteuerungs-Muster eingesetzt werden.

Zur Verfeinerung eines verriegelter Funktionsbausteinmusters, werden zunächst sporadische Tasks für den Funktionsbaustein, der eine Einzelgeräte-, Gruppen- oder eine Teilanlagensteuerung ist, verwendet. Damit die Verriegelungsbedingung ausgewertet werden kann, ist es möglich, dass mehrere Funktionsbausteine zur Anwendung kommen. Beispielsweise können die Werte aus mehreren Einzelgerätesteuerungen von Bedeutung sein. Außerdem wird für die Verfeinerung der Klasse des Verriegelungsobjektes selbst eine sporadische Task benutzt. Weiterhin kommt das Entwurfsmuster Recoverable Distributed Observer aus Abschnitt 7.2.3 bei der Verfeinerung zur Verwendung, das sicherstellt, dass der globale Zustand der beteiligten Funktionsbausteine im Verriegelungsobjekt bereitgestellt wird. Die Objekte des globalen und der lokalen Zustandsmanager werden jeweils durch die Task für die Verriegelung und die Task der Gruppensteuerung erzeugt und aufgerufen. In Abbildung 9.15 werden die beteiligten Klassen und ihre Verfeinerungen durch zwei Paketdiagramme angegeben.

9.3 fROPES als Softwareentwicklungsprozess zur Verwendung von Verfeinerungsmustern

Wie bereits eingehend erläutert, sind Verfeinerungsmuster Bausteine, die Unterstützung leisten, um Architekturen in Fertigbauweise zu erstellen. Durch die Anwendung der Verfeinerungsmuster wird sichergestellt, dass die funktionalen und nicht-funktionalen Anforderungen aus der Anforderungsanalyse vollständig und korrekt beim Entwurf berücksichtigt werden. Damit werden frühzeitig Entwurfsfehler ausgeschlossen, deren Behebung in späteren Phasen sehr kostenintensiv ist. In diesem Abschnitt wird fROPES (formal ROPES) als Erweiterung des Softwareprozesses ROPES [Dou99, Dou04], der bereits aus Abschnitt 3.5 bekannt ist, vorgestellt.

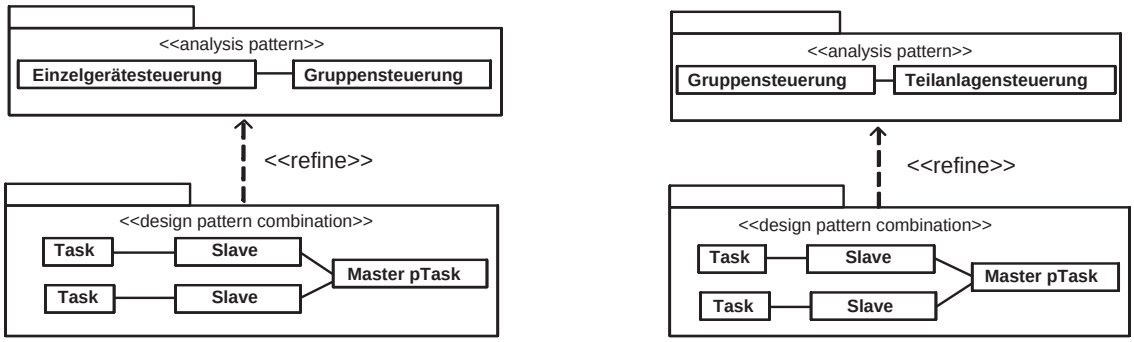


Abbildung 9.14: Die Verfeinerungsmuster mit Redundanzanforderungen

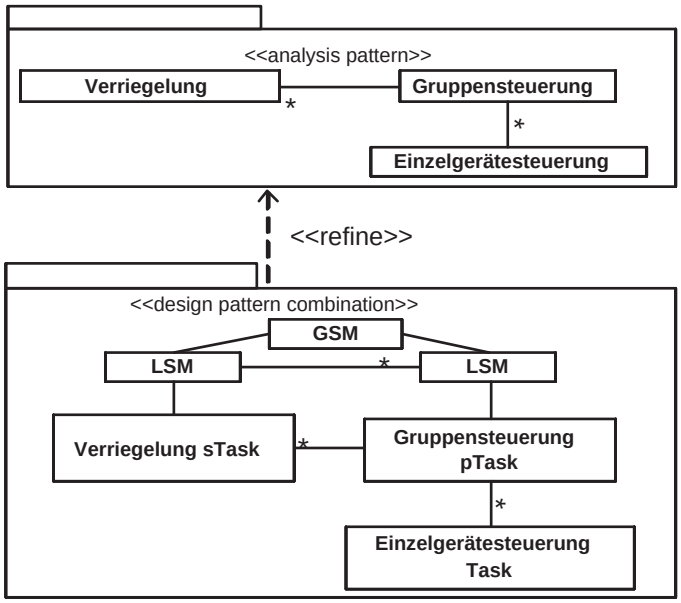


Abbildung 9.15: Das Verfeinerungsmuster verteilte, fehlertolerante Verriegelung

Am Anfang stehen die Modellierung, die Formalisierung und der Nachweis der Korrektheit eines Verfeinerungsmusters, wie sie in den folgenden Kapiteln auf der Basis von cTLA demonstriert wird. Die Verfeinerungsmuster werden in einen Katalog aufgenommen und für die Entwicklung der Anwendung einer bestimmten Domäne bereitgestellt.

Die Musterauswahl ist ein Problem, das für die Wiederverwendung von Entwurfsmustern speziell für Softwarearchitekturen bereits bekannt ist [Zdu07]. Hierfür existieren Entscheidungshilfen, die sich auch für Verfeinerungsmuster heranziehen lassen. Die Entscheidungshilfen umfassen die Berücksichtigung von funktionalen Anforderungen, Qualitätszielen, existierenden Mustervarianten und die Beziehungen zwischen den Mustern (s. Kapitel 4). Eine gute Vorgehensweise besteht darin, die Kriterien für die Bestimmung von Musterkandidaten und die Alternativen zwischen ihnen zu dokumentieren, um zu einer Entscheidung über die Anwendung eines Musters zu gelangen. Dadurch lassen sich Muster zu denen Beziehungen bestehen leichter ausschließen.

Voraussetzung für die Verwendung von Verfeinerungsmustern ist, dass die funktionalen und nicht-funktionalen Anforderungen in strukturierter Form vorliegen. Die Auswahl eines Verfeinerungsmusters erfolgt grundsätzlich auf der Basis des Domänenwissens. Um ein Verfeinerungsmuster anzuwenden, wird aus dem Verfeinerungsmuster-Katalog unter Berücksichtigung der Anforderungen ein geeignetes Verfeinerungsmuster ausgewählt, wie in Abbildung 9.16 angegeben. Dabei müssen existierende Bedingungen bzgl. Sicherheit, Lebendigkeit und Realzeit überprüft werden. Speziell bei der Überprüfung der Realzeitbedingungen muss sichergestellt werden, dass das ausgewählte Verfeinerungsmuster diese erfüllen kann.

Besonderes Augenmerk muss auf die Kombinierbarkeit der Verfeinerungsmuster gelegt werden. Da zwischen den Verfeinerungsmustern Überlappungen bestehen, die sich aus den Beziehungen zwischen den Analysemustern ergeben (s. Kapitel 8), kann man die Entwurfsmodelle der jeweiligen Verfeinerungsmuster leicht kombinieren. Damit geben die Beziehungen zwischen den Analysemustern wichtige Hinweise darauf, welche Analysemuster miteinander kombiniert werden können. Diese Erkenntnis soll auch für die Kombinierbarkeit von Verfeinerungsmustern ausgenutzt werden. Man erkennt, dass die Domäne – in dieser Arbeit also das Gebiet der Anlagensteuerung – besonderen Einfluss darauf hat, wie sich die Verfeinerungsmuster kombinieren lassen. Ein Analysemuster besteht in der Regel aus mindestens zwei Analyseklassen. Die Entwurfsmuster-Kombination besteht aus mehreren Klassen. Die Überlappungen manifestieren sich dadurch, dass zwei in Beziehung stehende Verfeinerungsmuster eine oder mehrere gemeinsame Analyseklassen besitzen. Eine solche Klasse wird als überlappende Analyseklasse bezeichnet. Da die beim Entwurf eingeführten

Entwurfsklassen identisch sind, fallen diese im Entwurfsmodell zusammen. Die Korrektheitseigenschaften beider Verfeinerungsmuster übertragen sich auf die Entwurfsmodelle, was zu korrekt verfeinerten Entwurfsmodellen führt, da korrekte Verfeinerung durch die Verfeinerungsmuster nachgewiesen worden ist. Dabei wird durch die Kompositionalität von cTLA die Verifikation eines Verfeinerungsmusters optimal unterstützt.

Da die Korrektheit eines Verfeinerungsmusters bereits nachgewiesen worden ist, erhält man – mit verhältnismäßig wenig Benutzerinteraktion – ein korrektes Design-Modell mit einer Entwurfsmuster-Kombination, für das kein formaler Nachweis mehr durchgeführt werden muss. Durch die Anwendung von Verfeinerungsmustern wird somit die Anzahl der Beweise reduziert, die notwendig sind, um die Korrektheit zu demonstrieren.

Die Entwurfsmuster-Kombination eines Verfeinerungsmusters muss instantiiert werden, um korrekte Entwürfe für Steuerungssoftware zu erhalten, die später in robusten Implementierungen von Steuerungssoftware – etwa durch die Anwendung von generatorbasierten MDA-Techniken [KWB03] – resultieren. Dieses Vorgehen ist ähnlich zu der in [Tai07] beschriebenen Instantiierung von Entwurfsmustern. Analysemuster hingegen werden nicht instantiiert, da sie nur der Beschreibung der Charakteristika einer Lösung dienen.

Die UML-Modelle der Entwurfsmuster-Kombination des Verfeinerungsmusters werden anschließend für den Entwurf der Steuerungssoftware verwendet. Aus diesem Entwurf wird der Programmcode – wie in ROPES – manuell oder generativ mit dem MDA-Ansatz gewonnen. Falls ein generativer Ansatz verfolgt wird, werden die UML-Modelle direkt als Eingabe für den Generator eingesetzt. Bei der Umsetzung in den Programmcode können immer noch Programmier- bzw. Generierungsfehler entstehen. Abschließend wird die korrekt entworfene Steuerungssoftware in den Produktivbetrieb gebracht.

Außerdem ist die Verwendung der Linearisierbarkeit zu bedenken. Linearisierbarkeit [HW90] ist ein Korrektheitskriterium für nebenläufige Objekte. Es beruht auf der Illusion, dass jede Operation auf einem Objekt augenblicklich ohne Zeitverbrauch – quasi atomar – ausgeführt wird und das Ergebnis äquivalent zu einer sequentiellen Ausführung ist. Deswegen ist diese Eigenschaft bei Mehrkernprozessoren von Bedeutung. Linearisierbarkeit [HS08] ist kompositional, d. h., ist eine Komponente linearisierbar, überträgt sich diese Eigenschaft auf die Komposition. Da die Analysemuster eines Verfeinerungsmusters rein sequentiell arbeiten, ist Linearisierbarkeit für die Entwurfsmuster-Kombination wichtig. Linearisierbare Entwurfsmuster-Kombinationen sind eine Möglichkeit, die Komposition von Verfeinerungsmustern zu garantieren. Allerdings sind hierfür zusätzliche Beweise notwendig, die nach [HS08] im Stil von Assume-Guarantee Beweisen geführt werden können. Auch hier ist die Kompositionalität von cTLA für die Verifikation von Interesse.

Das vorgeschlagene Vorgehen ist eine wesentliche Verbesserung zu existierenden Softwareprozessen wie etwa ROPES, bei denen Analysemodelle in vielen Iterationen transformiert werden und häufig sogar in Vergessenheit geraten. Die Analysemodelle sind aber wichtig, da sie die funktionalen und nicht-funktionalen Anforderungen für spätere Phasen bereitstellen. Auch in den Transformationsschritten vom Analyse- zum Entwurfsmodell wird umfangreiche und komplexe Arbeit geleistet. Allerdings geht auch hier das Wissen über durchgeführte Transformationen oft wegen mangelnder Dokumentation bzw. Formalisierung verloren. Gründe hierfür sind auch in Veränderungen der Projektorganisation, z. B. dem Wechsel von Mitarbeitern, zu sehen. Häufig erfolgt die Transformation ohne Verwendung bzw. Verständnis der formalen Verfeinerung, sodass nicht sichergestellt ist, dass der Entwurf alle Anforderungen korrekt und vollständig berücksichtigt. Daraus resultieren Fehler, die durch teure Maßnahmen zur Qualitätssicherung erst in späteren Phasen des Entwicklungsprozesses entdeckt werden. Möglicherweise treten Fehler auch erst beim Betrieb der Software auf und verursachen große Probleme. Die Wiederverwendung von Bestandteilen der Analysemodelle und den korrekt verfeinerten Entwurfsmodellen einer Domäne auf der Grundlage von Verfeinerungsmustern bietet das Potential, die Fehlerhäufigkeit bei Steuerungssoftware zu reduzieren. Weiterhin können Entwurfsentscheidungen bei der Wartung aus dem Design der Steuerungssoftware zurück bis zu den Anforderungen vollzogen werden.

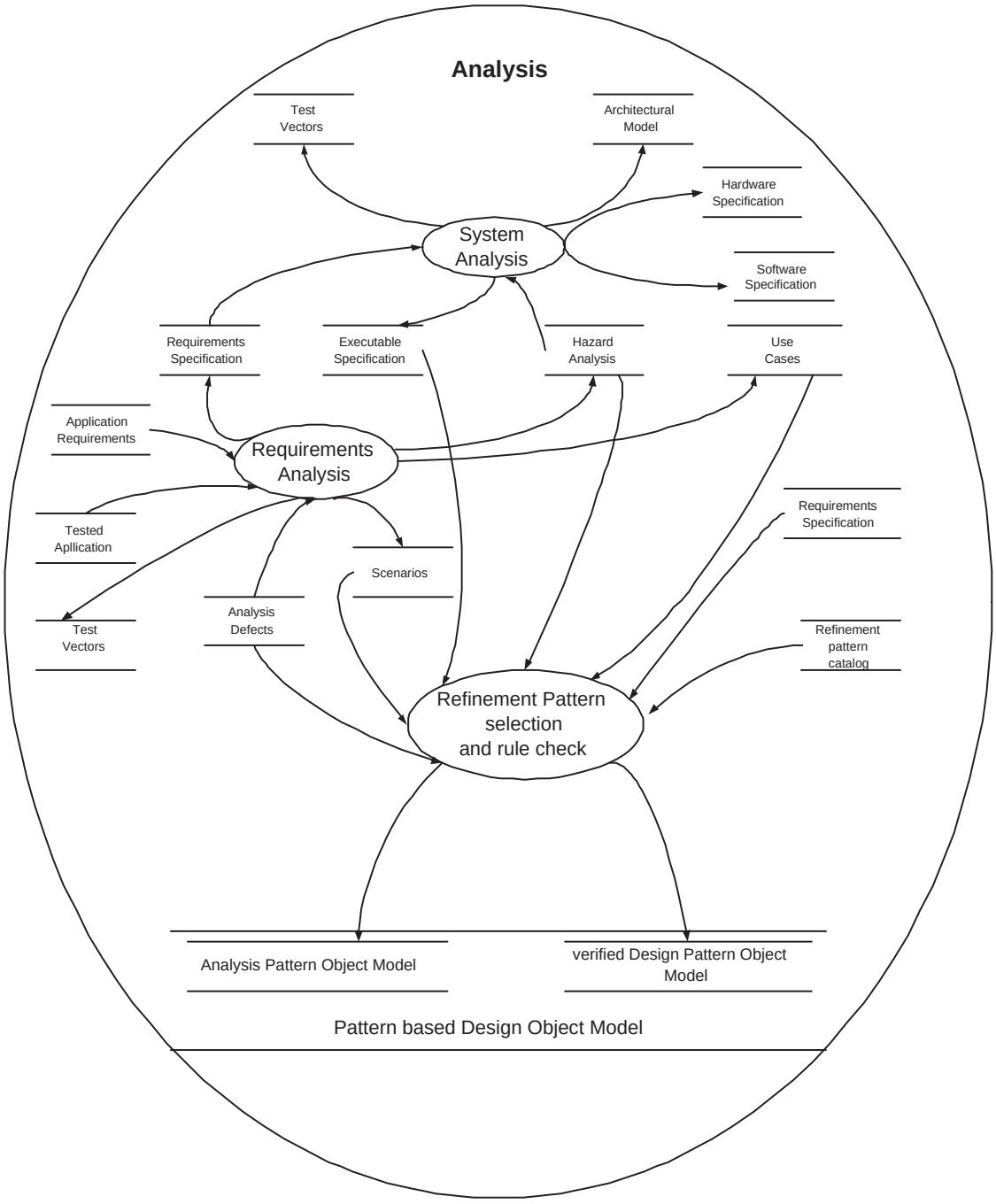


Abbildung 9.16: Die Anwendung von Verfeinerungsmustern mit fROPES

Kapitel 10

Übersetzung eines Analysemusters nach cTLA

In den letzten Jahren sind zahlreiche Ansätze zur semantischen Fundierung der UML publiziert worden. Diese basieren auf den unterschiedlichsten formalen Methoden. Da man davon ausgeht, dass ca. 500 verschiedene Ansätze für formale Methoden [Mes02] existieren, können nur die für die Arbeit relevanten Ansätze vorgestellt werden.

Zentrales Problem bei der Formalisierung der UML ist die unzureichende Beschreibung ihrer Semantik. Auf diese Problematik wird in [DH04a] hingewiesen und als Ursache hierfür die große Anzahl von Teilsprachen, aus denen die UML besteht, genannt. Harel hat mittlerweile eine Formalisierung der Semantik von Statechart-Diagrammen in [DH04b] angegeben. Eine Formalisierung der UML 1.3 Zustandsmaschine für Model-Checking wird in [LP99] behandelt, um die Verifikation auf der Basis von Spin [BBAF+01] vorzunehmen. Für die Zustandsmaschine wird in [DL99] eine operationale Semantik definiert. Eine Formalisierung der Zustandsmaschine auf der Basis von algebraischen Methoden wird in [RACH00] angegeben. Die Formalisierung von Sequenzdiagrammen auf der Basis von MSCs wird in [HS03] vorgestellt. In [Web97] wird die Transformation von UML-Diagrammen in die auf Mengenoperationen basierende Spezifikationssprache Z vorgestellt. Speziell wird auf Klassen-, Statechart-, und Sequenzdiagramme eingegangen. Auch Lano und Bicarregui [KL99] beschreiben die Übersetzung von UML-Diagrammen in die Spezifikationssprache Z sowie in der objektorientierten Realzeit-Spezifikationssprache VDM++. Giehse [GTB+03] beschreibt eine Formalisierung von Statechart-Diagrammen auf der Basis von CTL. Außerdem wird bei diesem Ansatz musterbasiert vorgegangen. Allerdings wird die Verfeinerung nicht betrachtet. Für die Verifikation wird der Model-Checker SMV [BBAF+01] von der Carnegie Melon University eingesetzt. Schaefer und Merz haben das Werkzeug Hugo [SKM01], das die Übersetzung von Statechart-Diagrammen für Spin unterstützt, vorgestellt.

Mittlerweile gibt es große, internationale Projekte wie das Omega Projekt[1] [GH04a], in dem an einer Formalisierung auf der Basis von Abstract State Machines (ASM) gearbeitet wird [FW04]. Zur Verifikation wird der Theorem-Beweiser PVS [HvdZ05] eingesetzt.

Herrmann und Kraemer verwenden UML-Aktivitätsdiagramme zur Spezifikation von Diensten und deren Kollaborationen im Rahmen von Service orientierten Architekturen (SOA) [KH07b, KBH07]. Die Aktivitätsdiagramme besitzen den semantischen Umfang von StructuredActivities, IntermediateActivities und CompleteActivities aus dem Metamodell der UML [OMG03]. Zur Formalisierung der Aktivitätsdiagramme wird cTLA verwendet. Die Formalisierung ermöglicht es, Sicherheits- und Lebendigkeitseigenschaften einer Dienstspezifikation zu verifizieren. Durch Modelltransformation werden die Aktivitätsdiagramme in UML Zustandsmaschinen übersetzt. Aus den Zustandsmaschinen wird anschließend ausführbarer Java Code generiert, der z. B. auf der Java EE 5 [BR07] Plattform ausgeführt werden kann. Zur Formalisierung der UML Zustandsmaschinen ist ein cTLA-Spezifikationsstil – nämlich cTLA/e [KHB06] – entwickelt worden. Der Spezifikationsstil ist eine ausführbare Form von cTLA. Eine Zustandsmaschine wird durch einen cTLA/e

[1] http://www-omega.imag.fr

Prozess formalisiert. Der Spezifikationsstil unterstützt Signale und die Definition von Eingangs- und Ausgangswarteschlangen sowie von Hilfsvariablen, etwa um die aktuelle Position in einer Warteschlange anzugeben. Für jede Warteschlange werden *Enqueue* und *Dequeue* Aktionen definiert. Jede Zustandsmaschine besitzt eine Zustandsvariable *state*, die einen Zustand aus einer Menge der Zustände eines Statecharts annehmen darf. Transitionen und Actions werden durch so genannte interne Aktionen umgesetzt. Diese bestehen bei Transitionen aus mehreren Subaktionen. Die erste Subaktionen prüft, ob ein Trigger ausgelöst worden ist und ob der Ausgangszustand zur Transition passt. Eine weitere Subaktion nimmt die Änderung des Kontrollzustandes vor. Weiterhin gibt es jeweils Subaktionen zur Behandlung der Warteschlangen und zur Weiterleitung von Signalen zwischen den Zustandsmaschinen. Abschließend nimmt eine Subaktion Änderungen an den Hilfsvariablen vor. In Systemprozessen werden die internen Aktionen miteinander gekoppelt. Wenn Zustandsmaschinen miteinander kommunizieren, werden die Aktionen zum Einfügen und Entfernen von Signalen miteinander gekoppelt. Insgesamt lässt sich durch die cTLA/e eine Formalisierung der Implementierung einer Zustandsmaschine erreichen. Mit der Formalisierung von Aktivitäten und Zustandsmaschinen auf der Basis TLA ist es möglich, Verfeinerungsbeweise zwischen beiden Diagrammarten, die sich auf unterschiedlichem Abstraktionsniveau befinden, durchzuführen. Weiterhin wird der Spezifikationsstil cTLA/c [KH07a] eingeführt, der es ermöglicht Aktivitäten, die das dynamische Verhalten von Kollaborationen beschreiben, auf der Basis von cTLA-Prozessen und Prozesskopplungen zu formalisieren. Zur Übersetzung der Aktivitäten in cTLA werden Produktionsregeln und cTLA-Prozesse verwendet.

Auch bei der Formalisierung von Entwurfsmustern sind substantielle Fortschritte erzielt worden, wie sie etwa in [TN03, Tai07] vorgestellt werden. Darin stellt T. Taibi die Formalisierung von Entwurfsmustern auf der Basis der Balanced Pattern Specification Language (BPSL) vor. Ein Teil dieser Sprache verwendet Prädikatenlogik für die Formalisierung der strukturellen Aspekte und wird mit S_{BPSL} bezeichnet. Zusätzlich gibt es eine Subsprache B_{BPSL}, die TLA-Formeln zur Spezifikation des Verhaltens von Entwurfsmustern einsetzt. Es wird beispielhaft die Komposition von Objekten bzw. das Anlegen von Links zwischen Objekten untersucht. Der Ansatz beschränkt sich allerdings nur auf Entwurfsmuster und betrachtet keine Analysemuster. Verhaltensdiagramme der UML werden nicht betrachtet. Die formale Verfeinerung von Entwurfsmustern mit TLA wird in [THNMN09] behandelt.

In den nun folgenden Abschnitten dieses Kapitel wird die Übersetzung von Verfeinerungsmustern aus den Beispielen in cTLA behandelt, damit der Nachweis der Korrektheit eines Verfeinerungsmusters auf einer formalen Basis durchgeführt werden kann. Die Übersetzung von UML-Diagrammen in cTLA ist bereits umfangreich in [GHK00] und [GH04b] vorgestellt worden. Die Semantik wird als operationale Semantik auf der Grundlage von Zustandstransitionssystemen (STS) (s. Kapitel 5) definiert [GHK99b, GHK99a]. In Abschnitt 10.1 werden die Prinzipien der Übersetzung von Mustern, die mit UML-Diagrammen spezifiziert worden sind, nach cTLA vorgestellt. Anschließend wird in Abschnitt 10.2 das Analysemuster Regler mit seinen Diagrammen und den nicht-funktionalen Anforderungen präsentiert. Der Abschnitt 10.3 zeigt die Übersetzung dieses Analysemusters nach cTLA.

10.1 Grundstruktur der Transformation von Mustern nach cTLA

In diesem Abschnitt wird die Grundstruktur für die Übersetzung der Analysemuster und Entwurfsmuster eines Verfeinerungsmusters nach cTLA beschrieben. Aufgrund der semantischen Mächtigkeit der UML ist es im Rahmen dieser Arbeit nicht möglich, eine Übersetzung sämtlicher UML-Diagramme mit ihren umfangreichen Konstrukten nach cTLA anzugeben. Stattdessen wird der Umfang der zu übersetzenden Sprachelemente auf den für die Spezifikation der Verfeinerungsmuster verwendeten Konstrukte eingeschränkt. Für die Übersetzung weiterer Konstrukte sei auf die schon im vorherigen Abschnitt vorgestellte Literatur verwiesen. Weitere Übersetzungsdetails werden in den folgenden Abschnitten des Kapitels näher erläutert.

Zur Erläuterung der Übersetzung wird als Beispiel der Aktor mit seinen UML-Diagrammen

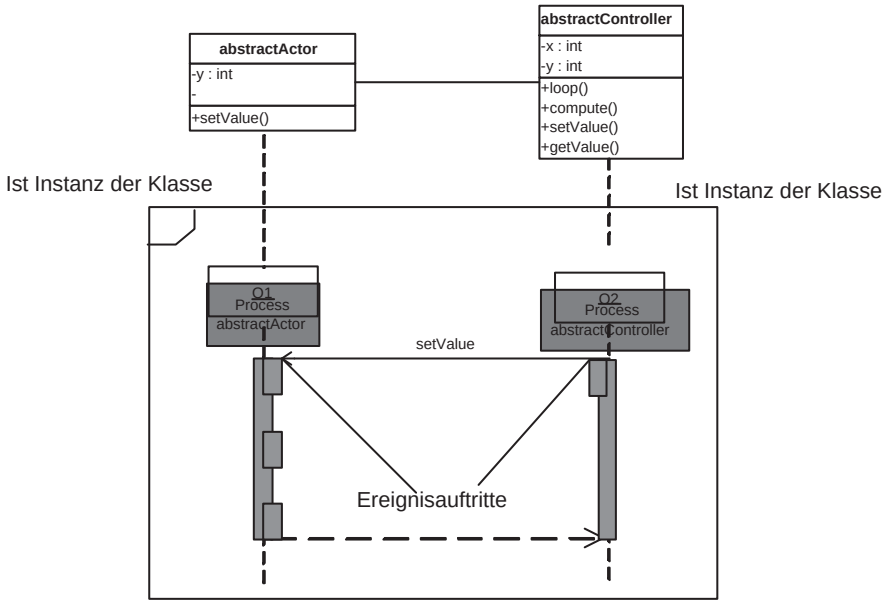


Abbildung 10.1: Das Klassendiagramm eines Aktors und ein typisches Sequenzdiagramm

herangezogen. In Abbildung 10.1 ist oben das Klassendiagramm des abstrakten Aktors gezeigt. Darunter ist ein typisches Sequenzdiagramm angegeben. In dem Sequenzdiagramm wird gezeigt, wie eine *setValue*-Nachricht vom *abstractController*-Objekt an das Aktorobjekt gesendet wird. Das Statechart-Diagramm der Klasse des Aktors wird in Abbildung 10.2 gezeigt. Es besteht aus den Zuständen *init* und *processed*, zwischen denen eine Transition modelliert ist, die bei einem *setValue*-Aufruf ausgeführt wird. An dieser Transition befindet sich eine Aktivität, die das Schreiben eines neuen Stellwertes modelliert. Diese Aktivität besteht aus der UML-Action *writey*, die einen eingehenden Objektfluss *val* besitzt. Dieser Objektfluss enthält den eingehenden Wert und stellt ihn der Aktion zur Verfügung. Zwischen den Zuständen *processed* und *init* ist eine weitere triggerless Transition spezifiziert, an der eine Aktivität modelliert ist. Bevor der Zustand *processed* verlassen wird, wird die Aktivität der Transition ausgeführt, die die UML-ReplyAction *reply* enthält. Diese UML-Action beantwortet den *setValue*-Aufruf an die Klasse *abstractActor*.

Zur Erläuterung der Übersetzung werden zunächst Klassen und Statechart-Diagramme betrachtet. Bei den Statechart-Diagrammen werden zuerst Zustände und dann Transitionen übersetzt. Da Transitionen Aktivitäten mit UML-Actions enthalten, wird deren Übersetzung im Kontext einer Transition beschrieben. Abschließend werden Sequenzdiagramme behandelt. Ausgangspunkt der Übersetzung ist eine Klasse, die in einem Muster verwendet wird. Für jede Klasse, deren Verhalten mit einem ihr zugeordneten Statechart-Diagramm spezifiziert ist, wird ein eigenständiger cTLA-Prozesstyp eingeführt, um die Semantik der Klasse durch cTLA zu definieren. Zur Übersetzung der Klasse *abstractActor* wird der Prozesstyp *abstractActor* in Abbildung 10.1, der schon aus Kapitel 6 bekannt ist (durch Schraffur hinter dem Objekt *abstractActor* angedeutet), verwendet. Weiterhin wird in Abbildung 10.3 ein Zustandsübergangsdiagramm mit den Zuständen $s_0, s_1, \ldots, s_6$ angegeben, das verwendet wird, um das abstrakte Prinzip der Übersetzung durch ein STS zu erläutern. Ein Zustand wird durch einen Kreis mit einem Bezeichner und einer Menge von Zustandsvariablen sowie deren Werten angegeben. Die Übergänge zwischen den Zuständen werden durch gerichtete Kanten mit dem Namen der Aktion, die zu einem Zustandsübergang führt, angegeben. Lamport verwendet in [Lam95a] sog. Prädikat-, Aktionsdiagramme, um TLA-Spezifikationen grafisch darzustellen. Später werden die textuellen Spezifikationen der zugehörigen cTLA-Prozesse in den Abbildungen 10.4 und 10.5 gezeigt. Die Aktionen dieser Prozesse führen zu den Transitionen zwischen den Zuständen in Abbildung 10.3.

Wenn eine Klasse Attribute besitzt, werden diese auf Zustandsvariablen des Prozesses mit demselben Datentyp abgebildet. Der Prozesstyp *abstractActor* besitzt die Zustandsvariable y vom

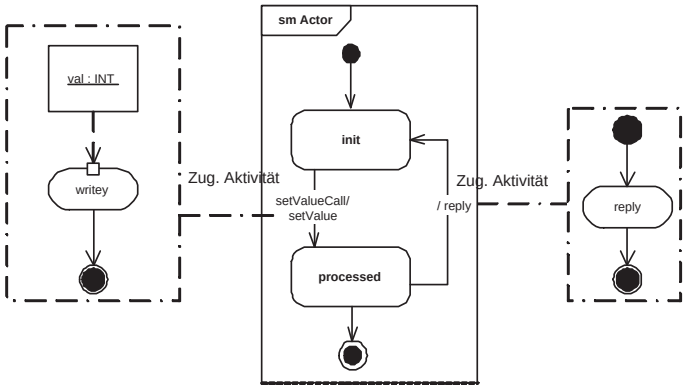


Abbildung 10.2: Das Statechart-Diagramm eines Aktors mit einer Action

Typ INTEGER, um den Stellwert zu modellieren und repräsentiert den Datenzustand.

Bei der Übersetzung von Statecharts werden ausschließlich flache, d. h. Statecharts, die aus einfachen Zuständen (Simple States im UML Metamodell) bestehen, betrachtet. In der Literatur existieren zahlreiche auf Graphgrammatiken (z. B. [GPP98]) beruhende Verfahren, um zusammengesetzte bzw. nebenläufige Zustände in flache Statecharts zu transformieren. Um die Zustände des Statecharts, die den Kontrollzustand eines Objektes bilden, zu übersetzen, wird eine zusätzliche Zustandsvariable mit dem Bezeichner *state* aufgenommen, die als Datentyp die Menge, die aus den Bezeichnern der einzelnen Kontrollzustände besteht, besitzt. Diese Zustandsvariable ist in jedem Zustand der Abbildung 10.3 enthalten.

Weiterhin muss die Warteschlange eines Statecharts bei der Übersetzung berücksichtigt werden, da der Zustand der Warteschlange von großer Bedeutung ist. Die Warteschlange eines Statecharts wird durch eine Zustandsvariable mit dem Bezeichner *qu* vom Datentyp SEQUENCE OF INTEGER modelliert. Durch Aktionen zum Einfügen *enqueue* und Entfernen *dequeue*[2] von Elementen der Warteschlange erfolgt eine Modifikation der Warteschlange. Die Zustandsvariable *qu* ist daher in jedem Zustand der Abbildung 10.3 enthalten. Da der Trigger einer Transition zur Konsumierung eines Ereignisses aus der Warteschlange führt, muss ein Element vom Kopf der Warteschlange entfernt werden und zur weiteren Verarbeitung bereitgestellt werden. Der Zugriff auf die Warteschlange wird durch die zusätzliche Boolesche Zustandsvariable *sync* synchronisiert. Das STS und der Prozesstyp *abstractActor* enthalten diese beiden Aktionen. Um Aufrufe und deren Verarbeitung durch die Warteschlange zu modellieren, erhält die Zustandsvariable *state* die Werte "enqueued" und "dequeued".

Die Übersetzung von Transitionen gestaltet sich komplexer, da diese Zustände, Ereignisse, Bedingungen und Aktivitäten miteinander in Beziehung setzen. Das kann dazu führen, dass eine Transition den Daten,- den Kontroll- sowie den Zustand der Warteschlange modifiziert. Für den Guard, den Ausgangs- sowie den Zielzustand einer Transition eines flachen Statechart-Diagramms, wird eine separate, parameterlose cTLA-Aktion in den Prozesstyp der Klasse zur Modellierung des entsprechenden Zustandsübergangs eingeführt. Dadurch wird die Semantik einer Transition eines Statecharts auf der Basis der Transitionsklassen einer cTLA-Aktion definiert. Wegen der Run-To-Completion Semantik (s. Abschnitt 3.4.3) der Zustandsmaschine eines Statecharts darf eine einmal gestartete Verarbeitung dieser Aktion nicht unterbrochen werden. Dies wird durch Prüfung der Zustandsvariable *sync* in der Vorbedingung einer zu einer Transition gehörigen Aktion gewährleistet. Weiterhin wird in der Vorbedingung der cTLA-Aktion der Ausgangszustand sowie gegebenenfalls ein Guard als Prädikat spezifiziert. Weil ein Trigger zur Konsumierung eines Ereignisses aus der Warteschlange führt, ist als Ausgangszustand nur ein Zustand zulässig, indem die Zustandsvariable *state* den Wert "dequeued" besitzt. Bei einer Transition, die keinen Trigger besitzt, sind auch Vorgängerzustände zulässig, bei denen *state* nicht den Wert "dequeued" besitzt. In der Abbildung 10.3

[2]In Abhängigkeit von der Spezifikation kann es durchaus sinnvoll sein mehrere *dequeue* Aktionen mit verschiedenen Aktionenparametern zu verwenden. Alternativ ist ein generischer Aktionenparameter *message* denkbar.

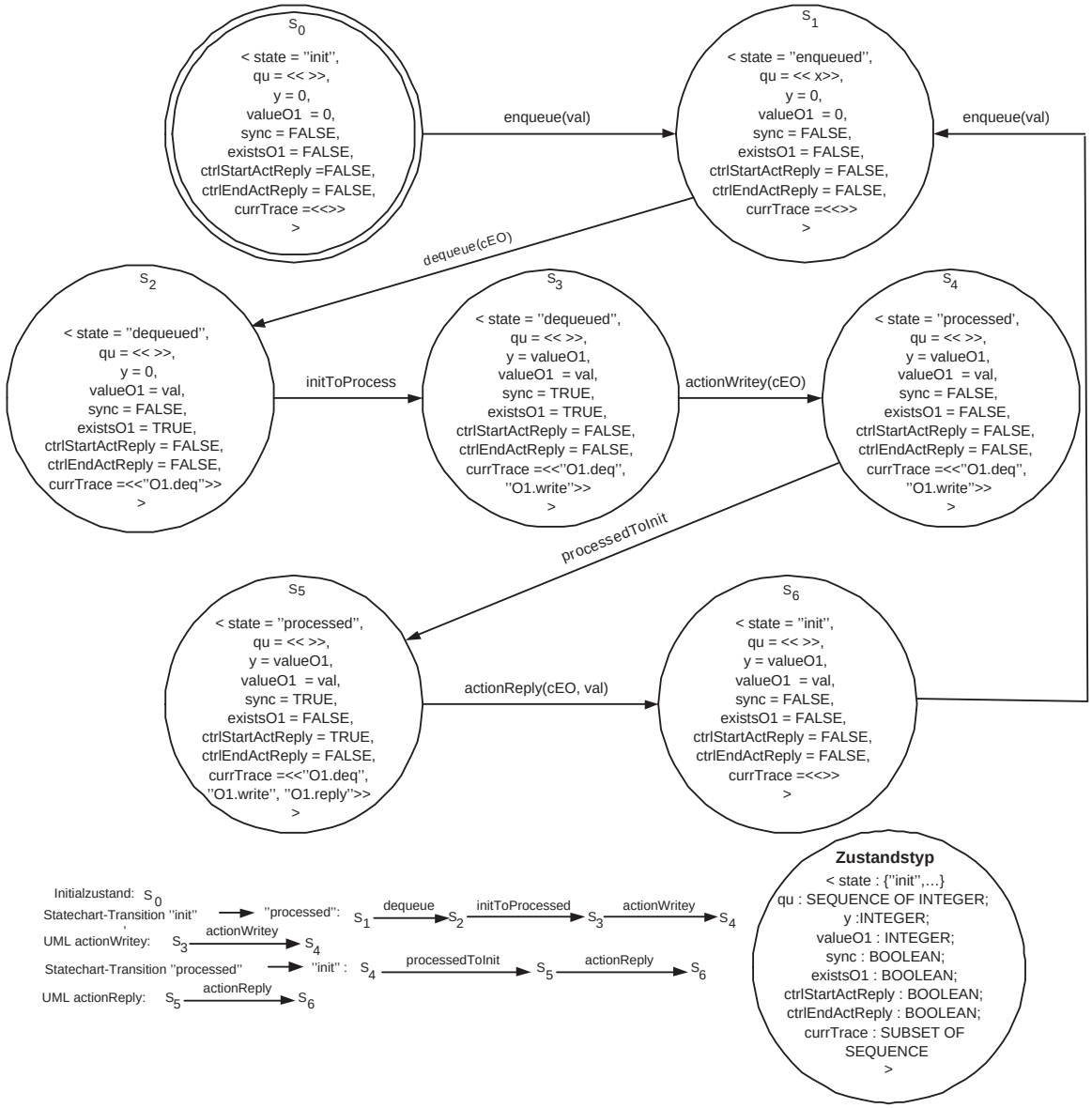


Abbildung 10.3: Das STS des Aktors

entspricht die parameterlose Aktion *initToProcessed* der Übersetzung der Transition vom Zustand ”init” in den Zustand ”processed” des Statechart-Diagramms. In der Vorbedingung dieser Aktion wird geprüft, ob die Zustandsvariable *state* den Wert ”dequeued” hat und ob ein Ereignis vorliegt. Man erkennt, dass s_2 ein Vorgängerzustand ist, bei dem *state* den Wert ”dequeued” besitzt. Im Effekt der Aktion wird die Zustandsvariable *sync* auf TRUE gesetzt, um die RTC-Semantik zu garantieren.

Für den Fall, dass an einer Transition keine Aktivität modelliert ist, wird in der Aktion auch der neue Zielzustand der Zustandsvariable *state* zugewiesen. Falls eine Transition eine UML-Aktivität mit UML-Actions enthält – wie bei der gerade betrachteten Transition zwischen den Zuständen ”init” und ”processed” – ergeben sich Ergänzungen für die Übersetzung, die nun betrachtet werden. Für die Kontroll- und Objektflüsse von Aktivitäten werden zusätzliche Zustandsvariablen in den zugehörigen Prozesstyp aufgenommen. Diese modellieren einerseits die Existenz eines Flusses und im Falle eines Objekflusses auch die Werte, die an UML-Actions übergeben und von diesen weitergereicht werden. Die zur Übersetzung der Action eingeführte Aktion stellt für die Boole-

schen Zustandsvariablen der eingehenden Kontroll- bzw. Objektflüsse einer Aktivität geeignete Werte bereit. Für die Übersetzung von UML-Actions werden eigenständige cTLA-Aktionen aufgenommen, die deren Semantik festlegen. Was eine derartige Aktion konkret tut, hängt von der Semantik ihrer UML-Action ab. Möglich sind Modifikationen des Datenzustandes und des Zustandes der Warteschlange des betrachteten Objektes. Natürlich sind Veränderungen der für Kontroll- und Objektflüsse eingeführten Zustandsvariablen möglich. Falls es sich um kommunikationsorientierte *Actions* bzw. *ReplyActions* handelt, werden Aktionenparameter mit geeigneten Datentypen aufgenommen. Die Aktionenparameter werden verwendet, um Werte zwischen den Prozessen, die Objekte repräsentieren, zu übertragen. Weiterhin besitzt jede Aktion einen Parameter *cEO* vom Typ *CEO*, um den zur Action gehörigen Ereignisauftritt zu modellieren. Der Parameter wird in der Vorbedingung der Aktion mit einem Wert belegt. Die abschließende[3] Aktion einer Aktivität nimmt die Transition in den Zielzustand vor.

In den Prozesstyp *abstractActor* wird zur Übersetzung der UML-Action *writey* die Aktion *actionWritey* aufgenommen. Zusätzlich werden zwei Zustandsvariablen – nämlich die Variablen *existsO1* und *valueO1* – aufgenommen. Dabei gibt die Boolesche Zustandsvariable *existsO1* an, ob ein Objektfluss vorliegt. Die Zustandsvariable *valueO1* vom Typ INTEGER nimmt den Wert des Objektflusses auf. Die UML-Action *actionWritey* wird durch die Zustandsfolge $s_3 \xrightarrow{actionWritey} s_4$ übersetzt. Man erkennt, dass der Wert der Zustandsvariable *valueO1* der Zustandsvariable y zugewiesen wird. Durch die Zustandsfolge $s_1 \xrightarrow{dequeue} s_2 \xrightarrow{initToProcessed} s_3 \xrightarrow{actionWritey} s_4$ wird damit insgesamt die Übersetzung der Transition des Statechart-Diagramms zwischen "init" und "processed" angegeben.

Mittels der Zustandsfolge $s_4 \xrightarrow{processedToInit} s_5 \xrightarrow{actionReply} s_6$ wird die triggerless Statechart Transition zwischen "processed" und "init" übersetzt. Durch die Aktion *actionReply*, die zur Übersetzung in den Prozesstyp *abstractActor* aufgenommen wird, wird die Antwort des Aufrufes an den Aktor modelliert. Durch die Zustandsfolge $s_5 \xrightarrow{actionReply} s_6$ wird somit die Übersetzung der UML-Action *actionReply* angegeben. Zunächst werden zwei zusätzliche Boolesche Zustandsvariablen – nämlich die Variablen *ctrlStartActReply* und *ctrlEndActReply* – aufgenommen, die den Start- und den Endknoten für den Kontrollfluss modellieren und angeben, ob ein Kontrollfluss existiert bzw. die Ausführung einer UML-Action abgeschlossen ist. In der Aktion *actionReply* wird geprüft, ob der Kontrollfluss *ctrlStartActReply*, der zur Ausführung der Aktion notwendig ist, existiert. Im Effekt werden die Zustandsvariablen *sync* und *ctrlStartActReply* zurück auf FALSE gesetzt. Weiterhin wird die Zustandsvariable *ctrlEndActReply*, die angibt, ob die UML-Action beendet worden ist, auf TRUE gesetzt.

Ein Sequenzdiagramm stellt Informationen darüber bereit, von welchen Klassen Objekte instantiiert werden und wie diese Objekte durch den Austausch von Nachrichten miteinander interagieren. Zur Übersetzung von Sequenzdiagrammen wird ein eigenständiger Prozesstyp mit dem Bezeichner *SequenceDiagram* hinzugenommen, der die zulässigen Traces eines Sequenzdiagramms aus Prozessparametern berechnet. Dieser Prozesstyp ist bereits in Kapitel 6 vorgestellt worden. Ein solcher Prozesstyp verfügt über eine spezielle Aktion – mit *permittedActionOfSD* bezeichnet – , die prüft, ob die Ausführung der Aktion eines Objektprozesses durch den totalen Trace des Sequenzdiagramms zugelassen wird. In dem Prozesstyp gibt es eine Zustandsvariable *currentTrace*, die auch in den Zuständen aus Abbildung 10.3 (abgekürzt durch *currTrace*) gezeigt wird. Diese Zustandsvariable wird verwendet, um alle bisherigen Ereignisauftritte in einer TLA Sequence aufzunehmen.

Ein Objektprozess ist eine Instanz eines Prozesstyps, der zur Übersetzung einer Klasse eingeführt worden ist. Durch einen Objektprozess wird das Verhalten eines Objektes beschrieben, wie es in einem Sequenzdiagramm angegeben wird. Das Prinzip für die Kopplung von separaten Objektprozessen besteht darin, für zwei Objektprozesse, die miteinander interagieren, eine gemeinsame Systemaktion zu spezifizieren, die die Aktionen der beiden Objektprozesse miteinander koppelt. Zusätzlich ist die Aktion *permittedActionOfSD* Bestandteil der Kopplung, um sicherzustellen, dass eine Interaktion der zugehörigen Objekte durch das Sequenzdiagramm spezifiziert worden ist. In der zugehörigen Systemaktion stottern die übrigen Objektprozesse. Prozesslokale Aktionen eines

[3]Dies entscheidet sich daran, ob die zugehörige Action mit einem Endzustand verbunden ist.

```
PROCESS abstractActor

IMPORT TLC, Sequences, Naturals

VARIABLES
 state : {"init", "enqueued", "dequeued", "processed"}; ! Kontrollzustand
 qu : SEQUENCE OF INTEGER; ! Warteschlange
 y, value01 : INTEGER;
 sync, exists01, ctrlStartActReply, ctrlEndActReply : BOOLEAN;

INIT ≜  ∧ state = "init"
        ∧ qu = ≪ ≫
        ∧ y = 0
        ∧ value01 = 0
        ∧ exists01 = FALSE
        ∧ ctrlStartActReply = FALSE
        ∧ ctrlEndActReply = FALSE
        ∧ sync = FALSE
ACTIONS
! Aktion zum Einfügen und Entfernen aus der Queue
enqueue(value : INTEGER) ≜ ∧ state = "init"
                           ∧ state′ = "enqueued"
                           ∧ qu′ = append(qu, value)
                           ∧ UNCHANGED ⟨y, value01, ctrlStartActReply, ctrlEndActReply⟩;

dequeue(cE0 : E0) ≜ ∧ state = "enqueued" ∧ cE0 = "actor.dequeue" ∧ ¬sync
                    ∧ state′ = "dequeued"
                    ∧ qu′ = tail(qu)
                    ∧ value01′ = head(qu)
                    ∧ exists01′ = TRUE
                    ∧ UNCHANGED ⟨y, ctrlStartActReply, ctrlEndActReply⟩;

! Übersetzung d. Transition zw. processed und init mit der Aktion actionReply
initToProcessed() ≜ ∧ state = "dequeued" ∧ exists01 = FALSE ∧ ¬sync
                    ∧ sync′ = TRUE
                    ∧ ctrlStartActReply′ = TRUE ! bereite die triggerless Trans vor
                    ∧ UNCHANGED ⟨qu,y, value01, exists01, ctrlEndActReply⟩;

actionWritey(cE0 : E0) ≜  ∧ state = "dequeued" ∧ exists01 = TRUE ∧ cE0 = "actor.actionWritey"
                          ∧ state′ = "processed"
                          ∧ y′ = value01
                          ∧ sync′ = FALSE
                          ∧ exists01′ = FALSE
                          ∧ UNCHANGED ⟨qu, ctrlStartActReply, ctrlEndActReply⟩;

! Übersetzung d. Transition zw. processed und init mit der Aktion actionReply
processedToInit() ≜  ∧ state = "processed" ∧ ctrlEndActReply = TRUE ∧ ¬sync
                     ∧ sync′ = TRUE
                     ∧ ctrlStartActReply′ = FALSE
                     ∧ UNCHANGED ⟨qu, y, value01, exists01, ctrlStartActReply⟩;

actionReply(cE0 : E0) ≜ ∧ cE0 = "actor.actionReply" ∧ state = "processed"
                        ∧ state′ = "init"
                        ∧ sync′ = FALSE
                        ∧ ctrlStartActReply′ = FALSE
                        ∧ ctrlEndActReply′ = TRUE
                        ∧ UNCHANGED ⟨qu, y, value01, exists01⟩;
END abstractActor
```

Abbildung 10.4: Der Prozesstyp Actor

Objektprozesses werden – falls notwendig – stattdessen nur mit der Aktion *permittedActionOfSD* des Prozesstyps Aktion *SequenceDiagram* gekoppelt, während alle übrigen Objektprozesse stottern. Dies wird in Abbildung 10.5 durch den Systemprozesstyp *ActorComposition*, der bereits in Kapitel 6 vorgestellt wurde, für das Sequenzdiagramm aus Abbildung 10.1 gezeigt. In diesem Prozesstyp wird neben den beiden Objektprozessen *abstActor* und *aT* der Prozess *sd* vom Typ *SequenceDiagram* instantiiert. Bei der Systemaktion *SetValueCallActor*, die den *setValue*-Aufruf an den Aktor modelliert, erkennt man, dass die *enqueue* Aktion des Prozesses *abstActor*, die *setValue*-Aktion des Prozesses *aT* und die des Prozesses *sd* miteinander gekoppelt werden. Die Systemaktionen *ActorInitToProcessed* und *ActorProcessedToInit* koppeln Aktionen, die für Transitionen eingeführt worden sind. Derartige Aktionen sind rein prozesslokal, sodass alle übrigen Prozesse stottern. In der Systemaktion *ActionReply* in Abbildung 10.5 wird gezeigt, wie die Aktionen *actionReply*, *permittedActionOfSD* und *enqueue* aller drei Prozesse gemeinsam schalten. Ebenso wird die Systemaktion *ActorWritey* für die Kopplung der Aktionen *actionWritey* und *permittedActionOfSD* verwendet.

```
PROCESS ActorComposition
 PROCESSES
  aT : abstractTask(k1); ! abstrakte Task
  abstActor : abstractActor();  ! abstrakter Aktor
  sd : SequenceDiagram({≪ ... ≫}); ! Prozess für Sequenzdiagramm

ACTIONS

SetValueCallActor(cEO : EO, value : INTEGER) ≙ ! SysAct to model
setValue call
     ∧ aT.actionSetValueCall(value, cEO)
     ∧ abstActor.enqueue(value)
     ∧ sd.permittedActionOfSD(cEO);

ActorDequeue(cEO : EO) ≙ ! SysAct modelling dequeuing in sensor process
     ∧ aT.stutter
     ∧ abstActor.dequeue(cEO)
     ∧ sd.permittedActionOfSD(cEO);

ActorWritey(cEO : EO) ≙ ! SysAct for action writey
     ∧ aT.stutter
     ∧ abstActor.actionWritey(cEO)
     ∧ sd.permittedActionOfSD(cEO);

ActorInitToProcess() ≙ ! SysAct for transition process
     ∧ aT.stutter
     ∧ abstActor.initToProcessed();
     ∧ sd.stutter;

ActorProcessedToInit() ≙ ! Transition processed to init
     ∧ aT.stutter
     ∧ abstActor.processedToInit()
     ∧ sd.stutter;

ActorActionReply(cEO : EO) ≙ ! SysAct to model actor reply
     ∧ aT.enqueue()
     ∧ abstActor.actionReply(cEO)
     ∧ sd.permittedActionOfSD(cEO);

END abstractComposition
```

Abbildung 10.5: Der Prozesstyp `ActorComposition` als Beispiel zur Komposition von Prozessen

10.2 Vollständige Spezifikation des Analysemusters Regler

In diesem Abschnitt wird eine vollständige Spezifikation des Analysemusters Regler vorgestellt. Durch dieses Muster werden die Charakteristika aller möglichen Lösungen, wie sie beim Entwurf erarbeitet werden, festgehalten. Erstmals werden hier auch die nicht-funktionalen Eigenschaften angegeben. In Abbildung 3.1 ist bereits das Klassendiagramm des Analysemusters für den abstrakten Regler angegeben, das hier aus Gründen der besseren Übersichtlichkeit in Abbildung 10.6 nochmal gezeigt wird. Die Klassen *abstractController* und *abstractSensor* sind über eine gerichtete Assoziation mit der Multiplizität 1 an den jeweiligen Assoziationsenden miteinander verbunden.

Zur Modellierung der Interaktionen der Objekte des Analysemusters ist in Abbildung 3.6 ein Sequenzdiagramm vorgestellt worden, das ein Objekt *O1* der Klasse *abstractController*, ein Objekt *O2* der Klasse *abstractSensor* und ein Objekt *03* der Klasse *abstractActor* enthält. Dieses Sequenzdiagramm wird nochmals in Abbildung 10.7 ausschnittweise gezeigt, weil es für das Verständnis dieses Abschnitts wichtig ist. Die Anzahl der gezeigten Objekte wird aber auf *O1* und *O2* beschränkt.

Für jede Klasse des Klassendiagramms aus Abbildung 10.6 ist ein Statechart-Diagramm modelliert worden. In Abbildung 10.8 wird nochmal das Statechart-Diagramm der Klasse *abstractController* gezeigt, das schon in Abbildung 3.3 vorgestellt wurde. Man erkennt die Regelschleife, die aus dem Zyklus der Zustände *wait*, *waitSensor*, *valuesComputed* und *waitActuator* besteht. Besonders interessant ist die Transition zwischen den Zuständen *waitSensor* und *valuesComputed*, die die Aktivität *compute* besitzt. Auch diese wird nochmals in Abbildung 10.9 gezeigt, um die Übersetzung der Aktionen vorzunehmen.

Als nächstes wird das Statechart-Diagramm der Klasse *abstractSensor*, das in Abbildung 10.10 gezeigt ist, beschrieben. Der Grundzustand *Init* wird nach der Erzeugung eines Objektes dieser Klasse erreicht, wobei die Initialisierung dieses Objektes erfolgt. Aus diesem Zustand kann ein Zustandsübergang in den Endzustand erfolgen, wenn das Sensorobjekt gelöscht wird. Eine Transition vom Zustand *Init* in den Zustand *processed* findet nach Ablauf der Wartezeit $t_{getValueCall}$ statt, wenn das Ereignis *getValueCall* auftritt und die Verarbeitung der ApplyFunctionAction *process* stattgefunden hat.

Nach Verstreichen der Wartezeit t_{return} kann eine Transition von *processed* in den Zustand *init* stattfinden, nachdem die ReplyAction *return* den Istwert aus dem Sensorobjekt zurück an das Objekt *O1* der Klasse *abstractController* übertragen hat. Der Zustand *Timeout* kann bei einer Zeitüberschreitung durch eine Transition wiederum aus dem Zustand *processed* betreten werden, sobald die Wartezeit $t_{ProcessTimeout}$ verstrichen ist. Dieser Zustand wird eingenommen, wenn eine Zeitüberschreitung aufgetreten ist. Anschließend findet aus diesem Zustand eine Transition in den Zustand *init* statt, nachdem die ReplyAction *return* mit dem Wert "Timeout" und dem Verstreichen der Wartezeit t_{return} ausgeführt worden ist. Der Zustand *error* kann im Fehlerfall durch eine Transition aus dem Zustand *processed* betreten werden, nachdem die Wartezeit $t_{ProcessError}$ verstrichen ist. Anschließend erfolgt eine Transition von *processed* in den Zustand *init*, nachdem die ReplyAction *return* mit dem Wert "Error" nach Verstreichen der Wartezeit t_{return} ausgeführt worden ist.

Das Statechart-Diagramm der Klasse *abstractActor* ist in Abbildung 10.2 angegeben.

Im Folgenden werden die nicht-funktionalen Eigenschaften des Analysemusters des untersuch-

Abbildung 10.6: Das Klassendiagramm eines abstrakten Controllers ohne Aktor

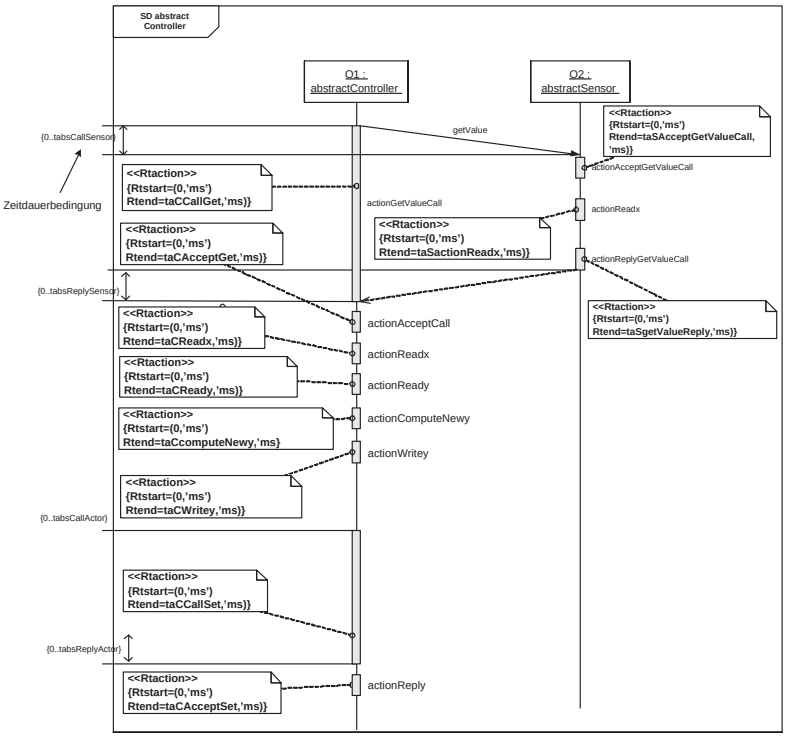


Abbildung 10.7: Das Sequenzdiagramm eines abstrakten Reglers

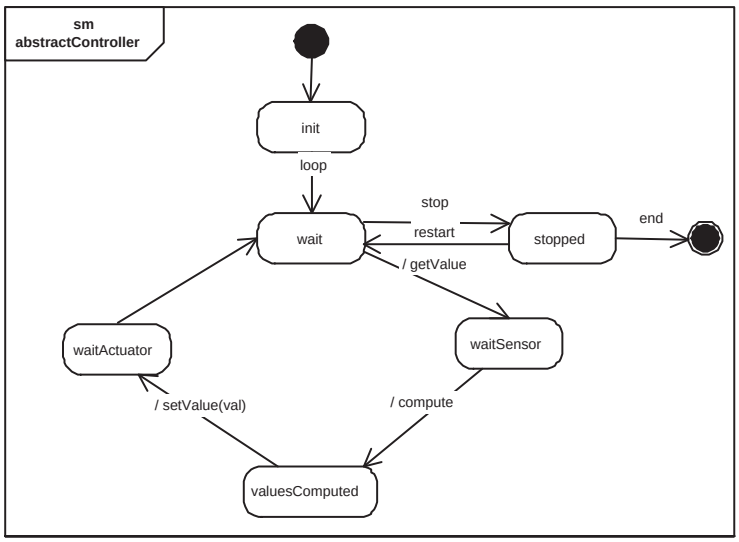


Abbildung 10.8: Das Statechart-Diagramm eines `abstractControllers`

ten Verfeinerungsmusters aufgeführt. Dies geschieht für Realzeitanforderungen[4] unter Verwendung von Parameter- bzw. Realzeitbedingungen, die aus Ungleichungen mit Parametern gebildet werden. Dabei werden entweder Zeiten für einzelne Verarbeitungsschritte definiert oder es werden Summen von Zeiten für ganze Folgen einzelner Verarbeitungsschritte angegeben. Eine typische nicht-funktionale Anforderung für einen Regler ist, dass die Regelschleife periodisch in bestimmten Fristen durchlaufen werden muss, wobei der Istwert zur Berechnung des Stellwertes verwendet

[4]Eine Anforderung ist eine Bedingung, die ein System erfüllen muss.

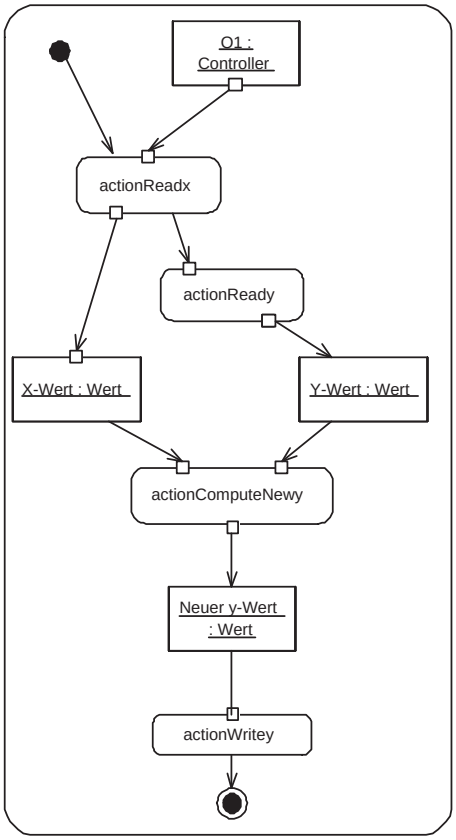


Abbildung 10.9: Die *Aktivität* `compute` zur Berechnung einer neuen Stellgröße

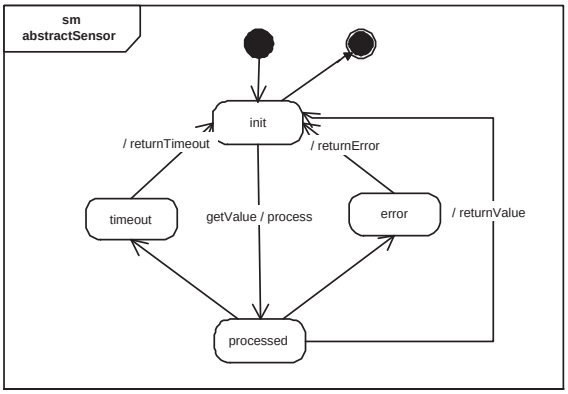


Abbildung 10.10: Das Statechart-Diagramm der Klasse *abstractSensor*

wird. Eine Verletzung der Fristen kann schwerwiegende Folgen haben.

Die Formalisierung der zulässigen nicht-funktionalen Eigenschaften verfolgt zwei Ziele:

1. Es werden Randbedingungen bzw. Charakteristiken für alle möglichen Lösungen angegeben. Damit wird ein Rahmen vorgegeben, in dem sich die zulässigen nicht-funktionalen Anforderungen bewegen dürfen. Bei der Anwendung eines Verfeinerungsmusters nach der Anforderungsanalyse kann leicht entschieden werden, ob ein Verfeinerungsmuster für die ermittelten nicht-funktionalen Anforderungen geeignet ist. Natürlich sind diese Randbedingungen nur sinnvoll, wenn sie zu den typischen nicht-funktionalen Anforderungen passen. Es ist davon auszugehen, dass diese typischen Anforderungen durch häufige Anwendung von klassischen

iterativen Softwareprozessen wie etwa ROPES [Dou99, KGC07] bekannt sind. Konrad geht sogar noch einen Schritt weiter und formuliert wiederum Muster für diese Realzeitbedingungen [KGC07]. Softwareprozesse zur Anwendung von Verfeinerungsmustern werden in Kapitel 14 behandelt.

2. Für den Nachweis von bestimmten Teilen der korrekten Verfeinerung eines Entwurfes wird auch ein formales, abstraktes Modell für die nicht-funktionalen Eigenschaften eines Analysemusters zur Verfügung gestellt.

Sicherheit/Zuverlässigkeit:
(S1) Mindestens jeder zweite Aufruf der *setValue*-Operation als auch der *getValue*-Operation ist nicht fehlerbehaftet (weder ein Fehler noch eine Zeitüberschreitung tritt auf) und führt zu einer korrekten Übertragung eines Wertes an das Sensor- bzw. Aktorobjekt. Dies ist zur Gewährleistung der Regelgüte erforderlich, da sonst der Regler nicht mehr sinnvoll auf die Regelstrecke eingreifen kann.

Umgebung: Die Umgebung blockiert die Aktionen *enqueue*, *getValueCall*, *setValueCall* des Reglerprozesses sowie die Aktionen *enqueue* und *return* des Aktors und des Sensors nie.

Lebendigkeit:
(L1) Auf jeden Aufruf der *getValue*-Operation des Reglers an den Sensor folgt irgendwann eine Antwort (Return) des Sensors, die den aktuellen Istwert überträgt, d. h. es ist kein Fehler oder eine Zeitverletzung aufgetreten. Ist dies nicht der Fall, rechnet der Regler mit veralteten Istwerten. Passiert dies oftmals hintereinander wird die Funktion der Regelung beeinträchtigt.
(L2) Auf jeden Aufruf der *setValue*-Operation des Reglers an den Aktor folgt irgendwann eine Antwort (Return) des Aktors, bei der zuvor der Stellwert aktualisiert worden ist. Sonst ist die Funktion der Regelung nicht mehr gewährleistet.

Parameterbedingungen: keine

Realzeit:
Die Realzeitanforderungen spezifizieren, welche Zeiten die einzelnen Verarbeitungsschritte des abstrakten Reglers benötigen. Zur Spezifikation werden Parameter verwendet, deren Benennung durch den Präfix t und den indizierten Namen der Action notiert wird. Weiterhin werden Ungleichungen verwendet, in denen Summen von Wartezeiten berechnet werden. Die konkreten Werte dieser Parameter werden durch Realzeitanforderungen festgelegt, die beim Requirements-Engineering ermittelt worden sind. Die Realzeitanforderungen des abstrakten Reglers werden später verwendet, um zu überprüfen, ob eine korrekte Verfeinerung durch die im Entwurf verwendete Entwurfsmuster-Kombination möglich ist.

Die Eigenschaften (R1)-(R4) definieren die Verarbeitungszeiten der Aktionen der Aktivität *compute* aus Abbildung 10.9. Da Verteilungsanforderungen vorliegen, müssen neben dem Fall der normalen Verarbeitung, bei dem ein korrekter Regelungszykus durchgeführt wird, auch das Verhalten im Fehlerfall bzw. bei Zeitverletzungen betrachtet werden. Mit der Realzeiteigenschaft (R5) wird die Antwortzeit für die Übertragung des Istwertes vom Sensor zum Regler durch eine Ungleichung angegeben. Der Fehlerfall zeichnet sich durch ein fehlerhaftes Verhalten während des Regelvorganges – etwa, dass der Sensor nicht erreichbar ist – aus. Durch (R6) werden mehrere Realzeitbedingungen durch Ungleichungen formuliert. Eine Zeitverletzung tritt auf, wenn die Übertragung eines Wertes nicht in den vorgesehenen Fristen erfolgt. Mit (R7) werden die Eigenschaften für den Fall einer Zeitverletzung angegeben. Die Eigenschaften (R8-R11) behandeln den Aktor. Mit der zentralen Eigenschaft (R12) wird die typische Anforderung an einen Regler nach einer definierten Periodendauer/Zykluszeit festgehalten.

(R1) $t_{aCReadx}$ (s. Abbildung 10.9 u. 10.7) bezeichnet die Zeit, die die Action *actionReadx* zum Auslesen des Attributes x benötigt.

(R2) Mit $t_{aCReady}$ (s. Abbildung 10.9 u. 10.7) wird die Zeit angegeben, die die Action *actionReady* zum Auslesen des Attributes y benötigt.
(R3) $t_{aCComputeNewy}$ (s. Abbildung 10.9 u. 10.7) notiert die Zeit, die die Action *actionComputeNewy* braucht, um einen neuen Stellwert zu errechnen.
(R4) $t_{aCWritey}$ (s. Abbildung 10.9 u. 10.7) bezeichnet die Zeit, die die Action *actionWritey* benötigt, um den neuen Stellwert in das Attribut y zu schreiben.
(R5) Für den Fall der normalen Verarbeitung, d. h. es tritt weder ein Fehler noch eine Zeitüberschreitung auf, wird mit $t_{abstSensor}$ die maximal zulässige Antwortzeit für die Ermittlung des Istwertes des Sensors definiert. Diese ist eine obere Schranke für die Summe aus der Zeit zur Verarbeitung des Aufrufes durch die Warteschlange, der Ermittlung des gegenwärtigen Istwertes und dessen Rücklieferung: $t_{abstSensor} \geq t_{aSenqueue} + t_{aSdequeue} + t_{aSactionReadx} + t_{aSgetValueReply}$.
Dabei bezeichnen $t_{aSenqueue}$, $t_{aSdequeue}$ die Wartezeiten zum Einfügen in die bzw. Entfernen eines Aufrufes aus der Warteschlange. Beide Zeiten können nicht aus den UML-Diagrammen abgelesen werden. Mit $t_{aSactionReadx}$ wird die Wartezeit zum Auslesen des abstrakten Sensors notiert, wie sie in Abbildung 10.9 als Constraint für die zugehörige UML-Action spezifiziert ist. Weiterhin wird mit $t_{aSgetValueReply}$ die Wartezeit zum Beantworten eines *getValueCall*-Aufrufes bezeichnet, die in Abbildung 10.9 für die UML-Action *actionReplyGetValueCall* angegeben wird.
(R6) Der Fehlerfall ist komplizierter als die normale Verarbeitung, da Fehler in allen UML-Actions – nämlich beim Einfügen und Entfernen des Aufrufs bzw. beim Lesen des Wertes – in sämtlichen Kombinationen auftreten können. Damit die Regelung nicht für einen zu großen Zeitraum ausfällt und Instabilitäten auftreten gilt, dass die Summe der Zeit zur Ermittlung des gegenwärtigen Istwertes und dessen Rücklieferung im Fehlerfall höchstens so groß wie die gesamte Verarbeitungszeit des abstrakten Sensors ist. Die maximale Wartezeit für den Fehlerfall $t_{MaxError}$ wird wie folgt definiert:
$t_{MaxError} = t_{aSenqueueError} + t_{aSdequeueError} + t_{aSactionReadxError} + t_{aSReplyGetValueCallError}$
Dabei bezeichnet $t_{aSenqueueError}$ die Wartezeit für einen Fehler, der beim Einfügen des Aufrufes in die Warteschlange auftritt. Mit $t_{aSdequeueError}$ wird ein Fehler notiert, der beim Entfernen des Aufrufs auf der Warteschlange auftritt. Für die Wartezeit, die bei einem Fehler beim Auslesen des Wertes durch die UML-Action *actionReadx* auftritt, wird $t_{aSactionReadxError}$ angegeben. Schließlich notiert $t_{aSReplyGetValueCallError}$ die Wartezeit für die Weiterleitung eines fehlerhaften *getVallCall*-Aufrufes. Es werden Vereinfachungen vorgenommen, um die Komplexität der Berechnungen für Realzeitbedingungen zu reduzieren. Die Annahmen sind praktisch sinnvoll, denn beim Einfügen oder Entfernen von Aufrufen aus der Warteschlange im normalen bzw. im Fehlerfall sind keine Unterschiede zu erwarten. Ähnliches gilt für das Auslesen eines Wertes. Daher wird Folgendes gefordert:
$t_{aSenqueueError} \leq t_{aSenqueue}$
$t_{aSdequeueError} \leq t_{aSdequeue}$
$t_{aSactionReadxError} \leq t_{aSactionReadx}$
$t_{aSReplyGetValueCallError} \leq t_{aSReplyGetValueCall}$
(R7) Auch für den Fall einer Zeitüberschreitung wird gefordert, dass die Summe der Zeit zur Ermittlung des gegenwärtigen Istwertes und dessen Rücklieferung so groß ist wie die gesamte Verarbeitungszeit des abstrakten Sensors. Auch bei Zeitüberschreitungen können alle Kombinationen auftreten. $t_{MaxTimeout}$, durch das die maximale Wartezeit im Timeout-Fall notiert wird, ist wie folgt definiert:
$t_{MaxTimeout} = t_{aSenqueueTimeout} + t_{aSdequeueTimeout} + t_{aSactionReadxTimeout} + t_{aSReplyGetValueCallTimeout}$
Die Wartezeit $t_{aSenqueueTimeout}$ bezeichnet eine Zeitverletzung beim Einfügen in die Warteschlange. Mit $t_{aSdequeueTimeout}$ wird die Wartezeit beim Entfernen aus der Warteschlange notiert. Durch $t_{aSactionReadxTimeout}$ wird in diesem Kontext eine Wartezeit angegeben, die bei einer Zeitüberschreitung entsteht, welche beim Auslesen des Istwertes auftritt. Abschließend bezeichnet $t_{aSReplyGetValueCallTimeout}$ die Wartezeit für eine Zeitüberschreitung bei der Rücklieferung des Istwertes an das Objekt *abstractController*. Es wird Folgendes gefordert:

$t_{aSenqueueTimeout} \leq t_{aSenqueue}$
$t_{aSdequeueTimeout} \leq t_{aSdequeue}$
$t_{aSactionReadxTimeout} \leq t_{aSactionReadx}$
$t_{aSReplyGetValueCallTimeout} \leq t_{aSReplyGetValueCall}$
(R8-R11) Die Wartezeiten für den Zugriff auf den Aktor werden nicht weiter betrachtet. Allerdings sind diese denen für den Zugriff auf den Sensor ähnlich.
(R12) Die Gesamtzeit für den Durchlauf einer Regelperiode setzt sich aus den Zeiten für den Start der Regelperiode r, der Kommunikation mit dem Sensor $t_{abstSensor}$, der Verarbeitungszeit der eingelesenen Werte $t_{aCactionReadx}$, $t_{aCactionReady}$, $t_{aCactionComputeNewy}$, $t_{acCactionWritey}$ und der Kommunikationszeit mit dem Aktor $t_{abstActor}$ zusammen.

10.3 Übersetzung des Analysemusters Regler nach cTLA

In diesem Abschnitt wird die Übersetzung des Analysemusters Regler nach cTLA beschrieben, die – wie in Abbildung 10.11 gezeigt – aus den Prozessen *abstractController*, *abstractSensor* und *abstractActor* besteht. Die Abbildung zeigt auch die Kopplung zur Übersetzung von *CallOperation-* und *ReplyActions* in verschiedenen Prozessen und die in diese eingeführten cTLA-Aktionen. Zur Darstellung der Kopplung durch eine Systemaktion werden die in den Prozessen enthaltenen Aktionen durch Linien verbunden. Ein ausgefüllter Kreis auf einer Linie bedeutet, dass die zugehörige Aktion Bestandteil der Kopplung ist. Man erkennt, dass die Aktion *actionGetValueCall* des Prozesses *abstractController* mit der Aktion *enqueue* des Prozesses *abstractSensor* gekoppelt ist. Dadurch wird eine *CallOperationAction* formalisiert. Weiterhin wird die Aktion *actionGetValueReply* des Prozesses *abstractSensor* mit der Aktion *enqueue* des Prozesses *abstractController* gekoppelt. Zunächst wird die Übersetzung der Klasse *abstractController* und anschließend die der Klasse *abstractSensor* betrachtet, die durch genau einen eigenständigen cTLA-Prozess übersetzt werden. Für die Übersetzung der Klasse *abstractActor* wird auf Abschnitt 10.1 verwiesen. Abschließend wird die Transformation des Sequenzdiagramms des Analysemusters aus Abbildung 10.7 in einen cTLA-Systemprozess angegeben. Das cTLA-System, das bei der Übersetzung der Klassen des Analysemusters entsteht, wird im Folgenden als Grobsystem des Verfeinerungsmusters bezeichnet.

10.3.1 Übersetzung der Klasse *abstractController* nach cTLA

Im Folgenden wird der Header des Prozesstyps *abstractController*, der zur Übersetzung der Klasse *abstractController* aus Abbildung 10.6 erzeugt worden ist, angegeben. Für die Attribute x und y dieser Klasse wird eine Zustandsvariable mit dem gleichen Bezeichner x bzw. y und dem gleichen Datentyp Real in den cTLA-Prozess, der die Klasse *abstractController* übersetzt, eingeführt, um den Datenzustand eines Objektes dieser Klasse zu repräsentieren.

Zur Übersetzung von Kontrollflüssen und Objektknoten werden jeweils eigenständige Zustandsvariablen eingeführt.

Für die Kontrollflüsse der in Abbildung 10.9 beschriebenen Aktivität *compute* werden zwei Boolesche Zustandsvariablen, nämlich *ctrlStartActReadx* und *ctrlStartActReadx*, in den Prozess *abstractController* aufgenommen, die jeweils angeben, ob der entsprechende Kontrollfluss existiert. Der Kontrollfluss vom Startknoten zur Action *actionReadx* (s. Abbildung 10.9) wird durch die Zustandsvariable *ctrlStartActReadx* repräsentiert. Für den Kontrollfluss zwischen den beiden Actions *actionReadx* und *actionReady* wird die Zustandsvariable *ctrlActReadxActReady* eingeführt.

```
PROCESS abstractController

IMPORT
      Sequence;

CONSTANT
 k : Integer;

VARIABLES
  x,y : Real; ! y für den Stellwert und x für den Istwert
  qu : Sequence of Message; ! Zustand der Ereigniswarteschlange
```

```
sync : Boolean; ! Synchronisationsvariable für nebenläufigen Zugriff
ctrlStartActReadx, ctrlActReadxActReady : Boolean; ! existiert KF
state : {"init", "waitSensor", "getReturnReceived", "getReturnDequeued","valuesComputed",
"waitActuator", "setReturnReceived", "stopped", "wait"}; ! Kontrollzustand
exists01, exists02, exists03 : Boolean; ! Existenz der Objektflüsse 01, 02 u. 03
value01, value02, value03 : Real; ! Übersetzung d. Objektknoten 01, 02, 03
```

Um einen Objektknoten, der über Eingangs- und Ausgangspins mit einer Action durch Objektflüsse verbunden ist, zu übersetzen, werden zwei separate Zustandsvariablen in den Prozess aufgenommen. Deshalb werden die Zustandsvariablen, die zur Übersetzung der Objektknoten *x-Wert*, *y-Wert* und *Neuer y-Wert* aus dem Aktivitätsdiagramm der Abbildung 10.9 eingeführt worden sind, angegeben. Hierbei wird angenommen, dass jeder Objektknoten als zugeordneten Typ eine Klasse besitzt, die nur über ein einziges Attribut eines Basisdatentyps verfügt. Dadurch wird ein Objektfluss zu einem Datenfluss. Zunächst wird eine weitere Boolesche Zustandsvariable eingeführt, die im Beispiel den Bezeichner *existsO1* besitzt, um anzugeben, ob der entsprechende Objektknoten mit einem gültigen Datenwert gefüllt ist.

Die zweite Zustandsvariable mit dem Bezeichner *valueO1* vom Typ Real, die in dem Objektknoten abgelegt wird, nimmt den Wert dieses Datums auf. Die drei für die Objektknoten neu eingeführten Zustandsvariablen *valueO1*, *valueO2* und *valueO3* bilden eine Erweiterung des Datenzustands.

Durch die Zustandsvariablen x und y werden die Attribute repräsentiert. Mit den Zustandsvariablen *sync* und *qu* wird die Warteschlange und deren Synchronisation übersetzt. Der Kontrollzustand eines Statechart-Diagramms wird durch die Zustandsvariable *state* übersetzt. Die Booleschen Zustandsvariablen *ctrlStartActReadx*, *ctrlActReadxActReady*, *existsO1*, *existsO2*, *existsO3*

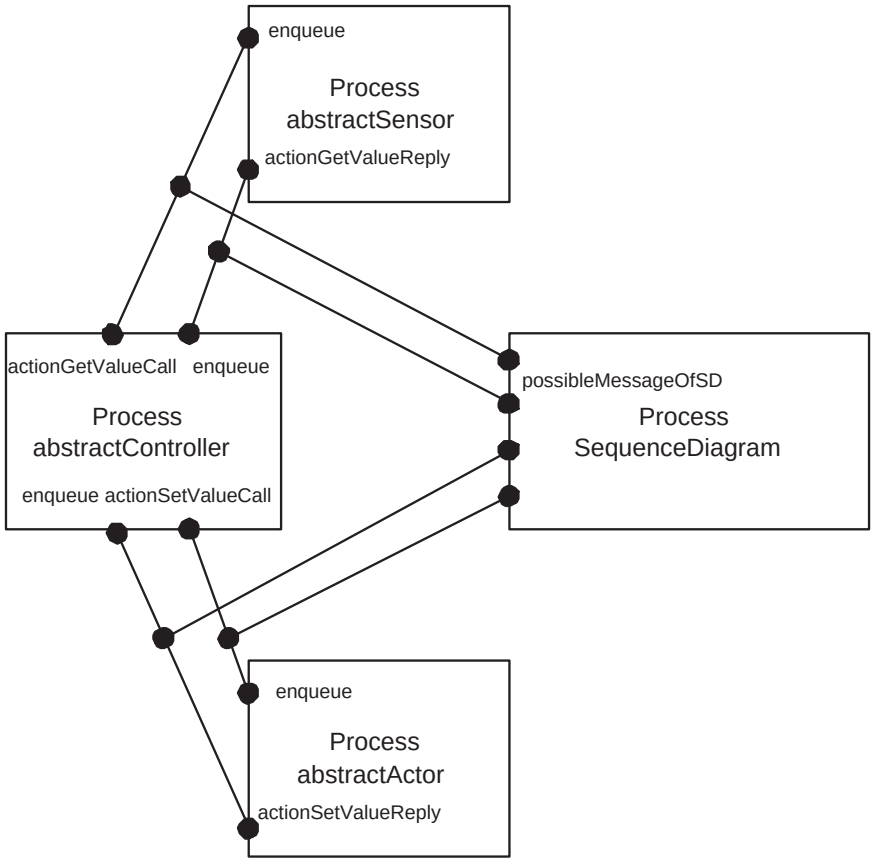


Abbildung 10.11: Die Prozesse des Analysemuster-Systems und deren Kopplung

repräsentieren die Existenz eines Kontroll- bzw. Objektflusses. Weiterhin repräsentieren die Zustandsvariablen *valueO1*, *valueO*2 und *valueO3* die Inhalte der Objektflüsse.

```
INIT ≜ ∧ x = 0
       ∧ y = 0
       ∧ sync = FALSE
       ∧ qu = ≪ ≫
       ∧ state = "init"
       ∧ ctrlStartActReadx = TRUE
       ∧ ctrlActReadxActReady = FALSE
       ∧ existsO1 = FALSE
       ∧ existsO2 = FALSE
       ∧ existsO3 = FALSE
       ∧ valueO1 = 0
       ∧ valueO2 = 0
       ∧ valueO3 = 0;
```

Das Initialisierungsprädikat `INIT` beschreibt die Menge der Initialzustände des Prozesses *abstractController*, die nur ein Element – nämlich den Zustand *init* des Statechart-Diagramms – enthält. Zunächst wird der Zustandsvariablen *state* der Wert "init" zugewiesen. Weiterhin werden in diesem Initialisierungsprädikat die übrigen Zustandsvariablen für den Zustand der Ereigniswarteschlange, für den Datenzustand sowie für die Kontroll- bzw Objektflüsse initialisiert. Hierzu werden die Zustandsvariablen x und y auf 0 sowie *qu* auf die leere Sequenz << >> gesetzt.

Weiterhin wird der Zustandsvariable *ctrlStartActReadx* der Wert TRUE zugewiesen. Die Zustandsvariablen *ctrlActReadxActReady*, *existsO1*, *existsO2* und *existsO3* werden auf FALSE gesetzt, was bedeutet, dass keiner der Kontrollflüsse existiert. Die Zustandsvariablen *valueO1*, *valueO2* und *valueO3* werden auf 0 gesetzt, da im Initialzustand keiner der Objektknoten mit einem Wert belegt ist.

Für die Objektknoten O_2 und O_3 der Aktivität *compute* werden zusätzlich die Zustandsvariablen *existsO2*, *valueO2*, *existsO3* und *valueO3* aufgenommen.

Auch die Pufferung von Ereignissen mittels einer Warteschlange zur weiteren Verarbeitung, wie sie in Abschnitt 3.4.3 bei der Vorstellung der Zustandsmaschine erläutert worden ist, muss berücksichtigt werden. Hierfür wird die Zustandsvariable *qu* vom Datentyp *Sequence* im Prozess verwendet.

Der Kontrollzustand des Statechart-Diagramms der Klasse *abstractController* wird in die cTLA-Zustandsvariable mit dem Bezeichner *state* übersetzt, die alle Zustandsnamen des zugehörigen Statechart-Diagramms annehmen kann. Zusätzlich zu den in Abbildung 10.8 ersichtlichen Zuständen gibt es Zustände, die für das Einfügen und Entfernen von Aufrufen aus der Warteschlange erforderlich sind.

Zum Einfügen eines Ereignisses in die Warteschlange wird die *enqueue* Aktion verwendet. Um die Synchronisation eines Objekts bei nebenläufigem Zugriff zu modellieren, wird die Zustandsvariable *sync* eingeführt, welche eine Blockierung des cTLA-Prozesses des *abstractController* Objektes, das die Zustandsmaschine repräsentiert, vornehmen kann.

Nun wird die Übersetzung der Transitionen des Statechart-Diagramms des Beispielmusters betrachtet. Der Prozess in Abschnitt A.1 des Anhangs enthält die zur Übersetzung sämtlicher *Actions* und Transitionen eingeführten cTLA-Aktionen.

Transitionen eines Statecharts modifizieren den Kontrollzustand eines Objektes. Die Übersetzung einer Transition eines Objektes erfolgt im Rahmen der Beispielmuster durch eine eigenständige, parameterlose cTLA-Aktion, d. h. die Semantik einer Transition aus einem Statechart-Diagramm wird auf der Basis einer cTLA-Aktion festgelegt, die zu einer Änderung der Zustandsvariable *state* führt. Grundsätzlich wird bei der Übersetzung zwischen *triggerless* Transitionen und Transitionen, die mit einem Trigger behaftet sind, unterschieden.

Unten wird die Übersetzung der triggerless Transition *waitSensorToValuesComputed* aus dem Statechart-Diagramm der Abbildung 10.8 gezeigt. In der Vorbedingung erfolgt im ersten Konjunkt die Prüfung, ob der gegenwärtige Kontrollzustand mit dem Ausgangszustand "valueDequeued" der Transition übereinstimmt. Im Effekt der Aktion wird im zweiten Konjunkt die Zustandsvariable

state, die den Kontrollzustand eines Objektes repräsentiert, auf den Zielzustand "valuesComputed" der Transition gesetzt. Den Zustandsvariablen, die angeben, ob ein Kontrollfluss für eine Action, welche mit einem Startknoten verbunden ist – hier ist das die Zustandsvariable *ctrlStartActReadx* – wird im dritten Konjunkt der Wahrheitswert TRUE zugewiesen. Alle übrigen Zustandsvariablen der Aktion bleiben unverändert. Die Aktionen *waitToStopped* und *stoppedToWait* sind Übersetzungen für Transitionen, die dem gleichen Prinzip unterliegen.

```
waitSensorToValuesComputed ≜ ! Transition waitSensor → valuesComputed
   ∧ state = ''valueDequeued'' ! Prüfung des aktuellen Kontrollzustandes
   ∧ state' = ''ValuesComputed'' ! Der Folgezustand wird gesetzt
   ∧ ctrlStartActReadx' = TRUE !
   ∧ UNCHANGED ⟨x, y⟩
   ∧ UNCHANGED ⟨qu, sync, ctrlActReadxActReady⟩
   ∧ UNCHANGED ⟨exists01, exists02, exists03, value01, value02, value03⟩;
```

Die Übersetzung der mit einem Trigger behafteten Transition *loop*, die beim Betreten der Regelschleife ausgeführt wird, wird unten betrachtet. Im ersten Konjunkt der Vorbedingung wird geprüft, ob der gegenwärtige Kontrollzustand "init" ist. Im zweiten Konjunkt erfolgt eine Prüfung, um festzustellen, ob am Anfang der Warteschlange eine Nachricht vom Typ "loop" enthalten ist. Im Effekt der Aktion wird die Nachricht aus der Ereigniswarteschlange im dritten Konjunkt entfernt. Das Setzen des Kontrollzustandes auf den Zielzustand "wait" der Transition erfolgt im vierten Konjunkt. Die letzten beiden Konjunkte geben an, dass die übrigen Zustandsvariablen nicht verändert werden.

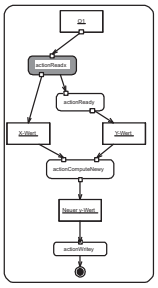
```
loop ≜ ∧ state = ''init''
       ∧ head(qu) = ''loop''
       ∧ qu' = tail(qu)
       ∧ state1' = ''wait''
       ∧ ctrlStartActReadx = TRUE
       ∧ UNCHANGED ⟨x, y, sync,ctrlStartActReadx, ctrlActReadxActReady⟩
       ∧ UNCHANGED ⟨exists01, value01, exists02, value02, exists03, value03⟩;
```

Für die hier beschriebene Übersetzung einer Transition gibt es aus Gründen der Vereinfachung eine Ausnahme. Wenn die Aktivität einer Transition nur aus einer einzelnen Action besteht, welche nur einen eingehenden und einen ausgehenden Kontrollfluss besitzt, werden die beiden Aktionen, die für die Übersetzung der Transition und der Action eingeführt worden sind, in der Übersetzung der Transition zusammengefasst. Ein Beispiel hierfür ist die Aktivität aus Abbildung 10.2, die nur ein Action besitzt. Dies ist möglich, da die Action – wenn die Transition stattfindet – sofort und ohne Abhängigkeiten von anderen Actions ausgeführt wird. Da diese Situation bei der Modellierung sehr häufig auftritt, soll diese zusätzliche Komplexität durch unnötige Zustandsvariablen für Kontrollflüsse bei der Übersetzung vermieden werden. Dieser Ausnahmefall wird später bei der Übersetzung der UML-Diagramme der Entwurfsmuster angewendet. Für die Übersetzung jeder UML-Action dieses Analysemusters wird eine neue cTLA-Aktion eingeführt. Diese Aktionen erhalten Parameter zur Kopplung mit dem Prozess *SequenceDiagram*, um zu prüfen, ob das Schalten zulässig ist. Zunächst werden *lese-* und *schreiborientierte Actions* des Beispielmusters behandelt. *Lese-* und *schreiborientierte Actions* werden durch cTLA-Aktionen übersetzt, die die Zustandsvariablen, welche die Attribute oder Objektflüsse eines Objektknotens repräsentieren, modifizieren. In jeder dieser Aktionen wird im ersten Konjunkt durch Auswertung der Zustandsvariable *sync* geprüft, ob das zugehörige Objekt bereits durch einen anderen Aufruf blockiert ist.

Nachfolgend ist die Aktion *actionReadx* aus dem Aktivitätsdiagramm der Aktivität *compute* spezifiziert. Das erste Konjunkt der Aktion dient der Prüfung der Zustandsvariablen *sync*, um sicherzustellen, dass keine Verletzung der RTC-Semantik durch eine nicht zulässige nebenläufige Modifikation erfolgt. Dies ist nur bei den Übersetzungen der Actions notwendig. Im zweiten Konjunkt der Vorbedingung wird geprüft, ob der eingehende Kontrollfluss existiert. Dies ist der Fall, wenn die Zustandsvariable *ctrlStartActReadx* TRUE ist. Der Parameter *currentEventOccurence* wird im dritten Konjunkt der Aktion mit dem Wert "O1.actionReadx" belegt, um sicherzustellen,

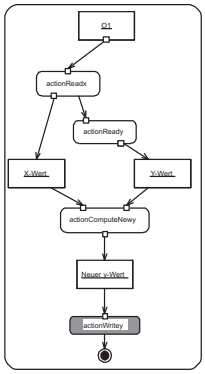
dass die Aktion im Rahmen des Traces des Sequenzdiagramms schalten darf. Die Zustandsvariable *valueO1* wird im vierten Konjunkt auf den Wert der Zustandsvariable x gesetzt, die das Attribut x repräsentiert. Im fünftem Konjunkt wird die Zustandsvariable *ctrlStartActReadx* für den eingehenden Kontrollfluss auf FALSE gesetzt, um ein nochmaliges Schalten der Aktion zu verhindern. Im sechsten Konjunkt wird die Zustandsvariable *existsO1* für den ausgehenden Objektfluss auf TRUE gesetzt. Der Zustandsvariable *ctrlActReadxActReady* für den ausgehenden Kontrollfluss wird im siebten Konjunkt hingegen der Wahrheitswert TRUE zugewiesen. Die übrigen Zustandsvariablen bleiben unverändert. Die Übersetzung der Aktion *actionReady* erfolgt nach dem gleichen Prinzip.

```
actionReadx(currentEventOccurence : String) ≜
    ∧ ¬sync ! ¬ bedeutet nicht durch RTC zulässig bzw. nicht blockiert
    ∧ ctrlStartActReadx = TRUE ! Pruefung der eingehenden ZV
    ∧ currentEventOccurence = "O1.actionReadx"
    ∧ valueO1′ = x ∧ ! Setzen der ZV eines ausgehenden OF
    ∧ ctrlStartActReadx′ = FALSE ! Ruecksetzen der ZV
    ∧ existsO1′ = TRUE ! Setzen der ausgehenden ZV
    ∧ ctrlActReadxActReady′ = TRUE ! Setzen d. ZV d. ausg. KF
    ∧ UNCHANGED state
    ∧ UNCHANGED ⟨x, y, qu, sync⟩
    ∧ UNCHANGED ⟨existsO2, existsO3, valueO2, valueO3⟩;
```



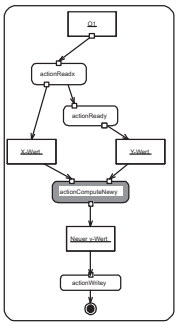
Bei der Übersetzung der WriteAttributeAction *actionWritey*, die im Anschluss gezeigt ist, wird in der Vorbedingung der Aktion *actionWritey* im zweiten Konjunkt ermittelt, ob der Objektfluss O_3 existiert, indem der Wert der zugehörigen Zustandsvariablen *existsO3* geprüft wird. Der Parameter *currentEventOccurence* wird mit dem Bezeichner des Ereignisauftritts "O1.actionWritey" belegt. Die Zustandsvariable *existsO3* für den eingehenden Kontrollfluss wird im vierten Konjunkt auf FALSE gesetzt, damit die Aktion nicht ein weiteres Mal schalten darf. Im fünften Konjunkt wird die Zustandsvariable für y auf den Wert der Zustandsvariable *valueO3* des Objektknotens gesetzt. Im sechsten Konjunkt wird die Zustandsvariable *state* auf den Nachfolgezustand *valueComputed* gesetzt, damit die nachfolgenden Aktionen schaltbereit werden.

```
actionWritey(currentEventOccurrence : String) ≜ ! Aktion schreibt
in ZV des Attributes y
    ∧ ¬sync ! durch RTC zulässig ?
    ∧ existsO3 = TRUE  ! Prüfung der Boole. ZV des eing. OF
    ∧ currentEventOccurrence = ''O1.actionWritey''
    ∧ existsO3′ = FALSE  ! Ruecksetzen der Boole. ZV des eing. OF
    ∧ y′ = valueO3 ! Setzen der ZV für y
    ∧ state′ = ''valueComputed'' ! Setzen des nachfolg. Kontrollzust.
    ∧ UNCHANGED ⟨x, qu, sync, state⟩
    ∧ UNCHANGED ⟨ctrlStartActReadx⟩
    ∧ UNCHANGED ⟨ctrlActReadxActReady⟩
    ∧ UNCHANGED ⟨existsO1, existsO2, valueO1, valueO2, valueO3⟩;
```



Jetzt soll die Übersetzung der ApplyFunctionAction *actionComputeNewy*, die unten aufgeführt ist, betrachtet werden. Die Idee zur Übersetzung besteht darin, eine cTLA-Funktion zur Berechnung des Funktionswertes auf der Basis der an den Eingabepins von Objektflüssen bereitgestellten Argumentwerte zu verwenden. Dies ist hier die im Folgenden aufgeführte cTLA-Funktion *computeNewy*. Im Rahmen der Vorbedingung erfolgt die Prüfung auf die Existenz der eingehenden Kontrollflüsse mittels Auswertung der Zustandsvariablen *existsO1* und *existsO2* im zweiten und dritten Konjunkt. Der Aktionenparameter *currentEventOccurence* wird mit dem Bezeichner des Ereignisauftritts "O1.actionComputeNew" im vierten Konjunkt belegt, um das Schalten dieser Aktion zu einem durch das Sequenzdiagramm zugelassenen Zeitpunkt zu ermöglichen. Der Funktion *computeNewy* werden als Argumente die Werte der Zustandsvariablen *valueO1, valueO2* übergeben. Das Ergebnis des Funktionsaufrufs der Funktion *computeNewy* wird der Zustandsvariable *valueO3* im vierten Konjunkt zugewiesen. Abschließend werden die Zustandsvariablen *existsO1*, *existsO2*, *existsO3* in den Konjunkten fünf bis sieben auf TRUE bzw. FALSE gesetzt. Die übrigen Zustandsvariablen werden durch die Aktion nicht verändert.

```
actionComputeNewy(currentEventOccurence : String) ≜  ! Aktion
berechnet neuen Funktionswert
    ∧ ¬sync ! durch RTC zulässig ?
    ∧ exists01 = TRUE
    ∧ exists02 = TRUE
    ∧ currentEventOccurence = ''01.actionComputeNewy''
    ∧ value03' = computeNewy[value01, value02]
    ∧ exists01' = FALSE
    ∧ exists02' = FALSE
    ∧ exists03' = TRUE
    ∧ UNCHANGED ⟨qu, sync, state⟩
    ∧ UNCHANGED ⟨ctrlActReadxActReady⟩
    ∧ UNCHANGED ⟨ctrlStartActReadx⟩
    ∧ UNCHANGED ⟨value01, value02⟩;
```



Nun soll die Übersetzung der unten angegebenen CallOperationAction *actionGetValueCall* betrachtet werden. In der Vorbedingung der Aktion *actionGetValueCall* wird geprüft, ob die Zustandsvariable *sync* auf FALSE gesetzt und das Objekt nicht blockiert ist. Im zweiten Konjunkt wird ausgewertet, ob sich das Objekt in dem Kontrollzustand "wait" befindet. Hierbei sieht man die Zusammenfassung einer Transition und einer Action in eine cTLA-Aktion. Im Rahmen der Parameterbelegung werden alle Aktionenparameter, die für einen Operationsaufruf eingeführt worden sind, mit den Werten der Zustandsvariablen belegt, die die eingehenden Objektflüsse repräsentieren. Der Aktionsparameter *currentEventOccurence* wird mit dem Wert "O1.actionGetValueCall" im dritten Konjunkt belegt, um die notwendigen Informationen für die Auswertung des Traces des Sequenzdiagramms bereitzustellen. Im Effekt der Aktion erfolgt das Setzen der Zustandsvariable *sync* im fünften Konjunkt auf TRUE, um das Objekt zu blockieren, damit die RTC-Semantik gewährleistet ist. Weiterhin wird in der Aktion die Zustandsvariable *state* auf den Nachfolgezustand "waitSensor" gesetzt.

```
actionGetValueCall(currentEventOccurence : String) ≜  !
parameterlose Aktion getValueCall für Operationsaufruf
    ∧ ¬sync
    ∧ state = ''wait''
    ∧ currentEventOccurence = ''01.actionGetValueCall''
    ∧ state' = ''waitSensor''
    ∧ sync' = TRUE
    ∧ UNCHANGED ⟨x, y, qu, ctrlStartActReadx, ctrlActReadxActReady⟩
    ∧ UNCHANGED ⟨exists01, exists02, exists03, value01, value02, value03⟩;
```

Die im Folgenden angegebene Aktion *enqueue* nimmt ein eingehendes Ereignis mit seinen Werten über einen Aktionenparameter entgegen, um es in die Warteschlange einzufügen. Die Warteschlange besitzt den Typ SEQUENCE OF REAL. Dies ist der Fall beim Einfügen von Operationsaufrufen und Antworten auf Operationsaufrufe. Für ein durch eine Nachricht übertragenes Ereignis wird eine Aktion *enqueue* mit dem Aktionenparameter *val* vom Typ REAL bereitgestellt.

```
enqueue(val : Real) ≜  ! Aktion zum Einfuegen von Ereignissen in
die Ereigniswarteschlange
    ∧ state = ''waitSensor'' ∨ state = ''waitActuator''
    ∧ state' = IF state = ''waitSensor''
               THEN ''getEnqueued''
               ELSE IF state = ''waitActuator''
               THEN ''returnEnqueued''
               ELSE state
    ∧ qu' = IF state = ''waitSensor''
            THEN Append(qu, value)
            ELSE qu
    ∧ UNCHANGED ⟨x, y, qu, ctrlStartActReadx, ctrlActReadxActReady⟩
    ∧ UNCHANGED ⟨exists01, exists02, exists03, value01, value02, value03⟩;
```

Mit der unten angegebenen *dequeue* Aktion werden Operationsaufrufe mit ihren Werten aus der Warteschlange entfernt. In der Aktion wird durch eine Fallunterscheidung geprüft, ob im Vorgängerzustand *state* =″ *getEnqueued*″ gilt. In diesem Fall befindet sich in der Warteschlange ein neuer *getValue*-Aufruf. Ist dies nicht der Fall ist die Warteschlange durch eine Rückantwort gefüllt worden, sodass die Verarbeitung von vorne beginnen kann.

```
dequeue ≜
       ∧ state = "getEnqueued" ∨ state = "returnEnqueued"
       ∧ state′ = IF state = "getEnqueued" ! getValue Call in Warteschlange
                  THEN "valueDequeued" ! Zielzustand valueDequeued
                  ELSE "init" ! Sonst Antwort zurück nach init
       ∧ x′ = Head(qu).val
       ∧ qu′ = Tail(qu)
       ∧ sync′ = FALSE
       ∧ UNCHANGED ⟨x, y, ctrlStartActReadx, ctrlActReadxActReady⟩
       ∧ UNCHANGED ⟨exists01, exists02, exists03, value01, value02, value03⟩;
```

10.3.2 cTLA-Übersetzung der Realzeitanforderungen für die Klasse *abstractController*

Jede für eine Transition eingeführte Aktion ist mit den Zeiten entsprechend der Modellierung aus den Statecharts zu versehen. Zur Spezifikation dieser Wartezeiten wird der Prozesstyp *abstContrTimes* aus Abbildung 10.12 verwendet. Zunächst werden in dem Prozesstyp sämtliche Aktionen deklariert, für die Wartezeiten spezifiziert werden. Die Wartezeiten sämtlicher Aktionen des Prozesstyps *abstractController* sind in diesem Prozesstyp angegeben. Aktionen, die Transitionen modellieren – wie etwa *waitSensorToValuesComputed* –, schalten unmittelbar. Anschließend werden die maximalen, persistenten Wartezeiten für jede Aktion, die zu einer UML-Action gehört, mit einem P MAX TIME Konstrukt angegeben.

10.3.3 Übersetzung der Klasse *abstractSensor*

Der cTLA-Prozess aus Abbildung 10.13 gibt die Übersetzung der Klasse *abstractSensor* an, die die Zustandsvariablen *x*, *sSensor* und *qu* enthält. Die Zustandsvariable *x* modelliert den Istwert. Durch die Zustandsvariable *sSensor* wird der Kontrollzustand beschrieben. Schließlich modelliert die Zustandsvariable qu die Warteschlange der Klasse *abstractSensor*. Mit dem Initialisierungsprädikat `INIT` werden die Zustandsvariablen für den Daten- und Kontrollzustand und den Zustand der Ereigniswarteschlange initialisiert. Die Aktion *enqueue* nimmt einen Aufruf der CallOperation-Action *getValueCall* des *abstractController* Objektes entgegen, während die Aktion *dequeue* einen Aufruf aus der Warteschlange entfernt. Die Aktion *process* modelliert einen Verarbeitungsvorgang im Sensorobjekt. Auch die Übersetzung der ReplyAction *actionGetValueCallReply* aus dem Sensorobjekt *O2* wird in Abbildung 10.13 gezeigt. Diese Aktion fasst wiederum die Übersetzung einer Action und einer Transition zusammen. Bei dieser ReplyAction werden zusätzliche Aktionenparameter aufgenommen, um das Ergebnis des Operationsaufrufes zurückzuliefern. In der Vorbedingung wird zunächst geprüft, ob der eingehende Kontrollfluss existiert. Anschließend wird durch die Auswertung der Zustandsvariablen des Kontrollzustandes *state* auf den Wert "processed" gesetzt. Speziell wird der Aktionenparameter *valueO1_Pin1* im dritten Konjunkt mit dem Wert der Zustandsvariable *x* belegt. Der Parameter *currentEventOccurence* wird mit dem Bezeichner "O2.actionGetValueCallReply" des zugehörigen Ereignisauftritts belegt. Im Effekt erfolgt das Setzen der Zustandsvariablen *state* auf den Nachfolgezustand "init", damit die Aktion nicht noch einmal schaltbereit wird. Die Aktionen *enqueueError*, *dequeueError* und *processError* spezifizieren Fehler, die beim Einfügen bzw. Entfernen von Aufrufen aus der Warteschlange im Rahmen der Verarbeitung auftreten. Dementsprechend werden die Aktionen *timeoutError*, *timeoutDequeue* und *timeoutProcess* dazu verwendet, das Verhalten bei Zeitüberschreitungen bei der Kommunikation zwischen dem Objekt *abstractController* und dem Objekt *abstractSensor* zu modellieren.

Als nächstes wird auf die Übersetzung des Sequenzdiagramms des Beispiel-Analysemusters aus Abbildung 10.7 eingegangen. Dieses Sequenzdiagramm beschreibt die Kopplung der objektlokalen

```
PROCESS abstContrTimes

ACTIONS

waitSensorToValuesComputed;

waitToStopped;

stoppedToWait;

actionReadx;

actionReady;

actionWritey;

actionComputeNewy;

actionGetValueCall;

actionSetValueCall;

enqueue;

dequeue;

! Die Wartezeiten an den Aktionen
IMMEDIATE waitSensorToValuesComputed;

IMMEDIATE waitToStopped;

IMMEDIATE stoppedToWait;

P MAX TIME actionReadx : tactionReadx;

P MAX TIME actionReady : tactionReady;

P MAX TIME actionWritey : tactionWritey;

P MAX TIME actionComputeNewy : tactionComputeNewy;

P MAX TIME actionGetValueCall : tactionSetValueCall;

P MAX TIME actionSetValueCall : tactionSetValueCall;

P MAX TIME enqueue : tenqueue;

P MAX TIME dequeue : tdequeue;

END abstContrTimes
```

Abbildung 10.12: Der cTLA-Prozesstyp `abstContrTimes`

Aktionen der bereits vorgestellten Objektprozesse.

Zur Übersetzung eines Sequenzdiagramms des Beispielmusters wird ein neuer Systemprozess eingeführt, der Prozessinstanzen sämtlicher im Sequenzdiagramm spezifizierter Objekte sowie eine Prozessinstanz des Prozesses *SequenceDiagram* enthält.

Der Prozesstyp *SequenceDiagram* in Abbildung 10.14 wird verwendet, um das Sequenzdiagramm aus Abbildung 10.7, das alle Interaktionen und Ereignisauftritte der Objekte eines Reglers beschreibt, zu modellieren. Erfolgt ein Ereignisauftritt (s. Abschnitt 3.4.4) an der Lifeline eines Objektes, der mit der Ausführung einer UML-Action durch das Objekt übereinstimmt, muss auf dessen Zulässigkeit im Rahmen des totalen Traces eines Sequenzdiagramms geprüft werden. Die cTLA-Aktionen, die für UML-Actions eingeführt worden sind, stellen über den Aktionenparameter *cEO* ihren Ereignisauftritt mit der Syntax "Objektname.Aktionsname" bereit. Abfolgen von Ereignisauftritten werden durch Traces beschrieben, wobei die Ereignisauftritte nach ihrem zeitlichen Auftreten innerhalb des Sequenzdiagramms in einem Trace angeordnet sind. Durch einen totalen Trace werden alle Ereignisauftritte eines Sequenzdiagramms entsprechend ihrer zeitlichen

```
PROCESS abstractSensor
IMPORT Sequence;
VARIABLES
      state : {"init", "enqueued", "dequeued", "processed", "returned", "Error", "Timeout"};
      qu : Sequence;
      x : Real;
Init ≜
ACTIONS
enqueue(m : Message) ≜ ! Einfuegen einer Anfrage
    ∧ state = "init"
    ∧ state′ = "enqueued"
    ∧ qu′ = Append( qu, m)
    ∧ UNCHANGED x;
dequeue(currentEventOccurence : String) ≜ ! Entfernen einer Anfrage
    ∧ state = "enqueued"
    ∧ currentEventOccurence = "O2.Dequeue"
    ∧ state′ = "dequeued"
    ∧ qu′ = Tail(qu)
    ∧ x′ = Head(qu);
enqueueError(currentEventOccurence : String) ≜ ! Fehler beim Einfuegen
    ∧ state = "enqueued"
    ∧ currentEventOccurence = "O2.EnqueueError"
    ∧ state′ = "Error"
    ∧ UNCHANGED ⟨qu, x⟩;
enqueueTimeout(currentEventOccurence : String) ≜ ! Zeitueberschreitung beim Einfuegen
dequeueError(currentEventOccurence : String) ≜ ! Fehler beim Entfernen
    ∧ state = "dequeued"
    ∧ currentEventOccurence = "O2.DequeueError"
    ∧ state′ = "Error"
    ∧ UNCHANGED ⟨qu, x⟩;
dequeueTimeout(currentEventOccurence : String) ≜ ... s.
process(currentEventOccurence : String) ≜
    ∧ state = "dequeued"
    ∧ currentEventOccurence = "O2.processed"
    ∧ state′ = "processed"
    ∧ UNCHANGED ⟨qu, x⟩;
processError(currentEventOccurence : String) ≜
    ∧ state = "processed"
    ∧ state′ = "Error"
    ∧ currentEventOccurence = "O2.ProcessError"
    ∧ UNCHANGED ⟨qu, x⟩;
callTimeout(currentEventOccurence : String) ≜ ... s. Anhang
callError(currentEventOccurence : String) ≜
getValueReply(data : Real; currentEventOccurence : String) ≜
    ∧ (state = "returned" ∨ state = "Error" ∨ state = "Timeout")
    ∧ currentEventOccurence = "O2.actionGetValueReply"
    ∧ data = IF state = "returned"
             THEN x
             ELSE IF state = "Error"
             THEN "Error"
             ELSE "Timeout"
    ∧ x′ = 0;
END
```

Abbildung 10.13: Der Prozesstyp `abstractSensor`

Abfolge beschrieben[5]. Ein partieller Trace [HS03] enthält nur eine Teilstück der Ereignisauftritte eines Sequenzdiagramms. Die Idee für Übersetzung eines Sequenzdiagramms besteht darin, dass vor der Ausführung einer Aktion, geprüft wird, ob ihr zugehöriger Ereignisauftritt zum gegenwärtigen Zeitpunkt durch das Sequenzdiagramm erlaubt ist. Um zu prüfen, ob eine UML-Action im Kontext eines betrachteten Sequenzdiagramms ausgeführt werden darf, wird ein partieller Trace *currentTrace* aufgebaut, der bei der Ausführung der zugehörigen cTLA-Aktion um deren Ereignisauftritt erweitert wird. Dabei ist zu garantieren, dass eine Aktion nur ausgeführt werden darf, wenn die Verkettung von *currentTrace* und dem Ereignisauftritt einer Aktion ein Anfangsstück des totalen Traces des Sequenzdiagramms ist.

Der Prozesstyp *SequenceDiagram* verfügt über den Prozessparameter *SetOfTraces*, der mit der Menge, die die Sequenz des totalen Traces des Sequenzdiagramms eines Systems enthält, parametriert wird. Damit wird eine Sequence, die alle Ereignisauftritte in der richtigen Reihenfolge enthält, an den Prozess übergeben. Durch die Konstante *setOfPossibleTraces* wird die Menge aller zulässigen Subtraces (auch partieller Trace genannt) des *totalen Traces* unter Verwendung des TLA-Operators *SubSeq* [Lam03], der Teilsequenzen parametrierbarer Länge einer Sequence bildet, ermittelt. Dazu werden dem Operator eine Sequenz $\langle S_1, \ldots, S_n \rangle$ und zwei Argumente m und n übergeben. Dabei gibt m das m-te und n das n-te Element der Sequenz an. Mit dem Operator *SubSeq* wird dann die Sequenz $\langle S[m], S[m+1], \ldots, S[n] \rangle$ berechnet. Falls $m < 1$ ist oder $n > Len(s)$, ist das Ergebnis nicht definiert, es sei denn die Länge der Sequenz ist null. Mit dem Operator *UNION* wird die Vereinigungsmenge aller zulässigen Traces konstruiert, indem die Traces unterschiedlicher Länge, zu einer Menge vereinigt werden.

Der Zustandsvariablen *currentTrace* wird der gegenwärtige Subtrace der Ausführung des Sequenzdiagramms, der bereits abgearbeitet worden ist, zugewiesen. Initialisiert wird die Zustandsvariable *currentTrace* mit der leeren Sequence << >> im Initialisierungsprädikat des Prozesses. Der Prozess besitzt die Aktion *permittedActionOfSD*, um zu prüfen, ob eine einzelner zu einer Action gehöriger Ereignisauftritt durch den *Trace* zulässig ist. Um den durch eine Action bereitgestellten Ereignisauftritt entgegenzunehmen, besitzt die Aktion den Parameter *currentEventOccurrence*. Durch den Booleschen Operator *startSequence* mit den Parametern *sot* und *d* wird geprüft, ob die Aktion schalten darf. Der Parameter *sot* gibt alle zulässigen Traces an, während durch den Parameter *d* die Verkettung des *currentTrace* mit einem neuen Ereignisauftritt beschrieben wird. Falls die Verkettung des durch den Aktionenparameter bereitgestellten Ereignisauftritts mit der Zustandsvariable *currentTrace* in der Menge *setOfPossibleTraces* enthalten ist, gibt der Operator TRUE zurück und die Aktion *permittedActionOfSD* darf schalten. Der aktuelle Trace, welcher durch die Verkettung des bisherigen Traces mit dem neuen Ereignisauftritt entsteht, wird der Zustandsvariable *currentTrace* zugewiesen.

10.3.4 Wartezeiten des Prozesstyps *abstractSensor*

Auch für den Prozesstyp *abstractSensor* müssen Wartezeiten spezifiziert werden. Diese werden in dem Prozesstyp *abstSensorTimes*, der in Abbildung 10.15 gezeigt wird, angegeben. Zunächst werden die Aktionen *enqueue* bis *process* deklariert. Anschließend werden für diese Aktionen *enqueue*, *dequeue*, *enqueueError*, *enqueueTimeout*, *dequeueError*, *dequeueTimeout*, *process* die maximalen, persistenten Wartezeiten spezifiziert.

10.3.5 Kopplung der cTLA-Prozesse des Analysemusters

Die Abbildung 10.16 zeigt den Systemprozesstyp *AbstractPatternProcess* des Analysemusters. Der **PROCESSES** Abschnitt des Prozesses *AbstractPatternProcess* deklariert Instanzen sämtlicher Objektprozesse, deren Objekte in dem Sequenzdiagramm auftreten. Dies sind die Prozesse *O1*, *O2*, *O3* und *sd* der Prozesstypen *abstractController*, *abstractSensor*, *abstractActor*, *SequenceDiagram*. Der Bezeichner eines Prozesses entspricht dem des Objektprozesses, der verwendete Prozesstyp

[5]Bei den hier gezeigten Sequenzdiagrammen erhält man den totalen Trace durch notieren der Ereignisauftritte unter Beachtung ihrer Reihenfolge innerhalb des Diagramms. Für umfangreiche Semantik der Sequenzdiagramme gelten in der Arbeit entsprechende Einschränkungen

```
PROCESS SequenceDiagram(SetOfTraces : SUBSET(Sequence))

! e.g. SetOfTraces is:
!(≪''O1.Loop'', ''O2.actionAcceptCallLoop'', ''O1.actionGetValue'',
!   ''O2.dequeue'',''O2.actionGetValueReply'', ''O1.dequeue'',
!   ''O1.actionSetValue'', ''O2.dequeue'',
!   ''O3.actionSetValueReply'', ..., ''O1.dequeue''≫)
CONSTANT

MaxTraceLength ≜ 50;

startSequence(sot, d) ≜
        ∃ s ∈ sot : SubSeq(sot, 1, length(d)) = d;

! compute the transitive closure of SetOfTraces
! setOfPossibleTraces is:
! { ≪''O1.Loop''≫, ≪''O1.Loop'', ''O2.actionAcceptCallLoop''≫
!   ≪''O1.Loop'', ''O2.actionAcceptCallLoop'', ''O1.actionGetValue''≫,
!   ≪''O1.Loop'', ''O2.actionAcceptCallLoop'', ''O1.actionGetValue'', ''O2.dequeue''≫,
!   ... }
setOfPossibleTraces ≜ UNION { {SubSeq(d, 1, 2) : d ∈ setOfTraces},
        {SubSeq(d, 1, 3) : d ∈ setOfTraces}, ...
        {SubSeq(d, 1, MaxTraceLength) : d ∈ setOfTraces}};

VARIABLES
 currentTrace : Sequence;

INIT ≜ currentTrace = ≪ ≫;

ACTIONS

    permittedActionOfSD(cEO : EventOccurrence) ≜
    startSequence(setOfPossibleTraces, currentTtrace ∘ cEO)  ∧
    currentTrace′ = currentTrace ∘ currentEventOccurrence;

END
```

Abbildung 10.14: Der Prozesstyp SequenceDiagram

dem zur Übersetzung der Klasse des Objektes eingeführten Prozesstyp. Der Prozess *TO1* ist eine Instanz des Prozesstyps *abstContrTimes*, um die Realzeitanforderungen des Prozesses *O1* zu behandeln. *TO2* ist eine Instanz des Prozesstyps *abstSensorTimes* der in Abbildung 10.15 angegeben ist. Der Prozess *TO3* enthält die Realzeitanforderungen für den Aktor. Weiterhin wird eine Prozessinstanz *sd* des wiederverwendbaren Prozesstyps *SequenceDiagram* eingebunden. Dieser Prozess wird bei seiner Instantiierung mit der cTLA-Sequence sämtlicher Ereignisauftritte parametriert. Ein Ereignisauftritt wird durch eine Zeichenkette – bestehend aus dem Objektnamen – und dem Bezeichner der zugehörigen UML-Action übersetzt. So bezeichnet "O1.actionGetValue" die Ausführung der CallOperationAction *actionGetValue*. Um den Trace der zulässigen Ereignisauftritte des Sequenzdiagramms zu kodieren, wird eine Sequence der zulässigen Ereignisauftritte des Sequenzdiagramms gebildet. Der Prozessparameter *completeTrace* des wiederverwendbaren Prozesses *SequenceDiagram*, der wie zuvor beschrieben aus dem Sequenzdiagramm berechnet worden ist, wird mit dem resultierenden totalen Trace des Sequenzdiagramms des Musters parametriert. Die Reihenfolge, in der diese Ereignisauftritte in einem Trace angeordnet werden, entspricht der durch das Sequenzdiagramm vorgegebenen Reihenfolge.

Nun soll der Aufbau der Systemaktionen des Prozesstyps *AbstractPatternProcess* – geordnet nach den gekoppelten Prozessen – betrachtet werden. Der Prozesstyp enthält zahlreiche Systemaktionen, die daher nur auszugsweise durch die wichtigsten Repräsentanten für jede Art – also z. B. Transition, CallOperationAction, ReadAttributeAction – einer Systemaktion dargestellt werden. Der Prozess ist vollständig in Anhang B.1 angegeben. Die zur Übersetzung von Transitionen

```
PROCESS abstSensorTimes
ACTIONS
enqueue;
dequeue;
enqueueError;
enqueueTimeout;
dequeueError;
process;
P MAX TIME enqueue : taSenqueue;
P MAX TIME dequeue : tasDequeue;
P MAX TIME enqueueError : taSenqueueError;
P MAX TIME enqueueTimeout : taSenqueueTimeout;
P MAX TIME dequeueError : tdequeueError;
P MAX TIME dequeueTimeout : taSdequeueTimeout;
P MAX TIME process : taSProcess;
END
```

Abbildung 10.15: Der Prozesstyp `abstSensorTimes`

eingeführten Aktionen, wie die Aktion *waitSensorToValuesComputed*, werden in der Systemaktion aus Abbildung 10.16 gekoppelt. In der Systemaktion *waitSensorToValuesComputed* schaltet nur die Aktion *waitSensorToValuesComputed* des Prozesses *O1*, während alle anderen Prozesse stottern. Die Kopplungen der Aktionen, die für die übrigen Transitionen eingeführt werden – z. B. *waitToStopped* –, erfolgen nach dem gleichen Prinzip.

Auch für jede Übersetzung einer UML-Action wird eine eigenständige Systemaktion eingeführt. Für die Aktionen von objektlokaler Bedeutung sind das die Systemaktionen *actionReadx*, *actionReady*, *actionWritey* und *actionComputeNewy*. Bei der Systemaktion *actionReadx* schalten die Aktionen *actionReadx* des Prozesses *O1* und die Aktion *permittedActionOfSD* des Prozesses *sd* gemeinsam, während alle übrigen Prozesse einen Stotterschritt vornehmen. Dabei sind die spezifizierten Wartezeiten aus *TO1* zu berücksichtigen. Durch den gemeinsamen Aktionenparameter *currentEventOccurence* wird sichergestellt, dass die Systemaktion nur schaltet, wenn dies durch das Sequenzdiagramm des Analysemusters spezifiziert ist. Das Kopplungsprinzip für die Systemaktionen *actionReady* und *actionReadx* ist identisch.

Die Systemaktion *actionWritey* wird – wie in Abbildung 10.17 gezeigt – gekoppelt, indem die Aktion *actionWritey* des Prozesses *O1*, die Wartezeiten für *actionWritey* aus *TO1* und die Aktion *sd.permittedActionOfSD* aus *sd* konjugiert werden. Beim Schalten der Systemaktion *actionWritey* stottern die übrigen Prozesse.

Bei der CallOperationAction *actionGetValueCall* werden in der Systemaktion *getCallValueSensor*, die zur Übersetzung der Action eingeführte Aktion des Prozesses *O1*, in dem die Action spezifiziert ist, mit der Aktion *enqueue* des Empfängerprozesses *O2*, der ebenfalls im Sequenzdiagramm angegeben wird, und der Aktion *permittedActionOfSD* des Prozesses *sd* gekoppelt. Weiterhin werden in der Systemaktion die Aktion *actionSetValueCall* des Prozesses *TO1* und die Aktion *enqueue* des Prozesses *TO2* gekoppelt, um die Aktionen mit ihren persistenten, maximalen Wartezeiten zu versehen. Alle anderen Objektprozesse nehmen in dieser Systemaktion einen Stotterschritt vor. Die Systemaktion besitzt die Parameter *currentEventOccurrence* und den Parameter *value* aus der Übersetzung der *CallOperationAction*, um Daten von dem aufgerufenen Objekt zu übertragen.

Aufgrund der Symmetrie von Sensor- und Aktoraufrufen werden die für die Interaktion des

```
PROCESS AbstractPatternProcess
! vollständige Spezifikation in Anhang B
PROCESSES
  O1 : abstractController(k); ! Prozessinstanz v. abstractController
  TO1 : abstContrTimes; ! Prozessinstanz v. abstractSensor
  O2 : abstractSensor(); ! Prozessinstanz v. abstractSensor
  TO2 : abstSensorTimes; ! Prozessinstanz v. abstSensorTimes
  O3 : abstractActor(); ! Prozessinstanz v. abstractActor
  TO3 : abstSensorTimes; ! im Anhang B
  sd : SequenceDiagram(≪''O1.Loop'', ''O2.actionAcceptCallLoop'', ''O1.actionGetValue'',
  ''O2.dequeue'',''O2.actionGetValueReply'', ''O1.dequeue'', ''O1.actionSetValue'',
  ''O2.dequeue'', ''O3.actionSetValueReply'', ..., ''O1.dequeue''≫); ! Prozessinstanz von
  ! SequenceDiagram

  waitSensorToValuesComputed ≙ ! Kopplung Transition waitSensor → ValuesComputed
     ∧ O1.waitSensorToValuesComputed
     ∧ TO1.stutter
     ∧ O2.stutter
     ∧ TO2.stutter
     ∧ O3.stutter
     ∧ TO3.stutter
     ∧ sd.stutter;
  waitToStopped ≙ ... ! Kopplung Transition wait → stopped

  actionReadx(currentEventOccurence : String) ≙ ! SysAct f. actionReadx
     ∧ O1.actionReadx(currentEventOccurence)
     ∧ TO1.actionReadx
     ∧ O2.stutter
     ∧ TO2.stutter
     ∧ O3.stutter
     ∧ TO3.stutter
     ∧ sd.permittedActionOfSD(currentEventOccurence);
  actionReady(currentEventOccurence : String) ≙ ... ! ! SysAct actionReady
  actionWritey(currentEventOccurence : String) ≙ ... ! SysAct actionWritey
  actionComputeNewy(currentEventOccurence : String) ≙ ! SysAct f. actionComputeNewy
     ∧ O1.actionComputeNewy(currentEventOccurence)
     ∧ TO1.actionComputeNewy
     ∧ O2.stutter
     ∧ TO2.stutter
     ∧ O3.stutter
     ∧ TO3.stutter
     ∧ sd.permittedActionOfSD(currentEventOccurence);
  getCallValueSensor(currentEventOccurence : String; val : Real) ≙ ! SysAct for getValue
     ∧ O1.actionGetValueCall(value, currentEventOccurence)
     ∧ TO1.actionGetValueCall
     ∧ O2.enqueue(value)
     ∧ TO2.enqueue
     ∧ O3.stutter
     ∧ TO3.stutter
     ∧ sd.permittedActionOfSD(currentEventOccurence);
  Tick(r : Real) ≙ ... ! SysAct Tick

  ...
   END
```

Abbildung 10.16: Die wesentlichen Systemaktionen für die Transitionen des Prozesstyps `AbstractPatternProcess`

```
actionWritey(currentEventOccurence : String) ≜ ! SysAct actionWritey
     ∧ O1.actionWritey(currentEventOccurence)
     ∧ TO1.actionWritey
     ∧ O2.stutter
     ∧ TO2.stutter
     ∧ O3.stutter
     ∧ TO3.stutter
     ∧ sd.permittedActionOfSD(currentEventOccurence);
```

Abbildung 10.17: Die Systemaktion `actionWritey`

```
Tick(r : R) ≜
    ∧ O1.Tick(r)
    ∧ TO1.stutter
    ∧ O1.Tick(r)
    ∧ TO2.stutter
    ∧ O1.Tick(r)
    ∧ TO3.stutter
    ∧ sd.stutter;
```

Abbildung 10.18: Die Systemaktion `Tick`

Controllers mit dem Aktor eingeführten Systemaktionen hier nicht erläutert, denn sie werden nach den gleichen Prinzipien übersetzt.

10.3.6 Aktion *Tick* des Grobsystems

Durch die Aktion *Tick* wird die Uhrenvariable *now* jedes cTLA-Prozesses in kleinen Schritten und lebendig inkrementiert. Das Zeitraster, welches durch Schaltfolgen der Aktion *Tick* vorgegeben wird, kann beliebig fein mit der Konstante *clockresolution* eingestellt werden. Die *Tick* Aktionen jedes Prozesses werden miteinander gekoppelt. Hierdurch wird gewährleistet, dass die Zeit in ausreichend kleinen Schritten fortschreitet. Die im Prozesstyp *AbstractPatternProcess* angegebene Aktion *Tick* konjugiert die *Tick* Aktionen der Prozesse *O1*, *O2* und *O3* des Grobsystems. Die vollständige Konjunktion der Teilaktionen zu einer Systemaktion ist in Abbildung 10.18 angegeben. Beim Schalten der Aktion nehmen die übrigen Prozesse einen Stotterschritt vor.

In Abbildung 10.19 ist exemplarisch die Aktion *Tick* des Prozesstyps *abstractSensor* gezeigt. Diese Aktion spezifiziert das Voranschreiten der Timervariablen für die Aktionen dieses Grobsystems – nämlich *timerEnqueue*, *timerDequeue*, *timerProcess*, *timerReturn*, *timerEnqueueError*, *timerDequeueError*, *timerProcessError* – für die einzelnen **P MAX TIME** Konstrukte der Aktionen sowie der Uhrenvariablen *now*. Die Aktion besitzt Implikationen als Vorbedingungen. Falls eine Aktion schaltbereit muss daraus folgen, dass die Zeitvariable *now'* kleiner oder gleich der Summe aus *now* und der Differenz aus der persistenten, maximalen Wartezeit sowie der Timervariable ist. Die Implikationen verhindern, dass *now* einen Wert überschreitet, der das Schalten einer Aktion erfordert, ohne dass diese eine Gelegenheit zum Schalten hatte. Somit wird die zugehörige Aktion zum Schalten gezwungen, bevor *Tick* die Zustandsvariable *now* weiter erhöhen darf. Die Implikation ist als Vorbedingung erfüllt, wenn die zugehörige Aktion nicht schaltbereit ist. Beispielsweise soll dies für das sechste Konjunkt der Aktion *Tick* betrachtet werden. Für die linke Seite der Implikation ergibt sich der Ausdruck $enabled(Process)$. Der Ausdruck auf der rechten Seite der Implikation $now' \leq now + (tProcess - TimerProcess)$ gibt an, dass bei einer Schaltbereitschaft von *Process now'* kleiner als die Summe aus *now* und der Differenz der maximalen, persistenten Wartezeit *tProcess* sowie der Timervariablen *TimerProcess* ist. Die übrigen Implikationen sind nach demselben Schema aufgebaut. Die Timervariablen für die Zeitüberschreitungen stehen im Anhang B.1. Für die Aktion wird starke Fairness spezifiziert.

```
Tick(t : Real) ≜ ! Aktion Tick des Grobsystems für Zeitschritt
       LET
          TimerProg(action, timerVariable) ≜
          IF(enabled(action))
          THEN timerVariable + now′ - now
          ELSE timerVariable
       IN
       ∧ now′ = t
       ∧ now′ > now
       ∧ now′ ≤ now + ϵ
       ∧ ∀ [d ∈ Data] :: enabled(Enqueue(d)) ⇒ now′ ≤ now + (tEnqueue - timerEnqueue[d])
       ∧ enabled(Dequeue) ⇒ now′ ≤ now + (tDequeue - TimerDequeue)
       ∧ enabled(Process) ⇒ now′ ≤ now + (tProcess - TimerProcess)
       ∧ ∀ [d ∈ Data] :: enabled(Return(d)) ⇒ now′ ≤ now + (tReturn - timerReturn[d])
       ∧ enabled(EnqueueError) ⇒ now′ ≤ now + (tEnqueueError - TimerEnqueueError)
       ∧ enabled(DequeueError) ⇒ now′ ≤ now + (tDequeueError - TimerDequeueError)
       ∧ enabled(ProcessError) ⇒ now′ ≤ now + (tProcessError - TimerProcessError)
       ∧ ∀ [d ∈ Data] :: timerEnqueue[d]′ = TimerProgress(Enqueue(d), timerEnqueue) ! gt1
       ∧ timerDequeue′ = TimerProg(Dequeue, timerDequeue)    ! gt2
       ∧ timerProcess′ = TimerProg(Enqueue(d), timerProcess) ! gt3
       ∧ timerReturn′ = TimerProg(Process, timerReturn) ! gt4
       ∧ timerEnqueueError′ = TimerProg(Return(d), timerEnqueueError)  ! gt5
       ∧ timerDequeueError′ = TimerProg(EnqueueError, timerDequeueError) ! gt6
       ∧ timerProcessError′ = TimerProg(ProcessError, timerProcessError) ! gt7
       ∧ UNCHANGED ⟨state, qu, x⟩;
```

Abbildung 10.19: Die Aktion `Tick` des Prozesstyps `abstractSensor`

Kapitel 11

Übersetzung einer Entwurfsmuster-Kombination nach cTLA

In diesem Kapitel wird ausführlich die Übersetzung der Entwurfsmuster-Kombination in cTLA erörtert. Dazu wird in Abschnitt 11.1 zunächst eine detaillierte UML Spezifikation der Entwurfsmuster-Kombination angegeben. In Abschnitt 11.2 wird die Formalisierung der Entwurfsmuster-Kombination behandelt. Die Übersetzung orientiert sich an den in Abschnitt 10.1 angegebenen Prinzipien.

Hierbei wird deutlich, dass objektorientierte Software-Modelle eine große semantische Lücke zu den formalen Modellierungssprachen – wie etwa cTLA – aufweisen. Dies führt zu umfangreichen cTLA-Spezifikationen.

11.1 Objektorientierter Entwurf der Entwurfsmuster-Kombination

In Abbildung 11.1 wird das Klassendiagramm für den objektorientierten Entwurf (OOD) des Verfeinerungsmusters, das die Klassen und Assoziationen der verwendeten Entwurfsmuster sowie die aus dem Klassendiagramm des Analysemuster entnommenen Klassen beschreibt, gezeigt. Die Klasse *periodicTask* besitzt die privaten Attribute x und y. Das Attribute x modelliert den Istwert und das Attribut y den Stellwert.

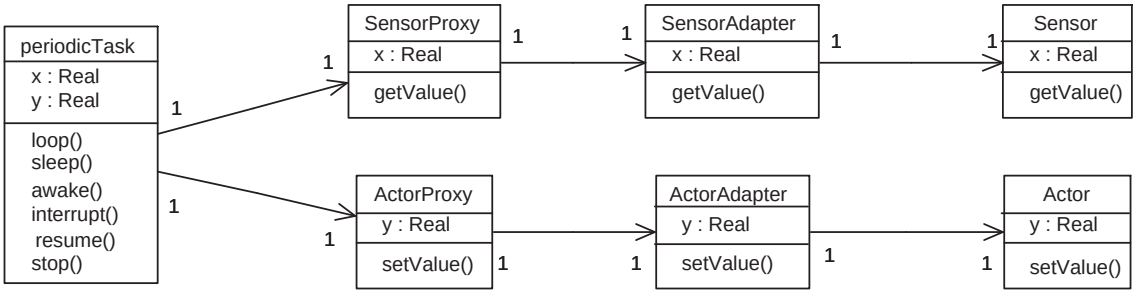


Abbildung 11.1: Das Klassendiagramm des objektorientierten Entwurfes des Verfeinerungsmusters *verteilter, entfernter Regler*

Die öffentliche Operation *loop* dient dem Start der Regelschleife. Eine weitere öffentliche Operation *sleep*, mit der ein Objekt dieser Klasse durch einen Eingabeparameter für einen bestimmten Zeitraum in einen inaktiven Zustand – der keine Betriebssystem-Ressourcen benötigt – versetzt wird. Mit dem Aufruf der öffentlichen Operation *awake* wird der inaktive Zustand beendet und die Bearbeitung der Regelschleife fortgesetzt. Durch die öffentliche Operation *interrupt* ist eine Unter-

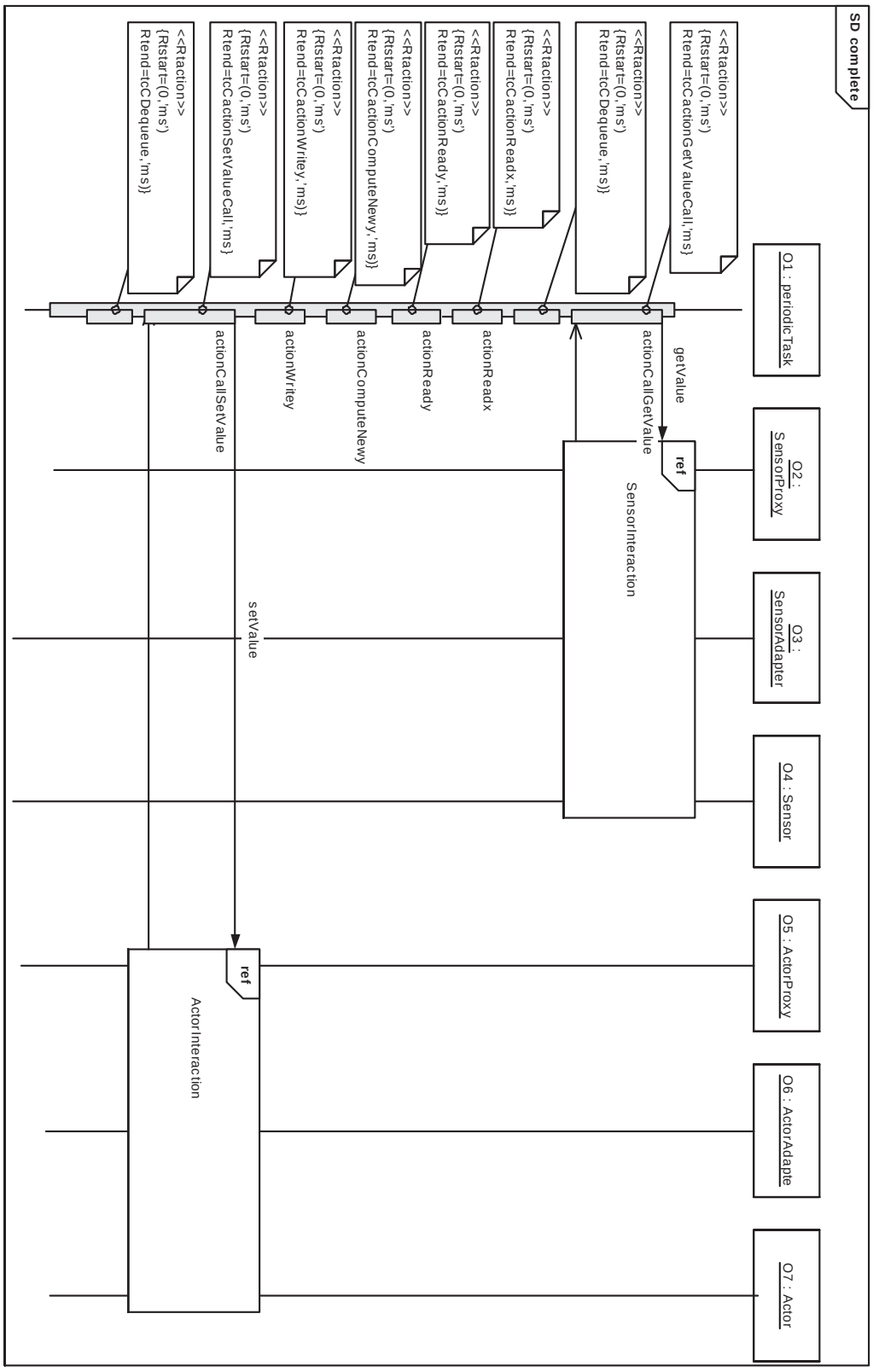


Abbildung 11.2: Das Sequenzdiagramm der Entwurfsmuster-Kombination

brechung der Abarbeitung der Regelschleife des Objektes *O1* möglich. Die öffentliche Operation *resume* führt zur Fortsetzung einer durch einen Interruptaufruf unterbrochenen periodischen Task. Durch die Operation *resume* wird der Regelablauf durch die Task fortgesetzt. Mit der Operation *sleep* wird die periodische Task vorübergehend angehalten. Nach Empfang der Nachricht *awake* wird die periodische Task, die vorübergehend durch Aufruf von *sleep* angehalten worden ist, fortgesetzt. Durch die Operation *sleep* wird bei Bedarf die Durchlaufzeit der Regelschleife angepasst. Dies ist sinnvoll, wenn es erforderlich ist die Regelzeit zu beeinträchtigen. Interrupts hingegen treten auf, wenn andere ablaufende Tasks Rechenzeit beanspruchen, etwa wenn Alarme oder Notsignale dringende Reaktionen erfordern. Die Klasse *SensorProxy* enthält das private Attribut x und die öffentliche Operation *getValue*, die den erhaltenen Istwert zurückliefert. Sie entspricht dem Proxy-Muster aus [GHJV93, BMR+96].

Die Klasse *SensorAdapter* enthält das private Attribut x und die öffentliche Operation *getValue*, die den erhaltenen Istwert zurückliefert. Sie geht auf das Adapter-Muster [GHJV93, BMR+96] zurück. Die Klasse *Sensor* modelliert die Software eines Sensors, der eine physikalische Größe misst. Sie besitzt ein privates Attribut x zur Aufnahme des Istwertes und eine öffentliche Operation *getValue* zum Auslesen des Istwertes. Die Klassen *ActorProxy* und *ActorAdapter*, die zwischen der periodischen Task des Reglers und der Klasse *Actor* modelliert sind, sind ähnlich zu den bisher beschriebenen Klassen für die Kommunikation – mit der Ausnahme, dass ein neuer Stellwert in den Aktor geschrieben wird – und werden daher nicht explizit beschrieben.

Die Klassen sind durch gerichtete Assoziationen im Klassendiagramm verbunden. Zwischen der Klasse *periodicTask* und der Klasse *SensorProxy* existiert eine gerichtete Assoziation. Die Multiplizität ist an beiden Assoziationsenden eins. Zwischen der Klasse *SensorProxy* und der Klasse *SensorAdapter* gibt es eine weitere gerichtete Assoziation. Auch an Assoziationsenden dieser Klassen ist die Multiplizität eins. Zwischen der Klasse *SensorAdapter* und der Klasse *Sensor* gibt es ebenfalls eine gerichtete Assoziation. Die Multiplizität an beiden Assoziationsenden ist auch hier jeweils eins. Die Assoziationen zwischen der Klasse *periodicTask* und der Klasse *Aktor* sind symmetrisch.

Zur Beschreibung der Interaktionen von Objekten des OOD des Verfeinerungsmusters ist in Abbildung 11.2 ein Sequenzdiagramm angegeben, das ein Objekt *O1* der Klasse *periodicTask*, ein Objekt *O2* der Klasse *SensorProxy*, ein Objekt *O3* der Klasse *SensorAdapter*, ein Objekt *O4* der Klasse *Sensor*, ein Objekt *O5* der Klasse *ActorProxy*, ein Objekt *O6* der Klasse *ActorAdapter* und ein Objekt *O7* der Klasse *Actor* enthält. Das Sequenzdiagramm referenziert zwei weitere Sequenzdiagramme – nämlich *SensorInteraction* und *ActorInteraction* –, die bestimmte Interaktionen im Detail angeben. In Abbildung 11.3 ist das Sequenzdiagramm *SensorInteraction*, das die Interaktionen zwischen einem *SensorProxy*-Objekt, einem *SensorAdapter*-Objekt und einem *Sensor*-Objekt angibt, gezeigt. Dabei werden die einzelnen Actions der beteiligten Objekte und deren Wartezeiten spezifiziert. Die anschließenden Erläuterungen ergeben sich aus dem Zusammenspiel der Diagramme.

Die folgenden Nachrichten werden bei Interaktionen zwischen den Objekten ausgetauscht. Die synchrone Nachricht *getValue* zwischen *O1* und *O2* repräsentiert einen Aufruf der Operation *getValue* auf *O2*. Mit dem *DurationConstraint* t_{Proxy} wird angegeben, dass die Ausführungszeit dieser Nachricht kleiner t_{Proxy} ist. Die synchrone Nachricht *getValue* zwischen *O2* und *O3* modelliert den Aufruf der Operation *getValue* auf dem Objekt *O3*. Es handelt sich bei dem Regler um ein verteiltes System, da das Task-Objekt mit den beiden Proxy-Objekten in einem eigenständigen Adressraum abläuft, während sich die Sensor-Task und die Aktor-Task und ihre Adapter-Objekte jeweils in einem anderen Adressraum befinden. Mit der synchronen *getValue*-Nachricht zwischen *O3* und *O4* wird der Aufruf der *getValue*-Operation auf *O4* modelliert.

Für jede Klasse des Klassendiagramms aus Abbildung 11.1 wird ein Statechart-Diagramm modelliert. In Abbildung 11.4 wird das Statechart-Diagramm der Klasse *periodicTask* gezeigt. Die Zustände *init*, *wait*, *stopped*, *waitSensor*, *valuesComputed* und *waitActuator* entsprechen den Zuständen des periodischen Reglers des Analysemusters und werden daher nicht noch einmal erläutert.

Im Folgenden werden die zusätzlichen Zustände, die ein Objekt der Klasse *periodicTask* einnimmt, erklärt. In den Zustand *sleep1* gelangt das Objekt, das sich im Zustand *waitSensor* befindet, sobald ein *sleep*-Aufruf vorgenommen wird. Dieser Zustand wird verlassen, wenn ein *awake*-Aufruf

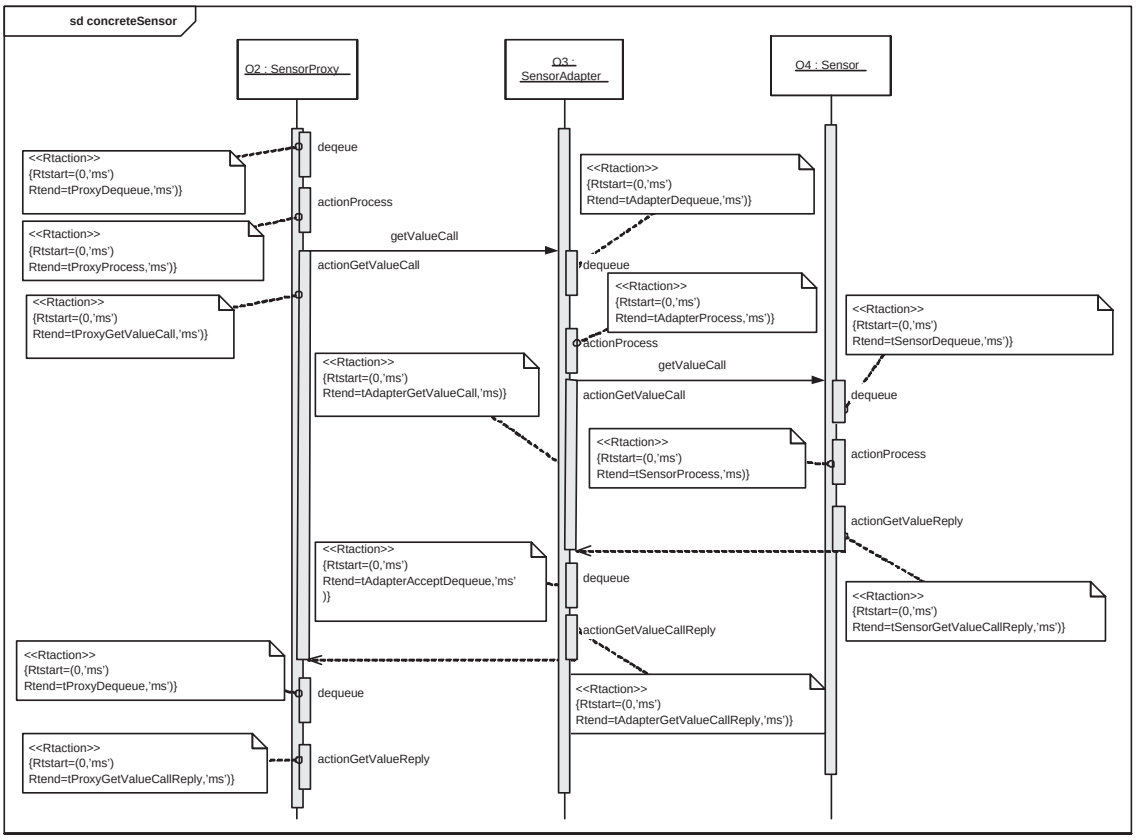


Abbildung 11.3: Das Sequenzdiagramm der Interaktionen vom *SensorProxy* bis zum *Sensor*

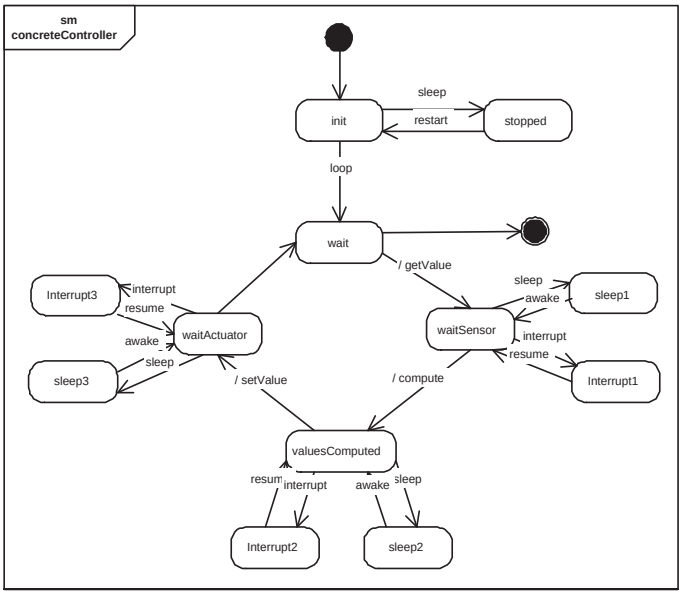


Abbildung 11.4: Das Statechart-Diagramm der Klasse *periodicTask*

an das Objekt abgesetzt wird. Der Zustand *interrupt1* eines *periodicTask*-Objektes wird betreten, wenn ein *interrupt*-Aufruf an das Objekt abgesetzt wird und es sich im Zustand *waitSensor* befindet. Der Zustand *sleep2* wird durch einen *sleep*-Aufruf betreten, wenn sich das Objekt im Zustand

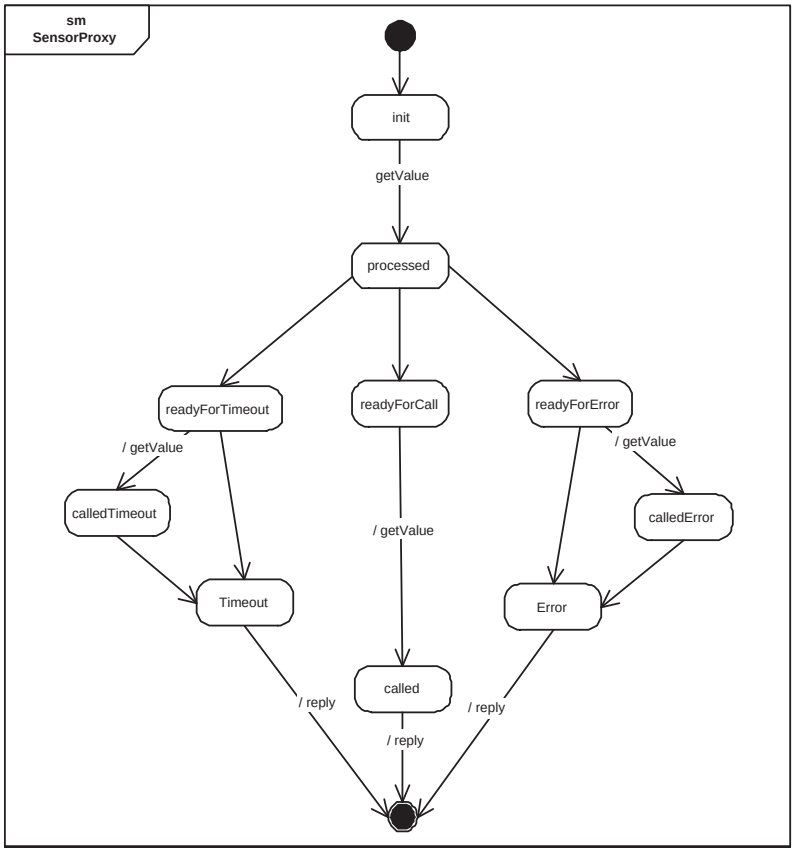


Abbildung 11.5: Das Statechart-Diagramm der Klasse *SensorProxy*

valuesComputed befindet und verlassen, wenn ein *awake*-Aufruf an das Objekt abgesetzt wird. In den Zustand *interrupt2* gelangt das Objekt, wenn es sich im Zustand *valuesComputed* befindet und ein *interrupt*-Aufruf erfolgt. Wenn sich das Objekt im Zustand *waitActuator* befindet und ein *sleep*-Aufruf erfolgt, findet ein Übergang in den Zustand *sleep3* statt. Der Zustand *interrupt3* wird aus dem Zustand *waitActuator* betreten, wenn ein *interrupt*-Aufruf erfolgt. Durch einen *resume*-Aufruf wird dieser Zustand wieder verlassen.

Im Anhang C.1 ist das Statechart-Diagramm der Klasse *SensorAdapter* angegeben. Abschließend wird in Anhang C.2 das Statechart-Diagramm der Klasse *Sensor* angegeben.

Das Statechart-Diagramm der Klasse *SensorProxy* ist in Abbildung 11.5 gezeigt. Im Anschluss an die Erzeugung eines solchen Objektes wird der Zustand *init* eingenommen. Nach einem *getValue*-Aufruf wird der Zustand *processed* betreten. Aus diesem Zustand führen drei Transitionen, die bestimmen, ob der Zweig der normalen bzw. der fehlerbehafteten Verarbeitung oder der Zweig mit einer Zeitüberschreitung durchlaufen wird. Man erkennt, dass die beiden letzten Zweige symmetrisch sind. Die Auswahl des nächsten Zustandes erfolgt nicht-deterministisch, weil die Transitionen weder über ein Ereignis noch über eine Bedingung verfügen. Damit ist auch die Auswahl des Zweiges nicht determiniert. Für den normalen Fall wird zunächst der Zustand *readyForCall* betreten. Anschließend wird ein *getValue*-Aufruf an den *SensorAdapter* abgesetzt und der Zustand *called* betreten. Sobald eine Rückantwort erfolgt, wird der Zustand verlassen und mit einer ReplyAction das Ergebnis an die *periodicTask* zurückgeleitet. Für den Fehlerzweig ist die Behandlung komplizierter. Zunächst wird der Zustand *readyForError* betreten. Fehler können noch vor dem Absetzen des Aufrufs (Transition von *readyForError* nach *Error*) und nach dem Absetzen des *getValue*-Aufrufes abgesetzt werden. Für den Fall, dass ein Fehler nach dem Absetzen des Aufrufes erfolgt, muss unterschieden werden, ob der Fehler außerhalb des *SensorProxys* oder nach dem Empfang der Antwort auftritt. Dafür sind die übrigen Zustände und Transitionen vorhanden. In jedem Fall

Variablenname	Beschreibung der Wartezeit
$t_{cCactionReadx}$	Zeit zum Lesen des Attributes x im *Controller*
$t_{cCactionReady}$	Zeit zum Lesen des Attributes y im *Controller*
$t_{cCactionComputeNewy}$	Zeit zum Berechnen eines neuen Funktionswertes für y
$t_{cCactionWritey}$	Zeit zum Lesen des Attributes y im *Controller*.
$t_{cCDequeue}$	Zeit zum Entfernen eines Aufrufes aus der Warteschlange des *Controllers*
$t_{cCCallInterrupt}$	Zeit für einen *interrupt*-Aufruf des *Controller*-Objektes
$t_{cCCallResume}$	Zeit für einen *resume*-Aufruf des *Controller*-Objektes
$t_{cCallSleep}$	Zeit, um einen *sleep*-Aufruf auf einem *Controller* durchzuführen
$t_{cCCallAwake}$	Zeit für den Aufruf eines *awake* auf einem *Controller*
$t_{cCSetValueCall}$	Zeit für einen *setValueCall*-Aufruf des *Controller*
$t_{ProxyEnqueue}$	Zeit, um einen *getValueCall*-Aufruf bzw. das zugehörige Return in die Warteschlange aufzunehmen
$t_{ProxyDequeue}$	Zeit zum Entfernen eines *getValueCall*-Aufrufes aus der Warteschlange
$t_{ProxyProcess}$	Zeit zur Verarbeitung eines *getValueCall*-Aufrufes im *SensorProxy*
$t_{ProxyGetValueCall}$	Zeit zum Aufruf eines *getValueCall* auf dem *SensorAdapter* Objekt
$t_{ProxyGetValueCallReply}$	Zeit zum Einfügen des Returns eines *getValueCall*-Aufrufes in die Warteschlange
$t_{ProxyTimeout1}$	Zeit bis zum Auftreten eines Timeouts (TO) vor einem *getValueCall*
$t_{ProxyTimeout2}$	Zeit für das Auftreten eines TO bei einem *getValueCall*
$t_{ProxyTimeout3}$	Zeit bis zum Auftreten eines TO, nachdem ein *getValueCall*-Return in die Warteschlange eingefügt wurde
$t_{ProxyTimeout4}$	Zeit für das Auftreten einer Zeitüberschreitung, nachdem ein *getValueCall*-Return aus der Warteschlange entfernt wurde
$t_{ProxyError1}$	Zeit zum Auftreten eines Fehlers, noch bevor ein *getValueCall* abgesetzt wurde
$t_{ProxyError2}$	Zeit für das Auftreten einer Fehlers, nachdem ein *getValueCall* abgesetzt wurde
$t_{ProxyError3}$	Zeit zur Verarbeitung bis zum Auftreten eines Fehlers nachdem ein *getValueCall*-Return in die Warteschlange eingefügt wurde
$t_{ProxyError4}$	Zeit für das Auftreten eines Fehlers nachdem ein *getValueCall*-Return aus der Warteschlange entfernt wurde
$t_{ProxySetCall1}$	Zeit, um den Zweig zur weiteren Verarbeitung zu wählen
$t_{ProxySetCall2}$	Zeit für die Auswahl des Fehlerzweiges
$t_{ProxySetCall3}$	Zeit, um den Zweig für eine Zeitüberschreitung zu wählen
$t_{AdapterEnqueue}$	Zeit zum Einfügen eines *getValue*-Aufrufes bzw. Returns in die *SensorAdapter*-Warteschlange.
$t_{AdapterDequeue}$	Zeit zum Entfernen eines Aufrufes aus der *SensorAdapter*-Warteschlange
$t_{AdapterProcess}$	Zeit zur Verarbeitung eines *getValueCall*-Aufrufes im *SensorAdapter*-Objekt
$t_{AdapterGetValueCallReply}$	Zeit für die Rückgabe des *getValueCall* des Adapters
$t_{SensorEnqueue}$	Zeit zum Einfügen eines Aufrufes *getValueCall* in die Warteschlange des Sensors
$t_{SensorDequeue}$	Zeit zum Entfernen eines Aufrufes aus der Warteschlange
$t_{SensorProcess}$	Zeit zur Verarbeitung eines *getValueCall*-Aufrufes im Sensor
$t_{SensorGetValueReply}$	Zeit zur Rückgabe des *getValueCall*-Aufrufes durch den Sensor

Tabelle 11.1: Die Definition der Variablen für die Wartezeiten aus der Entwurfsmuster-Kombination

erfolgt eine Antwort an die *periodicTask*.

Realzeit:

Bei den Entwurfsmodellen geben die Bedingungen für die Realzeit charakteristische Realzeit-Eigenschaften der Entwurfsmodelle an. Ohne die genaue Spezifikation dieser Eigenschaften ist kein Realzeit-Verfeinerungsbeweis, wie er in Kapitel 12 vorgestellt wird, für ein Verfeinerungsmuster durchführbar. Wiederum sind die Eigenschaften aus den UML-Diagrammen ablesbar. Hierbei notieren die Summenzeichen Wartezeiten, die durch mehrfache Ausführung einer Aktion aufsummiert werden. Die Zeiten in der Tabelle 11.1 werden in Übereinstimmung mit den UML-Diagrammen definiert. Die Wartezeiten für die periodische Task werden mit dem Präfix cC versehen.

Mit (R1)-(R4) werden die Eigenschaften für die Actions eines Controllers angegeben, die der Berechnung eines neuen Stellwertes dienen. Dafür werden sie mit den Eigenschaften des *abstractControllers* aus Abschnitt 10.2 in Beziehung gesetzt.

Durch (R5)-(R7) werden Bedingungen für die maximale Anzahl der *sleep*- bzw. *interrupt*-Aufrufe formuliert, da derartige Aufrufe natürlich Verzögerungen bewirken, die Einfluss auf die Erfüllung von Realzeiteigenschaften haben. Es ist wiederum erforderlich, diese mit den Eigenschaften des *abstractControllers* aus Abschnitt 10.2 zu verbinden.

Durch (R8)-(R10) werden die Eigenschaften eines *SensorProxys* und eines *SensorAdapters* für den Fall der normalen Verarbeitung (R8), einer Zeitverletzung (R9) und der fehlerbehafteten Verarbeitung (R10) beschrieben.
(R1) Die Bedingung $t_{aCactionReadx} \geq t_{cCactionReadx}$ bedeutet, dass die Wartezeit für das Auslesen im *abstractController*-Objekt mindestens so groß wie in der *periodicTask* ist.
(R2) $t_{aCactionReady} \geq t_{cCactionReady}$ Diese Bedingung gibt an, dass die Wartezeit für das Auslesen im *abstractController*-Objekt mindestens so groß wie in der *periodicTask* ist.
(R3) $t_{aCactionComputeNewy} \geq t_{cCactionComputeNewy}$ Durch diese Bedingung wird spezifiziert, dass die Wartezeit für die UML-Action *aCactionComputeNewy* des *abstractController*-Objektes mindestens so groß ist, wie die Wartezeit der zugehörigen Action des *periodicTask*-Objektes.
(R4) $t_{aCactionWritey} \geq t_{cCactionWritey}$ Ebenso ist die Wartezeit der UML-Action *actionWritey* des *abstractController*-Objektes mindestens so groß, wie bei der zugehörigen Action des *periodicTask*-Objektes.

(R5) Mit der im Folgenden angegebenen Bedingung wird festgelegt, dass nur eine begrenzte Anzahl von *sleep*- bzw. *interrupt*-Aufrufen an ein *periodicTask*-Objekt im Zustand *waitSensor* zulässig ist: $t_{aCgetValueCall} \geq \sum t_{cCcallInterrupt} + 2 * \sum t_{cCdequeue} + \sum t_{cCcallResume} + \sum t_{cCcallSleep} + \sum t_{cCcallAwake} + t_{cCgetValueCall}$
(R6) Durch diese Bedingung wird festgelegt, dass nur eine begrenzte Anzahl von *sleep*- bzw. *interrupt*-Aufrufen an ein *periodicTask*-Objekt im Zustand *valuesComputed* zulässig ist: $t_{aCcompute} \geq \sum t_{cCinterrupt} + 2 * \sum t_{cCdequeue} + \sum t_{cCcallResume} + \sum t_{cCcallSleep} + \sum t_{cCcallAwake}$
(R7) Durch diese Bedingung wird festgelegt, dass nur eine begrenzte Anzahl von *sleep*-Aufrufen an ein *periodicTask*-Objekt im Zustand *waitActuator* zulässig ist: $t_{aCsetValueCall} \geq \sum t_{cCcallInterrupt} + 2 * \sum t_{cCdequeue} + \sum t_{cCcallResume} + \sum t_{cCcallSleep} + \sum t_{cCcallAwake} + t_{cCsetValueCall}$
(R8) Jetzt soll das *Proxy*-Objekt betrachtet werden. Die Summe aus der Vorverarbeitungs- und der Aufrufzeit des Proxys sowie der vollständigen Verarbeitungszeit des Adapters und des Sensors ist im Falle der normalen Verarbeitung höchstens so groß wie die Verarbeitungszeit des abstrakten Sensors $t_{abstSensorProcess}$:
Es wird zusammengefasst:

1. Die Zeit $t_{FromProxyToAdpapter}$ gibt die Zeit für den *getValue*-Aufruf vom Proxy zum Sensor und zurück an. $t_{FromProxyToAdpapter} = 2 * t_{adapterEnqueue} + 2 * t_{adapterDequeue} + t_{adapterProcess} + t_{adapterGetValueCall} + t_{sensorEnqueue} + t_{sensorDequeue} + t_{sensorProcess} + t_{sensorGetValueReply} + t_{adapterGetValueCallReply}$

2. Die Zeit $t_{proxyToProcess} = t_{proxyEnqueue} + t_{proxyDequeue} + t_{proxyProcess}$ gibt die Summe der Zeiten an, die für die Verarbeitung im Proxy benötigt werden.

Insgesamt gilt: $t_{aSProcess} \geq 2 * t_{proxyEnqueue} + t_{proxyEnqueueGetValueCall} + t_{proxySetCall1} + t_{proxyGetValueCall} + t_{proxyEnqueue} + t_{proxyDequeue} + t_{proxyGetValueCallReply}$
(R9) Im Timeout-Fall ist die Summe aus der Verarbeitungszeit des Proxys und der Zeit für die Erkennung und Behandlung eines Timeouts höchstens so groß wie die Verarbeitungszeit des abstrakten Sensors:
$t_{processTimeout} = t_{proxySetCall3} + t_{FromProxyToAdpapter}$
$t_{aSProcessTimeout} \geq t_{Timeout1}$
$t_{aSProcessTimeout} \geq t_{processTimeout} + t_{Timeout2}$
$t_{aSProcessTimeout} \geq t_{processTimeout} + t_{Timeout3} + t_{proxyEnqueue}$
$t_{aSProcessTimeout} \geq t_{processTimeout} + t_{Timeout4} + t_{proxyEnqueue} + t_{proxyDequeue}$
(R10) Im Fehler-Fall ist die Summe aus der Verarbeitungszeit des Proxys und der Zeit für die Fehlerbehandlung höchstens so groß wie die Verarbeitungszeit des abstrakten Sensors:
$t_{processError} = t_{proxySetCall2} + t_{FromProxyToAdpapter}$
$t_{aSprocessError} \geq t_{Error1}$
$t_{aSprocessError} \geq t_{processError} + t_{Error2}$
$t_{aSprocessError} \geq t_{processError} + t_{Error3} + t_{proxyEnqueue}$
$t_{aSprocessError} \geq t_{processError} + t_{Error4} + t_{proxyEnqueue} + t_{proxyDequeue}$

Lebendigkeit:

Im Gegensatz zum abstrakten Controller sind die Lebendigkeitsanforderungen komplexer, da mehr Objekte an der Verarbeitung beteiligt sind. Die Anforderungen werden für jedes Objekt angegeben.
(L1) Auf jeden *getValueCall* der periodischen Task eines Reglers folgt ein Return eines *SensorProxys*. Das bedeutet, auf einen *getValueCall*-Aufruf von *O1* folgt irgendwann ein Return von *O2*.
(L2) Nach jedem *getValueCall* eines *SensorProxys* an einen *SensorAdapter* folgt ein Return. Somit folgt auf einen *getValueCall*-Aufruf von *O2* an *O3* schließlich eine Antwort von *O3*.
(L3) Jeder *getValueCall* eines *SensorAdapters* an einen Sensor wird durch ein Return beantwortet. Das bedeutet, dass auf einen *getValueCall*-Aufruf von *O3* an *O4* letztlich ein Return durch *O4* abgesetzt wird.
(L4) Auf jeden *setValueCall* der periodischen Task eines Reglers folgt ein Return eines *ActorProxys*. Auch ein *setValueCall* zwischen *O1* und *O5* darf nicht unbeantwortet bleiben.
(L5) Nach jedem *setValueCall* eines *ActorProxys* an einen *ActorAdapter* folgt ein Return, d. h. auch ein *setValueCall* zwischen *O5* und *O6* muss beantwortet werden.
(L6) Jeder *setValueCall* eines *ActorAdapters* an einen Aktor wird durch ein Return beantwortet. Schließlich erfolgt auch auf einen *setValueCall* von *O6* an *O7* ein Return.

11.2 Übersetzung der Entwurfsmuster-Kombination zum Verfeinerungsmuster des verteilten Reglers nach cTLA

In diesem Abschnitt werden die Übersetzungen der einzelnen Diagramme des OOD zum Verfeinerungsmuster des verteilten Reglers angegeben, die – ausgehend vom Sequenzdiagramm – erstellt werden. In Abbildung 11.6 werden die zur Übersetzung eingeführten Prozesse aufgeführt.

Der Prozesstyp *concreteController* übersetzt die Klasse *periodicTask*. Der Prozesstyp *Sensor* übersetzt den konkreten Sensor des Regelkreises. Zwischen dem Prozesstyp *concreteController* und dem Prozesstyp *Sensor* werden die Prozesstypen *SensorProxy* und *SensorAdapter* eingeführt, um jeweils ein Objekt der Klasse *SensorProxy* bzw. *SensorAdapter* zu übersetzen. Auch bei den Entwurfsmustern wird der Prozesstyp *SequenceDiagram* verwendet, um zu prüfen, ob die Aktionen eines Prozesses zum gegenwärtigen Zeitpunkt schalten dürfen. Das erste Subsystem, das als *SubsystemController* bezeichnet wird, bilden die Prozesstypen *concreteController* und *SequenceDiagram*.

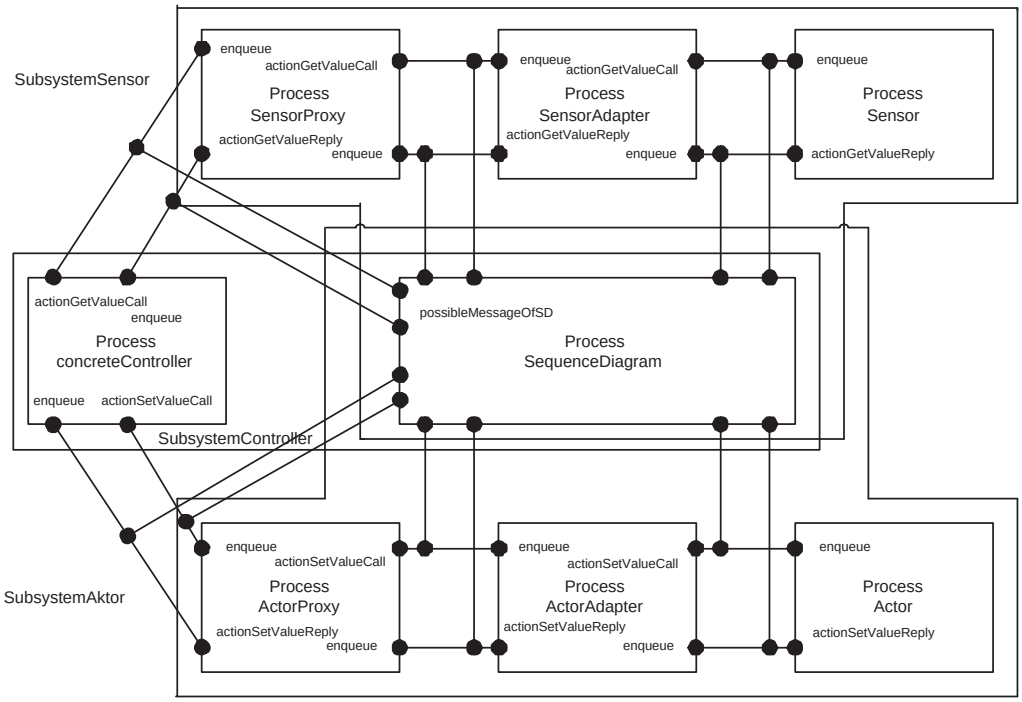


Abbildung 11.6: Die Prozesse für die Übersetzung der Entwurfsmuster-Kombination und deren Kopplung

```
VARIABLES
 ...
 controllerState : { "init", "stopped", "wait", "waitSensor", "getReturnReceived",
 "getReturnDequed", "valuesComputed", "waitActuator", "setReturnReceived",
 "s1", "s2", "s3", "s4", "s5", "s6", "s7", "s8", "s9", "s10", "s11", "s12" };
 ...
```

Abbildung 11.7: Die Zustandsvariable `controllerState` des Prozesses `concreteController`

Das zweite Subsystem, *SubsystemSensor*, wird aus den Prozesstypen *SensorProxy*, *SensorAdapter*, *Sensor* und *SequenceDiagram* komponiert, die im Verlauf dieses und des nächsten Kapitels ausführlich betrachtet werden. Die Abbildung 11.6 zeigt, dass jeweils die Aktionen *enqueue*, *getValueCall* und *isPermittedMessageOfSD* in einer Systemaktion miteinander gekoppelt werden.

Es handelt sich hierbei um ein verteiltes System, da das Objekt *periodicTask* mit den Proxy-Objekten einen Adressraum bildet, während sich die Sensor-Task und die Aktor-Task jeweils in einem anderen Adressraum befinden.

Es wird davon ausgegangen, dass die Implementierung des Controllers in einer realzeitfähigen objektorientierten Programmiersprache – etwa Java – erfolgt. Der Code wird in einer realzeitfähigen virtuellen Maschine ausgeführt, die wiederum auf einem realzeitfähigem Betriebssystem abläuft. Unterbrechungen sind nicht an allen beliebigen Stellen zulässig, da dadurch harte Realzeitanforderungen verletzt werden können. Realzeit-Betriebssysteme – wie etwa RT-Linux– gestatten es, die Verarbeitung von Unterbrechungen zu verhindern bzw. auf spätere Zeitpunkte zu verschieben [Abb02], um die Deadlines der gegenwärtigen Verarbeitung einer Task nicht zu gefährden.

Für die Klasse *Actor* wird der Prozesstyp *Actor* zur Übersetzung eingeführt. Zur Kommunikation zwischen dem Prozess *SensorAdapter* und *concreteController* werden die Prozesstypen *ActorProxy* und *ActorAdapter* aufgenommen, um jeweils ein Objekt der Klasse *ActorProxy* bzw. *ActorAdapter* zu übersetzen. Die Prozesstypen *ActorProxy*, *ActorAdapter*, *Actor* und *SequenceDiagram* bilden das dritte Subsystem, *SubsystemAktor*. Diese Prozesstypen werden im weiteren Verlauf

der Arbeit nicht weiter betrachtet. Das cTLA-System, das bei der Übersetzung der Klassen der Entwurfsmuster-Kombination nach cTLA entsteht, wird im Folgenden als Feinsystem bezeichnet. Das zugehörige Grobsystem, das durch die cTLA-Prozesse für das Analysemuster gebildet wird, wurde in Abschnitt 10.3 vorgestellt.

11.2.1 Übersetzung der Klasse *periodicTask* nach cTLA

Im Folgenden wird auf die Übersetzung eines Objektes der Klasse *periodicTask* durch den Prozesstyp *concreteController* eingegangen. Da der Prozesstyp ein ähnliches Verhalten wie der Prozesstyp *abstractController* besitzt, erbt der Prozesstyp *concreteController* vom Prozesstyp *abstractController* sämtliche Aktionen und Zustandsvariablen. Dafür wird das **EXTENDS** Schlüsselwort im Header des Prozesstyps verwendet.

Die Zustandsvariable *controllerState* enthält – neben den bereits bekannten Zuständen für die Zustandsvariable aus dem Prozess *abstractController* – weitere Zustände, wie in Abbildung 11.7 gezeigt. Die zusätzlich eingeführten Zustände besitzen die Bezeichner "s1" bis "s12".

Der Prozesstyp *concreteController* besitzt zusätzliche Aktionen zur Modellierung der Unterbrechung und dem Versetzen einer Task in einen Schlafzustand, die auch in Abbildung 11.8 gezeigt werden. Bei der Spezifikation der Aktionen sind die Zustandsvariablen, die unverändert bleiben, aus Platzgründen nicht immer vollständig angegeben und werden durch Punkte abgekürzt. Bei der Aktion *enqInterrupt* sind die unveränderten Zustandsvariablen exemplarisch angegeben.

Durch die Aktionen *enqInterrupt* und *deqInterrupt* werden das Einfügen und das Entfernen von Unterbrechungsereignissen aus der Warteschlange modelliert, die an die Klasse *concreteController* abgesetzt werden. Unterbrechungen der zyklischen Verarbeitung der Klasse *periodicTask* sind nur aus den Zuständen "waitSensor", "valuesComputed" und "waitActuator" möglich, wobei jeweils die Zustände "s1", "s2", "s5", "s6", "s9", "s10" betreten werden.

Die Aktionen *enqResume* und *deqResume* modellieren das Einfügen und das Entfernen von Ereignissen zur Fortsetzung der Verarbeitung einer Task, wenn die Verarbeitung zuvor unterbrochen wurde. Diese Aktionen dürfen daher nur schalten, falls die Zustandsvariable *state* einen der Werte "s2", "s3", "s6", "s7", "s10", "s11" angenommen hat. Die Aktionen *sleep* und *awake* modellieren Aufrufe zum Schlafen und zum Aufwecken einer Task. Eine Schaltung der Aktion *sleep* ist nur bei Zuständen möglich, für die die Zustandsvariable *state* den Wert "waitSensor", "valuesComputed" bzw. "waitActuator" hat. Die Aktion *awake* darf hingegen nur bei den Zuständen, für die $state \in \{''s4'', ''s8'', ''s12''\}$ gilt, schalten. Keine der hier vorgestellten Aktionen dieses Prozesstyps hat einen Aktionenparameter, d. h. es erfolgt keine Kopplung mit einer Aktion eines anderen Prozesses des *SubsystemSensor*.

11.2.2 Übersetzung der Klasse *Sensor* nach cTLA

In diesem Abschnitt wird die Übersetzung eines Objektes der Klasse *Sensor* behandelt. Die Klasse *Sensor* ist eine Verfeinerung der Klasse *abstractSensor*, die in Abschnitt 10.2 vorgestellt worden ist. Der Prozesstyp *Sensor* hat einen ähnlichen Aufbau wie der Prozesstyp *abstractSensor* in Bezug auf die normale Verarbeitung. Allerdings werden keine Aktionen für die fehlerbehaftete Verarbeitung und die Zeitverletzungen spezifiziert, da diese im Prozesstyp *SensorProxy* behandelt werden. In Abbildung 11.9 wird der Prozesstyp, der die Klasse *Sensor* übersetzt, gezeigt. Dieser Prozesstyp besitzt die Zustandsvariablen *sSensor*, *qu* und x, um den Kontrollzustand, die Warteschlange und das Attribut x zu modellieren. Weiterhin wird das Initialisierungsprädikat *INIT* in der Abbildung 11.9 gezeigt, das die Zustandsvariable x auf null setzt.

Die Aktion *enqueue*, die auch in Abbildung 11.9 gezeigt ist, nimmt den *getValue*-Aufruf des Prozesses *SensorAdapter* entgegen. Die Aktion *dequeue* mit dem Parameter *currentEventOccurrence* entfernt einen *getValueCall*-Aufruf aus der Warteschlange und weist das Ergebnis der Zustandsvariablen x zu. Weiterhin ändert sich der Wert der Zustandsvariablen *sSensor* von "enqueued" auf "dequeued".

Die Aktion *process* modelliert die Verarbeitung durch das *Sensor*-Objekt. Sie besitzt den Parameter *currentEventOccurrence*, der mit der Zeichenkette "O4.actionProcess" belegt wird.

```
PROCESS concreteController EXTENDS
        abstractController
ACTIONS

enqInterrupt(message : InterruptMessage)  ≜ ! Interruptaufruf in die Queue aufnehmen
       ∧ state = "waitSensor" ∨ state = "valuesComputed" ∨ state = "waitActuator"
       ∧ state′ = IF state = "waitSensor"
                  THEN "s1"
                  ELSE IF state = "valuesComputed"
                  THEN "s5"
                  ELSE "s9"
       ∧ qu′ = Append(qu, message)
       ∧ UNCHANGED ⟨x, y, sync, ctrlStartActReadx, ctrlActReadxActReady⟩
       ∧ UNCHANGED ⟨exists01, exists02, exists03, value01, value02, value03⟩;
deqInterrupt ≜  ! Entferne Interruptaufruf aus der Queue
       ∧ state = "s1" ∨ state = "s5" ∨ state = "s9"
       ∧ head(qu) = ''interrupt''
       ∧ state′ = IF state = "s1"
                  THEN "s2"
                  ELSE IF state = "s5"
                  THEN "s6"
                  ELSE "s10"
       ∧ qu′ = Tail(qu)
       ∧ UNCHANGED ⟨x, y, sync, ctrlStartActReadx, ctrlActReadxActReady ... ⟩;
enqResume(message : ResumeMessage) = ! Resumeaufruf in die Queue entgegennehmen
       ∧ state = "s2" ∨ state = "s6" ∨ state = "s10"
       ∧ state′ = IF state = "s2"
                  THEN "s3"
                  ELSE IF state = "s6"
                  THEN "s7"
                  ELSE "s11"
       ∧ qu′ = Append(qu, message)
       ∧ UNCHANGED ⟨x, y, sync, ctrlStartActReadx, ctrlActReadxActReady, ... ⟩;
deqResume ≜ ! Resumeaufruf aus der Queue entfernen
       ∧ state = "s3" ∨ state = "s7" ∨ state = "s11"
       ∧ head(qu) = ''resume''
       ∧ state′ = IF state = "s3"
                  THEN "waitSensor"
                  ELSE IF state = "s7"
                  THEN "valuesComputed"
                  ELSE "waitActuator"
       ∧ qu′ = Tail(qu)
       ∧ UNCHANGED ⟨x, y, sync, ctrlStartActReadx, ctrlActReadxActReady, ... ⟩;
sleep ≜  ... ! Versetzen einer periodischen Task in den Schlafzustand

awake ≜ ! Aufwecken einer periodischen Task
       ∧ state = "s4" ∨ state = "s8" ∨ state = "s12"
       ∧ state′ = IF state = "s4"
                  THEN "waitSensor"
                  ELSE IF state = "s8"
                  THEN "valuesComputed"
                  ELSE "waitActuator"
       ∧ UNCHANGED ⟨x, y, qu, sync, ctrlStartActReadx, ctrlActReadxActReady, ... ⟩;
END  concreteController
```

Abbildung 11.8: Der Prozesstyp `concreteController`

```
PROCESS Sensor

 IMPORT Sequence;

VARIABLES
sSensor : {"init"," enqueued", "dequeued", "processed"};
qu : Sequence OF Real;
x : Real;

INIT ≜
      ∧ sSensor = "init"
      ∧ qu = ≪ ≫
      ∧ x = 0;
ACTIONS
 enqueue(m : message) ≜ ! Einfügen des getValue-Aufrufes in die Queue
     ∧ sSensor = "init"
     ∧ sSensor′ ="enqueued"
     ∧ qu′ = append(qu, value)
     ∧ UNCHANGED x;
dequeue(currentEventOccurrence : String) ≜ ! Entfernen des
getValue-Aufrufes aus der Queue
     ∧ currentEventOccurrence = "O4.actionDequeue"
     ∧ sSensor = "enqueued"
     ∧ sSensor′ = "dequeued"
     ∧ qu′ = tail(qu)
     ∧ UNCHANGED x;
 process(currentEventOccurrence : String) ≜ ! Ermittlung des Sensorwertes
     ∧ currentEventOccurrence = "O4.actionProcess"
     ∧ sSensor = "dequeued"
     ∧ sSensor′ = "processed"
     ∧ x′ = 5*x
     ∧ UNCHANGED qu;
 getValueCallReply(value : Real; currentEventOccurrence : String) ≜ ! Rückübtr. Sensorwert
     ∧ currentEventOccurrence = "O4.actionGetValueCallReply"
     ∧ x = value
     ∧ sSensor = "processed"
     ∧ sSensor′ = "init"
     ∧ UNCHANGED ⟨x, qu⟩;
END Sensor
```

Abbildung 11.9: Der Prozesstyp Sensor

Durch die Aktion *getValueCallReply*, welche ebenfalls in Abbildung 11.9 aufgeführt ist, wird die Antwort des *Sensor*-Objektes an ein *SensorAdapter*-Objekt modelliert. Auch diese Aktion besitzt den Aktionenparameter *currentEventOccurrence*, der mit der Zeichenkette "O4.getValueCallReply" belegt wird. Weiterhin besitzt die Aktion den Parameter *value*, um den Istwert des *Sensor*-Objektes weiterzureichen. Die persistenten Wartezeiten für die Aktionen des Prozesstyps *Sensor* werden durch den Prozesstyp *SensorTimes* aus Abbildung 11.11 (s. Anhang D.1) angegeben. Zunächst werden die Aktionen des Prozesstyps deklariert. Für die Aktion *enqueue* wird die maximale, persistente Wartezeit *tSensorEnqueue* definiert. Auch die Definition der maximalen, persistenten Wartezeit *tSensorDequeue* für die Aktion *dequeue* wird angegeben. Für die Aktionen *process* und *getValueCallReply* werden die maximalen, persistenten Wartezeiten *tSensorProcess* und *tSensorGetValueCallReply* definiert.

11.2.3 Übersetzung der Klasse *SensorAdapter* nach cTLA

Das Statechart-Diagramm eines Objektes der Klasse *SensorAdapter* ist aus Platzgründen in Abbildung C.1 des Anhangs gezeigt.

```
PROCESS SensorAdapter ! Spezifikation des Prozesstyps SensorAdapter
 IMPORT Sequence;
VARIABLES

 sAdapter : {"init","enqueued","dequeued", "processed", "called",
             "returnEnqueued", "returnDequeued"};
 qu : Sequence OF Real;
 x : Real;
INIT ≜ ! Initialisierungsprädikat
     ∧ sAdapter = "init";
     ∧ qu = ≪ ≫
     ∧ x = 0;

ACTIONS
 enqueue(m : message) ≜ ! nimmt getValue-Aufruf entgegen
     ∧ sAdapter = "init"
     ∧ sAdapter′ = "enqueued"
     ∧ qu′ = append(qu, value)
     ∧ UNCHANGED ⟨x⟩;
 dequeue(currentEventOccurence : String) ≜ ! entfernt getValue-Aufruf aus der Queue
     ∧ currentEventOccurence = "O3.actionDequeue"
     ∧ sAdapter = "enqueued"
     ∧ sAdapter′ = "dequeued"
     ∧ qu′ = tail(qu)
     ∧ x′ = head(qu);
 process(currentEventOccurence : String) ≜ ! Verarbeitung im Adapter
     ∧ currentEventOccurence = "O3.actionProcess"
     ∧ sAdapter = "dequeued"
     ∧ sAdapter′ = "processed"
     ∧ UNCHANGED ⟨qu, x⟩;
 getValueCall(currentEventOccurence : String, value : Real) ≜ ! getValue Aufruf an Proxy
     ∧ currentEventOccurence = "O3.actionGetValueCall"
     ∧ sAdapter = "processed"
     ∧ sAdapter′ = "called"
     ∧ UNCHANGED ⟨qu, x⟩
 getValueCallReply(value : Real; currentEventOccurence : String) ≜
     ∧ currentEventOccurence = "O3.actionGetValueCallReply"
     ∧ value = head(qu)
     ∧ sAdapter = "called"
     ∧ sAdapter′ = "init"
     ∧ qu′ = tail(qu)
     ∧ UNCHANGED ⟨qu, x⟩;
END SensorAdapter
```

Abbildung 11.10: Der Prozesstyp `SensorAdapter`

Der Prozesstyp *SensorAdapter* aus Abbildung 11.10 besitzt die Zustandsvariablen *sAdapter*, *qu* und *x*. Die Zustandsvariable *qu* hat den Datentyp Sequence. Die Zustandsvariable *sAdapter* darf unterschiedliche Werte aus der Menge annehmen, welche den Kontrollzuständen des Statechart-Diagramms eines *SensorAdapter*-Objektes entsprechen. Im Initialisierungsprädikat *INIT* wird die Zustandsvariable *x* auf null gesetzt. Weiterhin wird die Zustandsvariable *qu* durch die leere Sequence << >> initialisiert. Die Zustandsvariable *sAdapter* wird auf den Zustand "init", der dem Zustand *init* des Statechart-Diagramms entspricht, gesetzt.

Die Aktionen *enqueue* und *dequeue* aus Abbildung 11.10 behandeln den Zugriff auf die Warteschlange *qu* eines *SensorAdapter*-Objektes. Die Aktion *enqueue* hat den Aktionenparameter *m* vom Datentyp *message*, um einen *getValue*-Aufruf entgegenzunehmen. Hierbei wird kein konkreter

```
PROCESS SensorTimes
ACTIONS
enqueue;
dequeue;
process;
getValueCallReply;
P MAX TIME enqueue : tSensorEnqueue;
P MAX TIME dequeue : tSensorDequeue;
P MAX TIME process : tSensorProcess;
P MAX TIME getValueCallReply : tSensorGetValueCallReply;
END SensorTimes
```

Abbildung 11.11: Die Wartezeiten des Prozesstyps `Sensor`

Wert übergeben, da ein *getValue*-Aufruf prinzipiell parameterlos ist. Beim Schalten der Aktion wird die Zustandsvariable *sAdapter* von "init" auf "enqueued" gesetzt. Die Aktion *dequeue* besitzt den Aktionenparameter *currentEventOccurrence*, der mit dem Wert "O3.actionDequeue" belegt wird, um sicherzustellen, dass die Warteschlange zum durch das Sequenzdiagramm vorgegebenen Zeitpunkt geleert wird. In der Aktion wird die Zustandsvariable *sAdapter* von "enqueued" auf "dequeued" gesetzt. Außerdem wird die Warteschlange mit der *Tail*-Operation geleert. Durch die Aktion *process* in Abbildung 11.10 wird der Verarbeitungsvorgang im *SensorAdapter*-Objekt modelliert. Das Schalten der Aktion führt dazu, dass sich der Wert der Zustandsvariable *sAdapter* von "dequeued" auf "processed" ändert. Der Aktionenparameter *currentEventOccurence* wird mit dem Wert "O3.actionProcess" belegt. Die Zustandsvariablen *qu* und *x* werden nicht verändert.

Die Aktion *getValueCall* spezifiziert den Aufruf *getValueCall* an ein *Sensor*-Objekt. Die Aktion besitzt den Aktionenparameter *currentEventOccurence*, der mit dem Wert "O3.actionGetValueCall" belegt wird. Beim Schalten der Aktion *getValueCall* ändert sich die Zustandsvariable *sAdapter* vom Wert "processed" auf den Wert "called". Da der *getValue*-Aufruf keinen Wert an das *SensorProxy*-Objekt überträgt, ist kein zweiter Aktionenparameter notwendig. Die Aktion *getValueCallReply*, die der Übermittlung des Istwertes aus dem Sensor dient, besitzt die zwei Aktionenparameter *currentEventOccurence* und *value*. Der Aktionenparameter *value* überträgt den Istwert zurück an das *SensorProxy*-Objekt. Nachdem die Aktion geschaltet hat, wird der Wert der Zustandsvariable *sAdapter* von "called" auf "init" gesetzt. Durch die Aktion *getValueReply* wird die Antwort auf den *getValueCallReply*-Aufruf an das *SensorProxy*-Objekt modelliert.

11.2.4 Wartezeiten des Prozesstyps *SensorAdapter*

Da an ein *SensorAdapter*-Objekt Realzeitanforderungen bestehen, müssen P MAX TIME Konstrukte spezifiziert werden. Die maximalen, persistenten Wartezeiten des Prozesstyps *SensorAdapter* werden in Abbildung 11.12 durch den Prozesstyp *SensorAdapterTimes* (s. Anhang D.2) angegeben. Zuerst werden die Aktionen des Prozesstyps *SensorAdapter* deklariert, bevor die persistenten, maximalen Wartezeiten für die einzelnen Aktionen aufgeführt werden.

11.2.5 Übersetzung der Klasse *SensorProxy* nach cTLA

Nun soll die Übersetzung eines Objektes der Klasse *SensorProxy* behandelt werden. Die Klasse *SensorProxy* ist verhältnismäßig komplex, da sie auch Fehler und Zeitüberschreitungen, die bei der Kommunikation zwischen *concreteController* und *Sensor*-Objekten auftreten können, behandelt. Deshalb wird ihre cTLA-Spezifikation in mehrere Abbildungen zerlegt. Die Abbildung 11.13

```
PROCESS SensorAdapterTimes

ACTIONS
    enqueue;

    dequeue;

    ...

P MAX TIME enqueue : tAdapterEnqueue;

P MAX TIME dequeue : tAdapterDequeue;

P MAX TIME process : tAdapterProcess;

P MAX TIME getValueCall : tAdapterGetValueCall;

P MAX TIME getValueCallReply : tAdapterGetValueCallReply;

END SensorAdapterTimes
```

Abbildung 11.12: Die persistenten, maximalen Wartezeiten des `SensorAdapter`

zeigt den Prozess *SensorProxy*, der zur Übersetzung eingeführt wird. Der Prozess enthält die Zustandsvariable x zur Aufnahme des Istwertes des *Sensor*-Objektes. Die Zustandsvariable *qu* vom Datentyp *Sequence* wird zur Modellierung der Warteschlange eingeführt. Die Zustandsvariable *sProxy* besitzt als Datentyp eine Menge, die alle Kontrollzustände aus dem in Abbildung 11.5 gezeigten Statechart-Diagramm enthält. Dies sind beispielsweise die Zustände "init", "readyForCall" und "called". Es wird angenommen, dass ein *SensorProxy*-Objekt zu einem bestimmten Zeitpunkt nur Aufrufe von einem anderen Objekt erhält. Daher wird auf die Einführung einer Zustandsvariable zur Synchronisation verzichtet. Durch das Initialisierungsprädikat wird die Zustandsvariable *sProxy* auf "init", die Sequence *qu* auf << >> und die Zustandsvariable x auf null gesetzt.

Mit Ausnahme der Aktion *enqueue* besitzen alle Aktionen den Parameter *currentEventOccurrence*, um sicherzustellen, dass jede Ausführung einer Aktion konform mit dem Trace des Sequenzdiagramms geschieht. Durch die Aktionen *enqueue* und *dequeue* aus der Abbildung 11.13 werden Zugriffe auf die Warteschlange des Statechart-Diagramms modelliert, die Aufrufe entgegennehmen und für die weitere Verarbeitung bereitstellen. Die Aktion *enqueue* nimmt sowohl die Aufrufe vom *periodicTask*-Objekt – durch den Prozess *concreteController* übersetzt – als auch die Rückantworten auf einen *getValueCall* eines *SensorAdapter*-Prozesses entgegen. Nach dem Schalten der Aktion ändert sich der Wert der Zustandsvariable *sProxy* von "init" auf "callEnqueued" und die Warteschlange *qu* ist gefüllt. Für die Rückantwort und den Fehlerfall ergeben sich abweichende Belegungen.

Die Aktion *dequeue* entfernt Aufrufe des *concreteController*-Objektes und Antworten des *SensorAdapter*-Objektes aus der Warteschlange. Durch die Schaltung der Aktion wird der Wert der Zustandsvariable *sProxy* von "callEnqueued" auf "callDequeued" gesetzt. Auch hier sind im Fehlerfall abweichende Belegungen möglich.

Zur Modellierung der von einem *SensorProxy* durchgeführten Verarbeitung, wie etwa dem Auffinden eines Kommunikationspartners, wird die Aktion *process*, die in Abbildung 11.14 gezeigt ist, verwendet. Sie ändert beim Schalten den Wert der Zustandsvariable *sProxy* von "callDequeued" auf "processed". Die Aktion ist nur schaltbereit, wenn der Aktionenparameter *currentEventOccurence* mit dem Wert "O2.actionProcess" belegt wird.

Die Aktionen *setCall1*, *setCall2* und *setCall3* beeinflussen, ob der Zweig zur normalen Verarbeitung, zur Fehlerbehandlung bzw. zur Behandlung einer Zeitüberschreitung betreten wird. Die Abbildung 11.15 zeigt exemplarisch die Aktion *setCall1* bei deren Schalten der Wert der Zustandsvariable *sProxy* von "processed" auf "readyForCall" gesetzt wird. Damit die Aktion schalten darf muss der Aktionenparameter *currentEventOccurence* mit dem Wert "O2.actionSetCall1" belegt werden. Die Aktionen *setCall2* und *setCall3* sind sehr ähnlich aufgebaut. Bei der Aktion *setCall2*, die den Fehlerfall modelliert, wird beispielsweise der Wert der Zustandsvariable *sProxy* von "processed" auf "readyForError" gesetzt.

```
PROCESS SensorProxy

 IMPORT Sequence;

VARIABLES
 sProxy : {"init", "callEnqueued", "callDequeued", "processed", "readyForCall","called",
 "returnEnqueued", "returnDequeued", "readyForError","calledError","returnEnqError",
 "Error", "returnDeqError", "readyForTimeout", "calledTimeout", "returnEnqTimeout",
 "returnDeqTimeout", "Timeout"};
 qu : Sequence;
 x : Real;
INIT ≜
        ∧ sProxy = "init"
        ∧ qu = ≪ ≫
        ∧ x = 0;
ACTIONS

enqueue(value : Real) ≜
 ∧ (sProxy = "init" ∨ sProxy =
 "called" ∨ sProxy = "calledError" ∨ sProxy = "calledTimeout")
 ∧ sProxy′ = IF ("sProxy" = "calledTimeout")
              THEN "returnEnqTimeout"
              ELSE IF ("sProxy" = "called")
              THEN "returnEnqueued"
              ELSE IF ("sProxy" = "calledError")
              THEN "returnEnqError"
              ELSE "callEnqueued"
 ∧ qu′ = Append(qu, value)
 ∧ UNCHANGED x;
dequeue(currentEventOccurence : String) ≜
  ∧ currentEventOccurence = "02.actionDequeue"
  ∧ sProxy = ("callEnqueued" ∨ sProxy = "returnEnqueued" ∨ sProxy = "returnEnqError" ∨
  sProxy = "returnEnqTimeout")
  ∧ sProxy′ = IF (sProxy = "callEnqueued")
               THEN "callDequeued"
               ELSE IF sProxy = "returnEnqueued"
               THEN "returnDequeued"
               ELSE IF (sProxy = "returnEnqError")
               THEN "returnDeqError"
               ELSE "returnDeqTimeout"
  ∧ qu′ = tail(qu)
  ∧ x′ = head(qu);
getValueCall(currentEventOccurence : String) ≜ ... s. Text
getValueCallReply(value : Real; currentEventOccurence : String) ≜ ... s. Text
process(currentEventOccurence : String) ≜ ... s. Text
setCall1(currentEventOccurence : String) ≜ ... s. Text
setCall2(currentEventOccurence : String) ≜ ... s. Anhang
error1(currentEventOccurence : String) ≜ ... s. Text
error2(currentEventOccurence : String) ≜ ... s. Text
error3(currentEventOccurence : String) ≜ ... s. Anhang
error4(currentEventOccurence : String) ≜ ... s. Anhang
timeout1(currentEventOccurence : String) ≜ ... s. Anhang
...
END SensorProxy
```

Abbildung 11.13: Der Prozesstyp SensorProxy

```
process(currentEventOccurence : String) ≜
 ∧ currentEventOccurence = "O2.actionProcess"
 ∧ sProxy = "callDequeued"
 ∧ sProxy′ = "processed"
 ∧ UNCHANGED ⟨qu, x⟩;
```

Abbildung 11.14: Die Aktion `process` des Prozesstyps `SensorAdapter`

```
setCall1(currentEventOccurence : String) ≜
          ∧ currentEventOccurence = "O2.actionSetCall1"
          ∧ sProxy = "processed"
          ∧ sProxy′ = "readyForCall"
          ∧ UNCHANGED ⟨qu, x⟩;
```

Abbildung 11.15: Die Aktion `getValueCall`

Die Aktion *getValueCall* aus Abbildung 11.16 modelliert den *getValue*-Aufruf des *SensorProxy*-Prozesses an den Prozess *SensorAdapter*. Die Aktion ist für die Werte "called", "readyForTimeout" und "readyForError" der Zustandsvariable *sProxy* schaltbereit. Die Aktion besitzt die Aktionenparameter *value* und *currentEventOccurence*. Der Aktionenparameter *value* wird mit dem Wert der Zustandsvariablen x belegt, während die Belegung des Aktionenparameters *currentEventOccurence* mit dem Wert "O2.actionCall" erfolgt. In Abhängigkeit vom Zustand vor dem Schalten ändert sich der Wert der Zustandsvariable *sProxy* auf "called", "calledError" oder "calledTimeout".

```
getValueCall(value : Real; currentEventOccurence : String) ≜
 ∧ currentEventOccurence = "O2.actionCall"
 ∧ (sProxy = "readyForTimeout" ∨ sProxy = "readyForError" ∨ sProxy = "readyForCall")
 ∧ value = x
 ∧ sProxy′ = IF (sProxy = "readyForTimeout")
             THEN "calledTimeout"
             ELSE IF (sProxy = "readyForError")
             THEN "calledError"
             ELSE "called"
 ∧ UNCHANGED ⟨qu, x⟩;
```

Abbildung 11.16: Die Aktion `getValueCall`

Die Aktion *getValueCallReply* aus Abbildung 11.17 hingegen modelliert die Rücklieferung des Sensorwertes an den *concreteController*. Zunächst besitzt die Aktion die Aktionenparameter *value* und *currentEventOccurence*. Die Aktion darf schalten, wenn die Zustandsvariable *sProxy* den Wert "Timeout", den Wert "returnDeqTimeout", den Wert "Error", den Wert"returnDequeued" oder den Wert "returnDeqError" besitzt. Die Aktionenparameter *value* und *currentEventOccurence* werden jeweils mit dem Wert von x – also dem Istwert des *Sensor* Objektes bzw. "O2.actionGetValueCall-Reply" – belegt.

Das Verhalten im Fehlerfall wird durch die Aktionen *error1*, *error2*, *error3* und *error4* modelliert. Die Aktion *error1* beschreibt einen Fehler, der noch vor dem *getValue*-Operationsaufruf – nämlich aus dem Zustand *readyForError* – auftritt. Durch die Aktion *error2* wird ein Fehler des Zustands mit *state* =″ *error*2″ modelliert, der unmittelbar vor dem *getValueCall* auftritt. Mit den Aktionen *error3* und *error4* werden Fehler spezifiziert, die nach dem Empfang des Returns auf die *getValue*-Nachricht eintreffen. Es existieren ähnliche Aktionen für Zeitüberschreitungen – nämlich *timeout1* bis *timeout4* –, die hier aus Platzgründen nicht näher betrachtet werden.

Nun soll in Abbildung 11.18 exemplarisch die Aktion *error1* angegeben werden. Sie wird schaltbereit, wenn die Zustandsvariable *sProxy* den Wert "readyForError" hat. Der Aktionenparameter

```
getValueCallReply(value : Real; currentEventOccurence : String) ≜
        ∧ currentEventOccurence = "O2.actionGetValueCallReply"
        ∧ value = x
        ∧ sProxy = "Timeout" ∨ sProxy = "returnDeqTimeout" ∨ sProxy = "returnDequeued"
             ∨ sProxy = "Error" ∨ sProxy = "returnDeqError"
        ∧ sProxy′ = "init"
        ∧ UNCHANGED ⟨qu, x⟩;
```

Abbildung 11.17: Die Aktion `getValueCallReply`

currentEventOccurence dieser Aktion wird mit "O2.error1" belegt. Nach dem Schalten der Aktion ändert sich der Wert der Zustandsvariable *sProxy* von "readyForError" auf "Error".

11.2.6 Prozesstyp *SensorProxyTimes*

Auch der Prozesstyp *SensorProxy* benötigt persistente, maximale Wartezeiten, um seine Realzeitanforderungen zu erfüllen. Der in Abbildung 11.19 gezeigte Prozesstyp (s. Anhang D.3) enthält die Spezifikation dieser Wartezeiten. Der Abschnitt **ACTIONS** enthält zunächst die Deklaration der einzelnen Aktionen. Anschließend werden für jede betroffene Aktion die maximalen, persistenten Wartezeiten spezifiziert.

11.2.7 Kopplung der Prozesse des *SubsystemSensor*

Das *SubsystemSensor* koppelt drei cTLA-Prozesse, die den Klassen des objektorientierten Entwurfs entsprechen, einen Prozess, der für das Sequenzdiagramm eingeführt worden ist und drei weitere cTLA-Prozesse, die Wartezeiten an Aktionen spezifizieren. Zur Kopplung der Prozesse wird der Prozesstyp *concreteCompositionOfSub1* eingeführt. Der Prozesstyp kann hier aufgrund der hohen Komplexität der Kopplungen nur in Auszügen angegeben werden, wobei die besonders interessanten Systemaktionen betrachtet werden. Die vollständige Spezifikation ist in Anhang D.4 angegeben. In Abbildung 11.20 ist der Header des Prozesstyps gezeigt.

Zunächst wird der Abschnitt **PROCESSES** betrachtet. Durch den Prozess *O1* wird die periodische Task modelliert. Die Klasse *SensorProxy* wird durch den Prozess *O2* spezifiziert und die Klasse *SensorAdapter* wird durch den Process *O3* modelliert. Der Prozess *O4* schließlich spezifiziert den konkreten Sensor.

Nun sollen die Systemaktionen, die der Kopplung der einzelnen Prozessaktionen dienen, behandelt werden. Alle im Folgenden beschriebenen Systemaktionen enthalten ein Konjunkt mit der Aktion *permittedActionOfSD* des Prozesses *sd*, die den Aktionenparameter *currentEventOccurence* besitzt, damit sichergestellt ist, dass sie nur im Rahmen der zulässigen Traces ausgeführt werden. Dieser Sachverhalt wirkt sich auch auf den Aufbau der jeweiligen Systemaktion aus, da auch sie den Aktionenparameter *currentEventOccurence* erhält. Durch die Systemaktion *ProxyDequeue* wird die gemeinsame Schaltung der Aktion *dequeue* des Prozesses *O2* sowie der Aktion *permittedActionOfSD* unter Einhaltung der Zeiten des Prozesses *TO2* beschrieben, während die übrigen Prozesse stottern. Die prozesslokale Aktion *process* in der Systemaktion *ProxyProcess* und die Aktion *permittedActionOfSD* schalten gleichzeitig unter Berücksichtigung der Wartezeiten von *TO2*

```
error1(currentEventOccurence : String) ≜
        ∧ currentEventOccurence = "O2.error1"
        ∧ sProxy = "readyForError"
        ∧ sProxy′ = "Error"
        UNCHANGED ⟨qu, x⟩;
```

Abbildung 11.18: Die Aktion `error1`

```
PROCESS SensorProxyTimes
ACTIONS
enqueue;
...
P MAX TIME enqueue : tProxyEnqueue;
P MAX TIME dequeue : tProxyDequeue;
P MAX TIME process : tProxyProcess;
P MAX TIME getValueCall : tProxyGetValueCall;
P MAX TIME getValueCallReply : tProxyGetValueCallReply;
P MAX TIME error1 : tProxyError1;
P MAX TIME error2 : tProxyError2;
P MAX TIME error3 : tProxyError3;
P MAX TIME error4 : tProxyError4;
! Alle übrigen maximalen, persistenten Wartezeiten s. Anhang
END SensorProxyTimes
```

Abbildung 11.19: Die persistenten Wartezeiten des Prozesstyps `SensorProxy`

für die Aktion *process*, während die übrigen Prozesse ebenfalls stottern. In der Systemaktion *setCall1* wird die Aktion *setCall1* des Prozesses *O2* mit der Aktion *permittedActionOfSD* gekoppelt. Hierbei müssen die Wartezeiten aus *TO2* berücksichtigt werden. Schließlich soll die Systemaktion *CallToAdapter* betrachtet werden, die den *getValue*-Aufruf vom *SensorProxy*-Objekt an das *SensorAdapter*-Objekt beschreibt. Die Aktion *getValueCall* des Prozesses *O2*, die Aktion *enqueue* des Prozesses *O3* sowie die Aktion *permittedActionOfSD* des Prozesses *sd* schalten gemeinsam unter Berücksichtigung der Zeiten aus den Prozessen *TO2* und *TO3*, während die Prozesse *O4* und *TO4* stottern.

Die Systemaktionen für Fehler und Zeitüberschreitungen – etwa *ProxyTimeout1* bzw. *ProxyError1* – sind auch in Anhang E.1 angegeben. Dabei wird die Spezifikation der Systemaktionen, die Zeitüberschreitungen betreffen, in völliger Symmetrie mit den Aktionen, die Fehler spezifizieren, vorgenommen. In der Abbildung 11.22 hingegen werden Systemaktionen für die Aktionen des

```
PROCESS concreteCompositionOfSub1
PROCESSES
 ! O1 : concreteController; für das Subsystem zunächst nicht relevant
 O2 : SensorProxy;
 TO2 : SensorProxyTimes;
 O3 : SensorAdapter;
 TO3 : SensorAdapterTimes;
 O4 : Sensor;
 TO4 : SensorTimes;
 sd : SequenceDiagram(≪''O2.dequeue'', ''O2.actionProcess'', ''O2.actionGetValueCall'',
 ''O3.enqueue'', ''O3.actionProcess'', ''O3.actionGetValueCall'', ''O4.dequeue'',
 ''O4.actionProcess'', ''O4.actionGetValueCallReply'', ''O3.dequeue'',
 ''O3.actionGetValueCallReply'', ''O2.dequeue'', ''O2.actionGetValueCallReply''≫);
ACTIONS ...
```

Abbildung 11.20: Der Prozesstyp `concreteCompositionOfSub1`

```
ProxyDequeue(currentEventOccurence : String) ≙ ! dequeue Aktion
d. Proxys
    ∧ O2.dequeue(currentEventOccurence)
    ∧ TO2.dequeue
    ∧ O3.stutter
    ∧ TO3.stutter
    ∧ O4.stutter
    ∧ TO4.stutter
    ∧ sd.permittedActionOfSD(currentEventOccurence);
ProxyProcess(currentEventOccurence : String) ≙ ! process Aktion
d. Proxys
    ∧ O2.process(currentEventOccurence)
    ∧ TO2.process
    ∧ O3.stutter
    ∧ TO3.stutter
    ∧ O4.stutter
    ∧ TO4.stutter
    ∧ sd.permittedActionOfSD(currentEventOccurence);
setCall1(currentEventOccurence : String) ≙ ! setCall1 Aktion des
Proxys
    ∧ O2.setCall1(currentEventOccurence)
    ∧ TO2.setCall1
    ∧ O3.stutter
    ∧ TO3.stutter
    ∧ O4.stutter
    ∧ TO4.stutter
    ∧ sd.permittedActionOfSD(currentEventOccurence);
CallToAdapter(val : Real; currentEventOccurence : String) ≙ !
getValue Aufruf des SensorAdapters
    ∧ O2.getcallValueCall(value, currentEventOccurence)
    ∧ TO2.getValueCall
    ∧ O3.enqueue(value)
    ∧ TO3.getValueCall
    ∧ O4.stutter
    ∧ TO4.stutter
    ∧ sd.permittedActionOfSD(currentEventOccurence);
...
```

Abbildung 11.21: Eine Zusammenstellung von Systemaktionen zur nicht-fehlerbehafteten Verarbeitung durch den Prozesstyp `SensorProxy`

Prozesses *SensorAdapter* gezeigt. Die Systemaktion *AdapterProcess* schaltet nur, falls die Aktion *process* des Prozesses *O3* und die Aktion *permittedActionOfSD* des Prozesses *sd* gemeinsam schalten, wobei die Wartezeit aus *TO3* für die Aktion *process* berücksichtigt werden muss. Die übrigen Prozesses stottern. Die Systemaktion *CallToSensor* besitzt einen komplizierteren Aufbau, da sie den *getValue*-Aufruf an den *Sensor*-Prozess weiterleitet, wofür die Aktion *getValueCall* des Prozesses *O3*, die Aktion *enqueue* des Prozesses *O4* und die Aktion *permittedActionOfSD* des Prozesses *sd* unter Berücksichtigung der Wartezeiten aus den Prozessen *TO3* und *TO4* gemeinsam schalten müssen. Alle übrigen Aktionen stottern. Die Aktion *ReturnFromAdapter* modelliert die Übertragung des Istwertes an den *SensorProxy*-Prozess. Der Aufbau der Systemaktion *ReturnFromAdapter*, die den Sensorwert zurück an den Prozess *SensorProxy* überträgt, ist dem der Systemaktion *CallToSensor* sehr ähnlich, wobei im Wesentlichen die prozesslokalen Aktionen *getValueCallReply* des Prozesses *O3*, *enqueue* des Prozesses *O2* und *permittedActionOfSD* des Prozesses *sd* beteiligt sind. Auch hierbei sind wiederum Wartezeiten zu berücksichtigen.

Die Abbildung 11.23 zeigt zwei interessante Systemaktionen für die Verarbeitung durch den Prozesstyp *Sensor*. Die Systemaktion *SensorProcess* schaltet, wenn die prozesslokale Aktion *process*

```
... AdapterProcess(currentEventOccurence : String) ≙ !
Verarbeitung des Adapters
    ∧ O2.stutter
    ∧ TO2.stutter
    ∧ O3.process(currentEventOccurence)
    ∧ TO3.process
    ∧ O4.stutter
    ∧ TO4.stutter
    ∧ sd.permittedActionOfSD(currentEventOccurence);
CallToSensor(value : Real; currentEventOccurence : String) ≙ !
Aufruf des Sensors
    ∧ O2.stutter
    ∧ TO2.stutter
    ∧ O3.getValueCall(value, currentEventOccurence)
    ∧ TO3.getValueCall
    ∧ O4.enqueue(value)
    ∧ TO4.enqueue
    ∧ sd.permittedActionOfSD(currentEventOccurence);
ReturnFromAdapter(value : Real; currentEventOccurence : String) ≙ !
Adapterantwort
    ∧ O2.enqueue(value)
    ∧ TO2.enqueue
    ∧ O3.getValueCallReply(value, currentEventOccurence)
    ∧ TO3.return
    ∧ O4.stutter
    ∧ TO4.stutter
    ∧ sd.permittedActionOfSD(currentEventOccurence);
```

Abbildung 11.22: Eine Auswahl von Systemaktionen für den Prozesstyp `Adapter`

des Prozesses *O4* und die prozesslokale Aktion *permittedActionOfSD* schalten, wobei die Wartezeiten an die Aktion *process* aus dem Prozess *TO4* eingehalten werden müssen. Beim Schalten dieser Systemaktion stottern die übrigen Prozesse. Die Schaltung der Systemaktion *ReturnFromSensor*, welche die Rückübertragung des Sensorwertes an den Prozess *SensorAdapter* modelliert, erfolgt gemeinsam mit den prozesslokalen Aktionen *getValueCallReply* von *O4*, *enqueue* von *O3* und *permittedActionOfSD* von *sd*. Dabei müssen natürlich die in den Prozessen *TO3* und *TO4* spezifizierten Wartezeiten für die prozesslokalen Aktionen *enqueue* und *getValueCallReply* eingehalten werden. In diesem Fall stottern die übrigen Prozesse.

11.2.8 Aktion *TickFS* des Feinsystems

In Abbildung 11.25 ist die Aktion *TickFS* des *SubsystemSensor* gezeigt, die sich durch die Kopplung (s. Abbildung 11.24) der *Tick*-Aktionen der Prozesstypen *SensorProxy*, *SensorAdapter* und *Sensor* ergibt.

Die Aktion *TickFS* sorgt für das Fortschreiten sämtlicher Timervariablen und der Uhrenvariablen *now*. Die Uhrenvariable *now* darf beim Schalten von *TickFS* höchstens um ϵ ansteigen. Die Aktion enthält für jede Aktion des *SubsystemSensor* eine Timervariable. Der Name einer Timervariablen setzt sich aus dem Präfix *Timer*, dem Prozessnamen und dem Aktionennamen zusammen. So bezeichnet die Zustandsvariable *TimerProxyProcess* die Timervariable für die Aktion *process* des Prozesses *SensorProxy*. Die übrigen Zustandsvariablen des *SubsystemSensor*, wie etwa *proxyx*, *proxyxqu*, *proxyxs*, werden durch die Aktion *TickFS* nicht verändert. Die maximalen, persistenten Wartezeiten werden, wie vorher in den Prozessspezifikationen definiert, verwendet. Durch ein Konstrukt der Form:

```
enabled(ProxyDequeue) ⇒ now′ ≤ now + (tProxyDequeue - TimerProxyDequeue)
```

, welches hier exemplarisch an der Aktion *ProxyDequeue* vorgestellt wird, ist sichergestellt, dass die

Zustandsvariable *now* nicht über den Wert der persistenten, maximalen Wartezeit *tProxyDequeue* steigt. Durch ein Konstrukt der Form:

```
TimerProxyProcess' = TimerProgress(ProxyProcess, TimerProxyProcess)
```

wird sichergestellt, dass die Timervariable *TimerProxyDequeue* um now' - now erhöht wird. Dabei wird wieder das **LET-IN** Konstrukt, das in Abschnitt 5.6 vorgestellt worden ist, verwendet. Für die Aktion *TickFS* wird starke Fairness `SF(TickFS)` spezifiziert.

```
SensorProcess(currentEventOccurence : String) ≜ ! Verarbeitung
des Sensors
    ∧ O2.stutter
    ∧ TO2.stutter
    ∧ O3.stutter
    ∧ TO3.stutter
    ∧ O4.process(currentEventOccurence)
    ∧ TO4.process
    ∧ sd.permittedActionOfSD(currentEventOccurence);
ReturnFromSensor(value : Real; currentEventOccurence : String) ≜ !
Antwort des Sensors
    ∧ O2.stutter
    ∧ TO2.stutter
    ∧ O3.enqueue(value)
    ∧ TO3.enqueue
    ∧ O4.getValueCallReply(value, currentEventOccurence)
    ∧ TO4.getValueCallReply
    ∧ sd.permittedActionOfSD(currentEventOccurence);
...
```

Abbildung 11.23: Einige ausgewählte Systemaktionen für den Prozesstyp `Sensor`

```
TickFS(t: Real) ≜
    ∧ O2.stutter
    ∧ TO2.Tick(t)
    ∧ O3.stutter
    ∧ TO3.Tick(t)
    ∧ O4.stutter
    ∧ TO4.Tick(t)
    ∧ sd.stutter;
```

Abbildung 11.24: Die Aktion `TickFS`

```
TickFS(t : Real) ≜
 LET
  TimerProg(action, timerVariable) ≜
      IF(enabled(action))
      THEN timerVariable + now′ - now
      ELSE timerVariable
 IN
 ∧ now′ = t
 ∧ now′ > now
 ∧ now′ ≤ now + ϵ ! Hilfskonstrukt, um Zeno-Verhalten auszuschließen
 ∧ [d ∈ Data] :: enabled(CallToProxy(d)) ⇒ now′ ≤ now + (tProxyEnqueue -
  TimerCallToProxy[d])
 ∧ enabled(ProxyDequeue) ⇒ now′ ≤ now + (tProxyDequeue - TimerProxyDequeue)
 ∧ enabled(ProxyProcess) ⇒ now′ ≤ now + (tProxyProcess - TimerProxyProcess)
 ∧ enabled(setCall3) ⇒ now′ ≤ now + (tsetCall3 - TimerSetCall3)
 ∧ ∀ [d ∈ Data] :: enabled(CallToAdapter(d)) ⇒ now′ ≤ now +
           (tCallToAdapter - TimerCallToAdapter[d])
 ∧ enabled(AdapterDequeue) ⇒ now′ ≤ now + (tAdapterDequeue - TimerAdapterDequeue)
 ∧ enabled(AdapterProcess) ⇒ now′ ≤ now + (tAdapterProcess - TimerAdapterProcess)
 ∧ ∀ [d ∈ Data] :: enabled(CallToSensor(d)) ⇒ now′ ≤ now + (tCallToSensor -
   TimerCallToSensor[d])
 ∧ enabled(SensorDequeue) ⇒ now′ ≤ now + (tSensorDequeue - TimerSensorDequeue)
 ∧ enabled(SensorProcess)) ⇒ now′ ≤ now + (tSensorProcess - TimerSensorProcess)
 ∧ ∀ [d ∈ Data] :: enabled(ReturnSensor(d)) ⇒ now′ ≤ now +
                   (tReturnSensor - TimerReturnSensor[d])
 ∧ ∀ [d ∈ Data] :: enabled(ReturnAdapter(d)) ⇒ now′ ≤ now +
                  (tReturnAdapter - TimerReturnAdapter[d])
 ∧ ∀ [d ∈ Data] :: enabled(ReturnProxy[d]) ⇒ now′  ≤ now +
                   (tReturnProxy - TimerReturnProxy[d})
 ∧ enabled(ProxyError1) ⇒ now′ ≤ now + (tProxyError1 - TimerProxyError1)
 ∧ enabled(ProxyError2) ⇒ now′ ≤ now + (tProxyError2 - TimerProxyError2)
 ∧ enabled(ProxyError3) ⇒ now′ ≤ now + (tProxyError3 - TimerProxyError3)
 ∧ enabled(ProxyError4) ⇒ now′ ≤ now + (tProxyError4 - TimerProxyError4)
 ∧ ∀ [d ∈ Data] :: TimerCallToProxy[d]′ = TimerProg(CallToProxy(d), TimerCallToProxy[d]) ! ft1
 ∧ TimerProxyDequeue′ = TimerProg(ProxyDequeue, TimerProxyDequeue) ! ft2
 ∧ TimerProxyProcess′ = TimerProg(ProxyProcess, TimerProxyProcess) ! ft3
 ∧ TimerSetCall1′ = = TimerProg(setCall1, TimerSetCall1)          ! ft4
 ∧ ∀ [d ∈ Data] :: TimerCallToAdapter[d]′ = TimerProg(CallToAdapter(d),
  TimerCallToAdapter[d]) ! ft5
 ∧ TimerAdapterDequeue′ = TimerProg(AdapterDequeue, TimerAdapterDequeue) ! ft6
 ∧ ∀ [d ∈ Data] :: TimerCallToSensor[d]′ = TimerProg(CallToSensor(d), TimerCallToSensor[d]) ! ft7
 ∧ TimerSensorDequeue′ = TimerProg(SensorDequeue, TimerSensorDequeue) ! ft8
 ∧ TimerSensorProcess′ = TimerProg(SensorProcess, TimerSensorProcess) ! ft9
 ∧ ∀ [d ∈ Data] :: TimerReturnSensor[d]′ = TimerProg(ReturnSensor(d), TimerReturnSensor[d]) ! ft9
 ∧ ∀ [d ∈ Data] :: TimerReturnAdapter[d]′ = TimerProg(ReturnAdapter(d), TimerReturnAdapter) ! ft10
 ∧ ∀ [d ∈ Data] :: TimerReturnProxy[d]′ = TimerProg(ReturnProxy(d), TimerReturnProxy[d]) ! ft11
 ∧ TimerSetCall2′ = TimerProg(setCall2, TimerSetCall2) ! ft12
 ∧ TimerProxyError1′ = TimerProg(ProxyError1,TimerProxyError) ! ft13
 ∧ TimerProxyError2′ = TimerProg(ProxyError2,TimerProxyError) ! ft14
 ∧ TimerProxyError3′ = TimerProg(ProxyError3,TimerProxyError) ! ft15
 ∧ TimerProxyError4′ = TimerProg(ProxyError4,TimerProxyError) ! ft16
 ∧ UNCHANGED ⟨sProxy, Proxyqu, Proxyx, sAdapter, Adapterqu, Adapterx, sSensor, Sensorqu, Sensorx⟩
 ∧ UNCHANGED ⟨x, y, qu, sync⟩;
```

Abbildung 11.25: Die Aktion TickFS des Feinsystems

Kapitel 12

Korrektheitsbeweis eines Verfeinerungsmusters

Im Rahmen dieses Projekts sind einige Verfeinerungsmuster verifiziert und publiziert worden [GHK00, GH04b].

Einer dieser Beweise, die Verifikation eines entfernten, verteilten Reglermusters unter Verwendung der vorgestellten Werkzeuge und Beweistechniken, wird in diesem Kapitel in Auszügen vorgestellt. Dazu wird die korrekte Verfeinerung für das *SubsystemSensor* in Abschnitt 12.1 und das *SubsystemController* in Abschnitt 12.2 bewiesen. Das *SubsystemAktor* wird nicht näher betrachtet.

Anschließend werden in Abschnitt 12.3 Handbeweise durchgeführt, um die Korrektheit der Verfeinerung der persistenten Wartezeiten der Aktionen des Analysemusters durch die Aktionen des Entwurfsmusters zu garantieren. In Abschnitt 12.4 wird der Ausschluss von Zeno-Verhalten für die *Tick* Aktion als Handbeweis gezeigt. Die Beweise sind – soweit möglich – zusätzlich mit dem Model Checker TLC nachgeprüft worden. TLC eignet sich sehr gut, um Invarianten zu finden, bzw. deren Korrektheit nachzuweisen.

12.1 Nachweis der Verfeinerung des SubsystemSensor

In diesem Abschnitt wird auf den Beweis der korrekten Verfeinerung des Verhaltens der Klasse *abstractSensor* durch das Verhalten des *SubsystemSensor* – bestehend aus den Prozessen der Klassen *SensorProxy*, *SensorAdapter* und *Sensor* – eingegangen. Da reale Verfeinerungsbeweise nicht direkt geführt werden können, weil in der Feinsystemspezifikation für eine Verifikation notwendiges Wissen über dauerhafte Systemeigenschaften nur implizit definiert wird, müssen zunächst geeignete Invarianten gefunden werden. Eine Invariante beschreibt dabei die für einen Beweis relevanten Systemeigenschaften. Sie ist speziell bei den Sicherheitseigenschaften für den Beweis einer Implikation $Inv \wedge A_F \Rightarrow A_G$ — also einer Sicherheitseigenschaft — notwendig. Hierbei bezeichnet Inv eine Invariante, A_G eine Grobsystem-Aktion und A_F eine Feinsystem-Aktion. Auch bei den Lebendigkeitsbeweisen sind Invarianten von großer Bedeutung. Da häufig Implikationsbeweise notwendig sind, sollen kurz grundlegende Beweismethoden abgegeben werden. Es gilt $(A \Rightarrow B) \Leftrightarrow (\neg A \vee B)$. Der Beweis von $A \Rightarrow B \wedge C$ wird in die Teilbeweise $A \Rightarrow B$ und $A \Rightarrow C$, der Beweis von $A \vee B \Rightarrow C$ zerfällt in die Beweise $A \Rightarrow C$ und $B \Rightarrow C$. Natürlich sind auch Kombinationen der beiden Fälle möglich. Für die Invarianten muss gezeigt werden, dass das Initialisierungsprädikat *Init* die Invariante *Inv* impliziert also $Init \Rightarrow Inv$. Weiterhin muss unter Anwendung der Beweisregel TLA INV1 die Implikation $Inv \wedge Next \Rightarrow Inv'$ gezeigt werden, wobei *Next* die Disjunktion aller Aktionen bezeichnet. Durch die Beweisregeln von TLA werden viele Beweise, die sich auf Zustände bzw. Zustandspaare beziehen, auf prädikatenlogische Beweise zurückgeführt.

Zum Nachweis der Verfeinerung des Prozesses *abstractSensor* durch das *SubsystemSensor* werden zunächst die Invarianten *Inv0*, *Inv1*, *Inv2* und *Inv3* bewiesen. In diesen Beweisen wird gezeigt, dass die Invarianzeigenschaften von der Initialbedingung des *SubsystemSensor* erfüllt werden. Da-

nach werden die Invarianzeigenschaften durch vollständige Induktion für jede Aktion nachgewiesen. Die Aktionen werden durch die Kopplung der Aktionen der Teilprozesse gebildet, wie sie in den Systemaktionen des vorherigen Kapitels beschrieben sind. Die Zustandsvariablen in den Aktionen erhalten zusätzlich ein Präfix, das dem Prozessnamen entspricht. Es erfolgen auch Anpassungen der Variablennamen, um Namenskonflikte zu vermeiden. So entspricht die Zustandsvariable *proxyxss* der Zustandsvariablen *sProxy* des Prozesses *SensorProxy*.

Anschließend wird die Verfeinerungsabbildung *RM1* zwischen den Zustandsvariablen des *SubsystemSensor*, der Entwurfsmuster-Kombination und den Zustandsvariablen des Prozesses *abstractSensor* vorgestellt und der Verfeinerungsbeweis durchgeführt. Die Invarianten- und Verfeinerungsbeweise in TLA tendieren zu umfangreichen, flachen Beweisstrukturen. Es gibt viele Möglichkeiten Invarianten zu strukturieren. In dieser Arbeit wird die Strategie verfolgt, die Anzahl der Invarianten zu minimieren und jeweils eine, allerdings komplexe Invariante pro cTLA-Prozess anzugeben. Interessant ist die Struktur der Invarianten. Jede Invariante eines Prozesses besteht aus Disjunkten, die auch Zustandsvariablen aus anderen Prozessen enthält. Durch Existenzquantifizierung wird die Anzahl der Disjunkte reduziert, falls ähnliche Disjunkte existieren, die sich nur um den Wert einer Variable unterschieden. Hierzu werden gebundene Variablen in die Invariante aufgenommen. Damit lässt sich eine Invariante signifikant zusammenfassen. Weiterhin werden zusätzliche Operatoren und Funktionen eingeführt, um die Invarianten lesbarer zu gestalten. Zunächst werden in den Abschnitten 12.1.1 bis 12.1.4 die Invarianten *Inv0*, *Inv1*, *Inv2* und *Inv3* für den *SubsystemSensor* bewiesen. Diese werden in Abschnitt 12.1.5 zum Nachweis der korrekten Verfeinerung benötigt.

12.1.1 Beweis der Invariante *Inv0*

Die Invariante *Inv0* aus Abbildung 12.1 behandelt die Wertebelegung der Zustandsvariablen *currentTrace* des Prozesstyps *SequenceDiagram*. Bei dieser Zustandsvariablen wird mit jedem Schalten einer Aktion des *SubsystemSensor* der aktuelle Trace um einen weiteren Ereignisauftritt verlängert. Ein Ereignisauftritt gibt dabei an, welche UML-Action durch ein Sequenzdiagramm gestattet ist. Sie wird durch eine Zeichenkette, bestehend aus dem Objektnamen und dem Namen der UML-Action spezifiziert (s. Abschnitt 10.3.3). Die Invariante *Inv0* nimmt eine Sonderrolle ein, weil sie die Ereignisauftritte aller Objekte behandelt und angibt welche Traces zulässig sind. Die Invariante ist daher sehr wichtig für den Beweis der übrigen Invarianten *Inv1*, *Inv2*, *Inv3*. In der Invariante werden die Sequenzen durch Funktionsaufrufe berechnet. Die Funktion *fProxy*, die in Abbildung 12.2 gezeigt wird, dient beispielsweise der Berechnung des Traces für den Prozess *SensorProxy*. Die Funktion erhält als Parameter die Menge aller möglichen Werte der Zustandsvariable *proxyxss* des Prozesses *Sensor*. Der Rückgabeparameter ist die *Sequence*, die zu dem Trace des entsprechenden Zustandes gehört. Über Fallunterscheidungen wird die zugehörige *Sequence* ermittelt. So wird für das Argument "callDequeued" die *Sequence SubSeq(sonet, 1, 1)* zurückgeliefert. Mit *sonet* (set of non error traces) wird die Menge der zulässigen Traces für normale Verarbeitung notiert. Alternativ wird durch *soet* (set of error traces) die Menge der zulässigen Traces für den Fehlerfall angegeben. Die Operatoren *adapterTraceOr* und *sensorTraceOr* berechnen die Traces für den *SensorAdapter* sowie den *Sensor*.

Durch Existenzquantifizierung (E1) wird die Anzahl der Disjunkte von *Inv0* reduziert, weil eine gebundene Variable Werte aus einer Menge annehmen darf. Deshalb wird die gebundene Variable x eingeführt, die die beiden möglichen Werte der Zustandsvariablen *proxyxss* – nämlich "called" und "calledError" – annehmen darf. Die Disjunkte (D1) bis (D13) von *Inv0* beschreiben, wie die Zustandsvariable *currentTrace* bei der normalen Verarbeitung durch das *SubsystemSensor* aufgebaut wird, d. h. es treten weder Fehler noch Zeitverletzungen auf. Im Gegensatz dazu wird durch die Disjunkte (D14) bis (D17) die Behandlung im Fehlerfall beschrieben. Nun sollen die Disjunkte im Einzelnen erläutert werden. Die Disjunkte (D1) bis (D5) beschreiben die Zustände und Transitionen bis zum *getValue*-Aufruf des Prozesses *SensorProxy*. Das Disjunkt (D6) beschreibt den Zustand aus dem ein *getValue*-Aufruf an den Prozess *SensorAdapter* erfolgen kann. Durch das Disjunkt (D7) wird der Zustand, der nach dem Schalten der Aktionen des Prozesses *SensorAdapter* gilt, beschrieben. Das Disjunkt (D8) spezifiziert den Zustand nach dem *getValue*-Aufruf an den Prozess *Sensor*. Die Disjunkte (D9) und (D10) stellen eine Beziehung zwischen den Zuständen

```
Inv0 ≜
(E1) ∃ x ∈ {"called", "calledError"} :
(D1) (aSxsSensor ="init" ∧ currentTrace = ≪≫ ) ∨
(D2) (proxyxss = "callEnqueued" ∧ currentTrace =  ≪≫ ) ∨
(D3) (proxyxss = "callDequeued" ∧ currentTrace = fProxy["callDequeued"]) ∨
(D4) (proxyxss = "processed" ∧ currentTrace = fProxy["processed"]) ∨
(D5) (proxyxss = "readyForCall" ∧ currentTrace = fProxy["readyForCall"]) ∨
(D6) (proxyxss = x ∧ adapterxsAdapter = "enqueued" ∧ adapTraceOr(x, "enqueued") ) ∨
(D7) (proxyxss = x ∧ adapterxsAdapter = "dequeued" ∧ adapTraceOr(x,"dequeued")) ∨
(D8) (proxyxss = x ∧ aSxsSensor = "enqueued" ∧ sensorTraceOr(x,"enqueued")) ∨
(D9) (proxyxss = x ∧ aSxsSensor = "dequeued" ∧ sensorTraceOr(x,"dequeued")) ∨
(D10) (proxyxss = x ∧ aSxsSensor = "processed" ∧ sensorTraceOr(x,"processed")) ∨
(D11) (proxyxss = x ∧ adapterxsAdapter = "returnEnqueued"
∧ adapTraceOr(x,"returnEnqueued")) ∨
(D12) (proxyxss = "returnEnqueued" ∧ currentTrace = fProxy["returnEnqueued"]) ∨
(D13) (proxyxss = "returnDequeued" ∧ currentTrace = fProxy["returnDequeued"]) ∨
(D14) (proxyxss = "readyForError" ∧ currentTrace = fProxy["readyForError"]) ∨
(D15) (proxyxss = "returnEnqError" ∧ currentTrace = fProxy["returnEnqError"]) ∨
(D16) (proxyxss = "returnDeqError" ∧ currentTrace = fProxy["returnDeqError"]) ∨
(D17) (proxyxss = "Error" ∧ currentTrace = fProxy["Error"])
```

Abbildung 12.1: Die Invariante `Inv0`

und Transitionen her, die durch separate *Sensor*-Aktionen zustande kommen. Durch die Disjunkte (D11) und (D12) wird die Beziehung zwischen den Zuständen und Transitionen für die Rückgabe des Istwertes durch den *Sensor* und den Prozess *SensorAdapter* spezifiziert. Die Disjunkte (D13) bis (D17) wiederum setzen Zustände und Transitionen in Bezug, die durch Aktionen des Prozesses *SensorProxy* betreten werden. Speziell werden durch die Disjunkte (D14) bis (D17) Zustände und Transitionen für die Fehlerbehandlung in Beziehung gesetzt. Das Disjunkt (D17) gibt für die Zustandsvariable *currentTrace* die möglichen Sequenzen im Fehlerfall an. Für Zeitüberschreitungen werden entsprechende Disjunkte aufgenommen. Der Beweis der Invariante *Inv0* wird hier nur angedeutet. Im folgenden Abschnitt wird ein detaillierterer Beweis gezeigt.

```
fProxy ≜ [ p ∈ {"callEnqueued", "callDequeued", "processed", "readyForCall",
                "readyForError", "called, "calledError", "returnEnqueued",
                "returnDequeued", "returnEnqError", "returnDeqError", "Error"}
                                ↦
                                IF p = "callDequeued"
                                THEN SubSeq(sonet, 1, 1) !
                                ELSE IF p = "processed"
                                THEN SubSeq(sonet, 1, 2) !
                                ELSE IF p = "readyForCall"
                                THEN SubSeq(sonet, 1, 3)
                                ELSE IF p = "called"
                                THEN SubSeq(sonet, 1, 4)
                                ELSE IF p = "readyForError"
                                THEN SubSeq(soet, 1, 3)
                                ...
                                ELSE IF p = "returnEnqueued"
                                THEN SubSeq(sonet, 1, 10)
                                ELSE IF p = "returnDequeued"
                                THEN SubSeq(sonet, 1, 11)
                                ELSE ≪ ≫
```

Abbildung 12.2: Die Funktion `fProxy`

12.1.2 Invariante *Inv1* und deren Beweis

Die Invariante *Inv1* aus Abbildung 12.3 ist ein Zustandsprädikat, das alle Zustände des Prozesses *SensorProxy* beschreibt. Die einzelnen Disjunkte der Invariante *Inv1* werden mit (D1) bis (D14) bezeichnet. Mit den Disjunkten (D1) und (D2) wird das Verhalten bei einem Aufruf aus dem *SensorProxy*-Prozess an den *SensorAdapter*-Prozess beschrieben, indem entsprechende Zustände und Transitionen in Beziehung gesetzt werden. Die Disjunkte (D3) bis (D9) sowie (D14) beschreiben die normale Verarbeitung durch die Aktionen des Prozesses *SensorProxy*, die Disjunkte (D10) bis (D13) das Verhalten im Fehlerfall. Die Disjunkte (D10) und (D11) beschreiben das Einfügen und Entfernen von Aufrufen im Fehlerfall. Abschließend werden durch die Disjunkte (D12) und (D13) Fehlerzustände beschrieben. Das Verhalten im Fall einer Zeitüberschreitung wird in der Invariante *Inv1* nicht berücksichtigt.

Im Folgenden wird die Invarianzeigenschaft $\Box Inv1$ zunächst für das Initialisierungsprädikat und dann für die einzelnen Aktionen des *SubsystemSensor* nachgewiesen. In der Invariante findet der Operator *oneCallInQueue* mit dem Argument *qu* Verwendung, der in Abbildung 12.4 gezeigt ist. Dieser Operator prüft, ob genau ein Aufruf in der Warteschlange vorhanden ist.

```
Inv1 ≜
(D1-D2)  proxyxss = "called" ∨ proxyxss = "calledError" ∨
(D3) ((proxyxss = "init" ∧ proxyxx = 0 ∧ proxyxqu = ≪ ≫ ∧
 adapterxsAdapter = "init" ∧ adapterxqu = ≪ ≫ ∧ adapterxx = 0 ∧
 aSxsSensor = "init" ∧ aSxqu = ≪ ≫  ∧ aSxx = 0) ∨
(D4)  (proxyxss = "callEnqueued" ∧ proxyxx = 0 ∧ oneCallInQueue(proxyxqu) ∧
 adapterxsAdapter = "init" ∧ adapterxqu = ≪ ≫ ∧ adapterxx = 0 ∧
 aSxsSensor = "init" ∧ aSxqu = ≪ ≫ ∧ aSxx = 0) ∨
(D5)  (proxyxss = "callDequeued" ∧ proxyxx ∈ Data ∧ proxyxqu = ≪ ≫ ∧
 adapterxsAdapter = "init" ∧ adapterxqu = ≪ ≫  ∧ adapterxx = 0 ∧
 aSxsSensor = "init" ∧ aSxqu = ≪ ≫ ∧ aSxx = 0) ∨
(D6)  (proxyxss = "processed" ∧ proxyxx ∈ Data ∧ proxyxqu = ≪ ≫ ∧
 adapterxsAdapter = "init" ∧ adapterxx = 0 ∧ adapterxqu = ≪ ≫ ∧
 aSxsSensor = "init" ∧ aSxqu = ≪ ≫ ∧ aSxx = 0) ∨
(D7)  (proxyxss = "readyForCall" ∧ proxyxx ∈ Data ∧ proxyxqu = ≪ ≫ ∧
 adapterxsAdapter = "init" ∧ adapterxx = 0 ∧ adapterxqu = ≪ ≫ ∧
 aSxsSensor = "init" ∧ aSxqu = ≪ ≫ ∧ adapterxx = 0 ∧ aSxx = 0) ∨
(D8)  (proxyxss = "readyForError" ∧ proxyxx ∈ Data ∧ proxyxqu = ≪ ≫ ∧
 adapterxsAdapter = "init" ∧ aSxsSensor = "init" ∧ aSxqu = ≪ ≫ ∧ aSxx = 0) ∨
(D9)  (proxyxss = "returnEnqueued" ∧ proxyxx ∈ Data ∧ oneCallInQueue(proxyxqu) ∧
 adapterxsAdapter = "init" ∧ adapterxqu = ≪ ≫ ∧ adapterxx = 0 ∧
 aSxsSensor = "init" ∧ aSxqu = ≪ ≫ ∧ aSxx = 0) ∨
(D10) (proxyxss = "returnDeqError" ∧ proxyxx ∈ Data ∧ oneCallInQueue(proxyxqu) ∧
 adapterxsAdapter = "init" ∧ adapterxqu = ≪ ≫ ∧ adapterxx = 0 ∧
 aSxsSensor = "init" ∧ aSxqu = ≪ ≫ ∧ aSxx = 0) ∨
(D11) (proxyxss = "returnEnqError" ∧ proxyxx ∈ Data ∧ oneCallInQueue(proxyxqu) ∧
 adapterxsAdapter = "init" ∧ adapterxqu = ≪ ≫ ∧ adapterxx = 0 ∧
 aSxsSensor = "init" ∧ aSxqu = ≪ ≫ ∧ aSxx = 0) ∨
(D12) (proxyxss = "Error" ∧ proxyxx ∈ Data ∧ (proxyxqu = ≪ ≫ ∨
 proxyxqu = ≪"Error"≫) ∧ adapterxsAdapter = "init" ∧ adapterxqu = ≪ ≫ ∧
 adapterxx = 0 ∧ aSxsSensor = "init" ∧ aSxqu = ≪ ≫ ∧ aSxx=0) ∨
(D13) (proxyxss = "Error" ∧ proxyxx = 0 ∧ (proxyxqu =
≪ ≫ ∨ proxyxqu = ≪"Error"≫ ∨ oneCallInQueue(proxyxqu)) ∧
 adapterxsAdapter = "init" ∧ adapterxqu = ≪ ≫ ∧ adapterxx = 0 ∧
 aSxsSensor = "init" ∧ aSxqu = ≪ ≫ ∧ aSxx=0) ∨
(D14) (proxyxss = "returnDequeued" ∧ proxyxx ∈ Data ∧ proxyxqu = ≪ ≫ ∧
 adapterxsAdapter = "init" ∧ adapterxqu = ≪ ≫ ∧ adapterxx = 0 ∧
 aSxsSensor = "init" ∧ aSxqu = ≪ ≫ ∧ aSxx = 0)))
```

Abbildung 12.3: Die Invariante `Inv1`

```
oneCallInQueue(qu) ≜ Head(qu) ∈ Data ∧ Tail(qu) = ≪ ≫
```

Abbildung 12.4: Der Operator `oneCallInQueue`

Das Initialisierungsprädikat *Init* des gesamten *SubsystemSensor* muss *Inv1* implizieren, also (12.1) gelten.

$$Init \Rightarrow Inv1(D3) \tag{12.1}$$

Dies gilt wegen des Disjunktes (D3) in der folgenden Implikation:

```
∧ proxyxx = 0                                  ∧ proxyxx = 0
∧ proxyxqu = ≪ ≫                               ∧ proxyxqu = ≪ ≫
∧ proxyxss = "init"                            ∧ proxyxss = "init"
∧ adapterxx = 0                                ∧ adapterxx = 0
∧ adapterxqu = ≪  ≫          ⇒                 ∧ adapterxqu = ≪ ≫
∧ adapterxsAdapter = "init"                    ∧ adapterxsAdapter = "init"
∧ aSxqu = ≪ ≫                                  ∧ aSxqu = ≪ ≫
∧ aSxx = 0                                     ∧ aSxx = 0
∧ aSxsSensor = "init"                          ∧ aSxsSensor = "init"
∧ currentTrace = ≪ ≫
∧ hqu = ≪ ≫; ! Hilfsvariable für Refinement-Mapping
```

Weiterhin muss für alle Aktionen A_i von *SubsystemSensor* $A_i \wedge Inv1 \Rightarrow Inv1'$ nachgewiesen werden. Bei der Aktion *valueToProxy* aus Abbildung 12.5 wird dies gezeigt. Hier ist für alle $(D_i)\ i \in \{1, 2, \ldots, 14\} \setminus \{3\}$ die linke Seite der Implikation falsch, weil diese Disjunkte alle im Widerspruch zur Schaltbedingung $enabled(valueToProxy(d))$ stehen. Damit gilt trivialerweise $\exists d \in ValueToProxy(d) \wedge Inv1(D_i) \Rightarrow \exists\ d \in Data\ Inv1'(D_i)$.

Das Disjunkt (D3) der Invariante *Inv1*führt zum Schalten der Aktion (s. (12.2)), da die Vorbedingung erfüllt wird. Nach dem Schalten der Aktion gilt das Disjunkt (D4) von *Inv1*, da die Aktion die Effekte $proxyxss' =''\ callEnqueued''$ und $proxyxqu' = Append(proxyxqu, d)$ besitzt. Da das Disjunkt (D4) von *Inv1* gültig ist, gilt auch *Inv1*'(D4).

$$\exists d \in Data\ :\ ValueToProxy(d)(K1) \wedge Inv1(D3) \Rightarrow \exists d \in Data\ :\ Inv1'(D4) \tag{12.2}$$

Die TLA Regel INV2 findet Verwendung, um eine weitere Invariante zum Beweis – hier *Inv0* – heranzuziehen. Die Aktion *ProxyDequeue* aus Abbildung 12.6 ist bei den drei Disjunkten (D4), (D9) und (D11) von *Inv1* schaltbereit. Alle anderen Disjunkte $(D_i)\ i \in \{1, 2, \ldots, 14\} \setminus \{4, 9, 11\}$ erfüllen die Schaltbedingung nicht. Aus der Invariante *Inv1* erkennt man, sobald die Aktion durch das Disjunkt (D4) von *Inv1* schaltbereit wird, ist nach dem Schalten von *ProxyDequeue* (D5) von *Inv1* erfüllt. Wenn durch (D9) von *Inv1* die Aktion schaltbereit wird, ist nach dem Schalten (D14) von *Inv1* erfüllt. Für den Fall, dass (D11) von *Inv1* vor dem Schalten erfüllt ist, gilt nach dem Schalten (D10). Daher gelten die Implikationen (12.3), (12.4) und (12.5) unter der Annahme, dass (D2), (D12) und (D15) von *Inv0* gültig sind.

$$ProxyDequeue(K1, K2, K3) \wedge Inv0(D2) \wedge Inv1(D4) \Rightarrow Inv1'(D5) \tag{12.3}$$

$$ProxyDequeue(K1, K2, K3) \wedge Inv0(D12) \wedge Inv1(D9) \Rightarrow Inv1'(D14) \tag{12.4}$$

$$ProxyDequeue(K1, K2, K3) \wedge Inv0(D15) \wedge Inv1(D11) \Rightarrow Inv1'(D10) \tag{12.5}$$

Die Aktion *ProxyProcess* aus Abbildung 12.7 wird mittels des Disjunktes (D5) der Invariante *Inv1* schaltbereit.

Nach dem Schalten der Aktion gilt:

```
ValueToProxy(d : Data) ≜
          ∧ proxyxss = "init"
          ∧ proxyxss′ = "callEnqueued"
          ∧ proxyxqu′= Append(proxyxqu, d)
          ∧ hqu′ = Append(proxyxqu, d)
          ∧ UNCHANGED proxyxx
          ∧ UNCHANGED ⟨adapterxsAdapter, adapterxqu, adapterxx⟩
          ∧ UNCHANGED ⟨aSxsSensor, aSxqu, aSxx⟩
          ∧ UNCHANGED currentTrace;
```

Abbildung 12.5: Die Aktion `ValueToProxy`

$$ProxyProcess(K1, K2, K3) \ \wedge \ Inv0(D3) \ \wedge \ Inv1(D5) \ \Rightarrow \ Inv1'(D6) \tag{12.6}$$

Bei den übrigen Disjunkten von *Inv1* ist – mit Ausnahme von (D5) – die linke Seite der Implikation (12.6) falsch, weshalb die Implikation gültig ist.

Die Aktion *setCall1*, die in Abbildung 12.8 gezeigt ist, wird durch das Disjunkt (D6) von *Inv1* und das Disjunkt (D4) von *Inv0* schaltbereit. Durch die linke Seite der Implikation gilt – dann unter Anwendung der TLA Regel INV2 – nach dem Schalten $Inv1'(D7)$ also:

$$setCall1(K1, K2, K3) \ \wedge \ Inv0(D4) \ \wedge \ Inv1(D6) \ \Rightarrow \ Inv1'(D7) \tag{12.7}$$

Für alle übrigen Disjunkte mit Ausnahme von (D6) von *Inv1* ist die linke Seite der Implikation (12.7) falsch.

Das Disjunkt (D6) von *Inv1* bewirkt, dass die Aktion *setCall2* gemäß Abbildung 12.9 schaltbereit wird. Im Anschluss an das Schalten von *setCall2* gilt (D8). Die Aktion *setCall2* wird durch das Disjunkt (D6) von *Inv1* und das Disjunkt (D4) von *Inv0* schaltbereit. Nach dem Schalten der Aktion gilt dann $Inv1'(D8)$, also:

$$setCall2(K1, K2, K3) \ \wedge \ Inv0(D4) \ \wedge \ Inv1(D6) \ \Rightarrow \ Inv1'(D8) \tag{12.8}$$

Bei den übrigen Disjunkten (D_i) $i \ \in \{1 \ldots 14\} \setminus \ \{6\}$ ist die linke Seite der Implikation (12.8) nicht erfüllt.

```
ProxyDequeue(cE0 : String) ≜
              ∧ cE0 = "02.actionDequeue"
              ∧ Append(currentTrace, cE0) ∈ setOfPossibleTraces
              ∧ (proxyxss = "callEnqueued" ∨ proxyxss = "returnEnqueued"
                      ∨ proxyxss = "returnEnqError" ∨ proxyxss = "returnEnqTimeout")
              ∧  proxyxss′ = IF(proxyxss = "callEnqueued")
                             THEN "callDequeued"
                             ELSE IF proxyxss = "returnEnqueued"
                             THEN "returnDequeued"
                             ELSE IF proxyxss = "returnEnqError"
                             THEN "returnDeqError"
                             ELSE "returnDeqTimeout"
              ∧ proxyxx′ = Head(proxyxqu)
              ∧ proxyxqu′ = Tail(proxyxqu)
              ∧ currentTrace′ = Append(currentTrace, cE0)
              ∧ UNCHANGED ⟨adapterxsAdapter, adapterxqu, adapterxx⟩
              ∧ UNCHANFGED ⟨aSxsSensor, aSxqu, aSxx⟩
              ∧ UNCHANGED hqu;
```

Abbildung 12.6: Die Aktion `ProxyDequeue`

```
ProxyProcess(cEO : String) ≜
          ∧ cEO = "O2.actionProcess"
          ∧ Append(currentTrace, cEO) ∈ setOfPossibleTraces
          ∧ proxyxss = "callDequeued"
          ∧ proxyxss′ = "processed"
          ∧ currentTrace′ = Append(currentTrace, cEO)
          ∧ UNCHANGED ⟨proxyxqu, proxyxx, proxyxss⟩
          ∧ UNCHANGED ⟨adapterxsAdapter, adapterxqu, adapterxx⟩
          ∧ UNCHANGED ⟨aSxsSensor, aSxqu, aSxx⟩
          ∧ UNCHANGED hqu;
```

Abbildung 12.7: Die Aktion `ProxyProcess`

```
setCall1(cEO : String) ≜
          ∧  cEO = "O2.actionSetCall1"
          ∧  Append(currentTrace, cEO) ∈ setOfPossibleTraces
          ∧  proxyxss = "processed"
          ∧  proxyxss′ = "readyForCall"
          ∧  currentTrace′ = Append(currentTrace, cEO)
          ∧  UNCHANGED ⟨proxyxqu, proxyxx⟩
          ∧  UNCHANGED ⟨adapterxsAdapter, adapterxqu, adapterxx⟩
          ∧  UNCHANGED ⟨aSxsSensor, aSxqu, aSxx⟩
          ∧  UNCHANGED hqu;
```

Abbildung 12.8: Die Aktion `setCall1`

Durch das Disjunkt (D8) von *Inv1* wird die Aktion *ProxyError1* aus der Abbildung 12.10 schaltbereit. Nach dem Schalten gilt $Inv1'(D12)$.

$$ProxyError1(K1, K2, K3) \ \wedge \ Inv0(D14) \ \wedge \ Inv1(D8) \ \Rightarrow \ Inv1'(D12) \qquad (12.9)$$

Für alle übrigen Disjunkte (D_i) $i \in \{1, \ldots, 14\} \setminus \{8\}$ ist die linke Seite der Implikation (12.9) nicht erfüllt.

Die in Abbildung 12.11 gezeigte Aktion *ProxyError3* wird schaltbereit, wenn $Inv1(D11)$ gilt. Nach dem Schalten der Aktion gilt das Disjunkt (D12) oder (D13) von *Inv1*.

$$ProxyError3(K1, K2, K3) \ \wedge \ Inv0(D15) \ \wedge \ Inv1(D11) \ \Rightarrow \ Inv1'(D12 \ oder \ D13) \qquad (12.10)$$

Für alle übrigen Disjunkte (D_i) $i \in \{1, \ldots, 14\} \setminus \{11\}$ ist die linke Seite der Implikation falsch.

Bei der in Abbildung 12.12 gezeigten Aktion *ProxyError4* führt das Disjunkt (D11) von *Inv1* zur Schaltbereitschaft. Nach dem Schalten der Aktion gilt das Disjunkt (D12) oder (D13) von *Inv1*.

```
setCall2(cEO : String) ≜
          ∧ cEO = "O2.actionSetCall2"
          ∧ Append(currentTrace, cEO) ∈ setOfPossibleTraces
          ∧ proxyxss = "processed"
          ∧ proxyxss′ = "readyForError"
          ∧ currentTrace′ = Append(currentTrace, cEO)
          ∧ UNCHANGED ⟨proxyxqu, proxyxx⟩
          ∧ UNCHANGED ⟨adapterxsAdapter, adapterxqu, adapterxx⟩
          ∧ UNCHANGED ⟨aSxsSensor, aSxqu, aSxx⟩
          ∧ UNCHANGED hqu;
```

Abbildung 12.9: Die Aktion `setCall2`

```
ProxyError1(cE0 : String) ≜
          ∧ cE0 = "02.actionError1"
          ∧ Append(currentTrace, cE0) ∈ setOfPossibleTraces
          ∧ proxyxss = "readyForError"
          ∧ aSxqu′ = ≪ ≫
          ∧ aSxsSensor′ = "init"
          ∧ aSxx′ = 0
          ∧ adapterxsAdapter′ = "init"
          ∧ adapterxqu′ = ≪ ≫
          ∧ adapterxx′ = 0
          ∧ proxyxss′ = "Error"
          ∧ UNCHANGED ⟨proxyxqu, proxyxx⟩
          ∧ UNCHANGED hqu;
```

Abbildung 12.10: Die Aktion `ProxyError1`

```
ProxyError3(cE0 : String) ≜
              ∧ cE0 = "02.actionError3"
              ∧ Append(currentTrace, cE0) ∈ setOfPossibleTraces
              ∧ proxyxss = "returnEnqError"
              ∧ proxyxss′ = "Error"
              ∧ currentTrace′ = Append(currentTrace, cE0)
              ∧ UNCHANGED ⟨proxyxqu, proxyxx⟩
              ∧ UNCHANGED ⟨adapterxsAdapter, adapterxqu, adapterxx⟩
              ∧ UNCHANGED ⟨aSxsSensor, aSxqu, aSxx⟩
              ∧ UNCHANGED hqu;
```

Abbildung 12.11: Die Aktion `ProxyError3`

$$ProxyError4(K1, K2, K3) \ \wedge \ Inv0(D16) \ \wedge \ Inv1(D10) \ \Rightarrow \ Inv1'(D12 \ oder \ D13) \qquad (12.11)$$

Für alle übrigen Disjunkte (D_i) $i \in \{1, \ldots, 14\} \setminus \{10\}$ von *Inv1* ist die linke Seite der Implikation (12.11) falsch.

Die Argumentation bzgl. der Korrektheit der Invariante *Inv1* wird für die Aktionen *ProxyTimeout1*, *ProxyTimeout2*, *ProxyTimeout3* und *ProxyTimeout4* wegen der Symmetrie von den *Error*-Aktionen zu den *Timeout*-Aktionen nicht wiederholt.

Sämtliche *SensorAdapter*-Aktionen Akt_i mit Ausnahme der Aktion *enqueue* des Prozesses *SensorAdapter* bewirken, dass $Inv1 \wedge Akt_i \Rightarrow Inv1'$ gilt. Wegen $Inv1 \wedge Akt_i = False$ wird keine der Aktionen durch *Inv1* schaltbereit.

Auch alle Aktionen Akt_i des Prozesses *Sensor* bewirken, dass $Inv1 \wedge Akt_i \Rightarrow Inv'$ gilt, da $Inv1 \wedge Akt_i = False$ und keine der Aktionen durch *Inv1* schaltbereit werden kann.

12.1.3 Beweis der Invariante *Inv2*

In diesem Abschnitt wird die Invarianzeigenschaft $\Box Inv2$ der Invariante *Inv2* aus Abbildung 12.13 bewiesen. Hierfür sind die Werte unmittelbar vor bzw. nach dem Schalten einer Aktion des Prozesses *SensorAdapter* von besonderem Interesse. Mit dem Existenzquantor in (E1) wird die Variable y, die die Werte "ReadyForCall" und "ReadyForError" annehmen darf, in dem folgenden Disjunkt (D6) gebunden. Durch den Existenzquantor (E2) wird die Variable x in den Disjunkten (D7) bis (D10) gebunden. Sie darf die Werte "calledError" und "Error" annehmen. Durch das Disjunkt (D6) wird der Zustand vor dem Schalten der Aktion *ValueToAdapter* angegeben. Mit den Disjunkten (D7) bis (D10) werden die Zustandsänderungen durch die Aktionen des Prozesses *SensorAdapter* verursacht, wenn für die Zustandsvariable des *SensorProxy*-Prozesses $proxyxss =''called''$ bzw.

```
ProxyError4(cE0 : String) ≜
              ∧ cE0 = "02.actionError4"
              ∧ Append(currentTrace, cE0) ∈ setOfPossibleTraces
              ∧ proxyxss = "returnDeqError"
              ∧ proxyxss′ = "Error"
              ∧ currentTrace′ = Append(currentTrace, cE0)
              ∧ UNCHANGED ⟨adapterxsAdapter, adapterxqu, adapterxx⟩
              ∧ UNCHANGED ⟨aSxsSensor, aSxqu, aSxx⟩
              ∧ UNCHANGED hqu;
```

Abbildung 12.12: Die Aktion `ProxyError4`

$proxyxss =''$ *calledError*$''$ gilt. Weiterhin wird mit dem Disjunkt (D8) der Zustand vor dem Schalten der Aktion *AdapterDequeue* bei dem der Aufruf in die Warteschlange des Prozesses *SensorAdapter* aufgenommen worden ist, beschrieben. Durch das Disjunkt (D9) wird der *getValue*-Aufruf an den Prozess *Sensor* spezifiziert, der durch die Aktion *ValueToSensor* vorgenommen wird. Das Disjunkt (D10) behandelt die Zustandsänderung, die durch die Aktion *ReturnFromAdapter* des Prozesses *SensorAdapter* verursacht wird.

Es muss *Inv2* für das Initialisierungsprädikat *Init* erfüllt sein, was wegen (D5) von *Inv2* in (12.12) gilt.

$$Init \Rightarrow Inv2(D5) \tag{12.12}$$

```
Inv2 ≜
(D1-D2)  ∨ (proxyxss = "ReadyForError" ∨ proxyxss = "returnEnqError"
(D3-D5)  ∨ proxyxss = "returnEnqueud" ∨  proxyxss = "Error" ∨ aSxsSensor = "init")
(E1)     ∨ ∃ y ∈ {"ReadyForCall", "ReadyForError"} :
(E2)     ∨ ∃ x ∈ {"called", "calledError"} :
(D6)     ∨ (proxyxss = y ∧ proxyxx = 0 ∧ proxyxqu = ≪ ≫ ∧
           adapterxsAdapter = "init" ∧ adapterxx = 0 ∧ adapterxqu = ≪ ≫ ∧
           aSxsSensor = "init" ∧ aSxx = 0 ∧ aSxqu = ≪ ≫)
(D7)     ∨ (proxyxss = x ∧ proxyxx = 0 ∧ proxyxqu = ≪ ≫ ∧
           adapterxsAdapter = "enqueued" ∧ adapterxx = 0 ∧ oneCallInQueue(adapterxqu) ∧
           aSxsSensor = "init" ∧ aSxx = 0 ∧ aSxqu = ≪ ≫)
(D8)     ∨ (proxyxss = x ∧ proxyxx = 0 ∧ proxyxqu = ≪ ≫ ∧
           adapterxsAdapter = "dequeued" ∧ adapterxx ∈ Data ∧ adapterxqu = ≪ ≫ ∧
           aSxsSensor = "init" ∧ aSxx = 0 ∧ aSxqu = ≪ ≫)
(D9)     ∨ (proxyxss = x ∧ proxyxx = 0 ∧ proxyxqu = ≪ ≫ ∧
           adapterxsAdapter = "called" ∧ adapterxx ∈ Data ∧ adapterxqu = ≪ ≫
(D10)    ∨ (proxyxss = x ∧ proxyxx = 0 ∧ proxyxqu = ≪ ≫ ∧
           adapterxsAdapter = "returnEnqueued" ∧ adapterxx ∈ Data ∧
           oneCallInQueue(adapterxqu) ∧ aSxsSensor = "init" ∧ aSxx = 0 ∧ aSxqu = ≪ ≫))
```

Abbildung 12.13: Die Invariante `Inv2`

Die Aktion *ProxyError2* aus Abbildung 12.14, wird mittels der Disjunkte (D7), D(8), (D9) und (D10) der Invariante *Inv2* schaltbereit, falls $x =''$ *calledError*$''$ gilt. In allen Fällen gilt nach dem Schalten dieser Aktion (D4) von *Inv2*.

$$ProxyError2(K1, K2, K3) \ \wedge \ Inv0(D6) \ \wedge \ Inv2(D7) \ \Rightarrow \ Inv2(D4) \tag{12.13}$$

$$ProxyError2(K1, K2, K3) \ \wedge \ Inv0(D7) \ \wedge \ Inv2(D8) \ \Rightarrow \ Inv2(D4) \tag{12.14}$$

$$ProxyError2(K1, K2, K3) \ \wedge \ Inv0(D8) \ \wedge \ Inv2(D9) \ \Rightarrow \ Inv2(D4) \tag{12.15}$$

```
ProxyError2 ≜
          ∧ cE0 = "02.actionError2"
          ∧ Append(currentTrace, cE0) ∈ setOfPossibleTraces
          ∧ proxyxss = "calledError"
          ∧ currentTrace′ = Append(currentTrace, cE0)
          ∧ aSxqu′ = ≪ ≫
          ∧ aSxsSensor′ = "init"
          ∧ aSxx′ = 0
          ∧ adapterxsAdapter′ = "init"
          ∧ adapterxqu′ = ≪ ≫
          ∧ adapterxx′ = 0
          ∧ proxyxqu′ = Append(proxyxqu, "Error")
          ∧ proxyxss′ = "Error"
          ∧ UNCHANGED proxyxx
          ∧ UNCHANGED hqu;
```

Abbildung 12.14: Die Aktion `ProxyError2`

```
ValueToAdapter(d : Data, cE0 : String) ≜
                        ∧ cE0 = "02.actionCall"
                        ∧ Append(currentTrace, cE0) ∈ setOfPossibleTraces
                        ∧ (proxyxss = "readyForCall" ∨
                                proxyxss = "readyForError" ∨
                                proxyxss = "readyForTimeout")
                        ∧ adapterxsAdapter = "init"
                        ∧ adapterxsAdapter′ = "enqueued"
                        ∧ adapterxqu′ = Append(adapterxqu, d)
                        ∧ proxyxss′ = IF proxyxss = "readyForTimeout"
                                      THEN "calledTimeout"
                                      ELSE IF proxyxss = "readyForError"
                                      THEN "calledError"
                                      ELSE "called"
                        ∧ currentTrace′ = Append(currentTrace, cE0)
                        ∧ UNCHANGED ⟨proxyxqu, proxyxx⟩
                        ∧ UNCHANGED adapterxx
                        ∧ UNCHANGED ⟨aSxsSensor, aSxqu, aSxx⟩
                        ∧ UNCHANGED hqu;
```

Abbildung 12.15: Die Aktion `ValueToAdapter`

$$ProxyError2(K1, K2, K3) \ \wedge \ Inv0(D11) \ \wedge \ Inv2(D10) \ \Rightarrow \ Inv2(D4) \tag{12.16}$$

Bei den übrigen Disjunkten (D_i) $i \ \in \{1, \ldots, 10\} \setminus \{7, 8, 9, 10\}$ von *Inv2* ist die linke Seite der Implikationen (12.13), (12.14), (12.15), (12.16) falsch.

Die in der Abbildung 12.15 angegebene Aktion *ValueToAdapter* wird durch das Disjunkt (D6) von *Inv2* für alle Belegungen der Variable x schaltbereit.

$$\exists d \in Data : ValueToAdapter(d)(K1, K2, K3, K4) \ \wedge \ Inv0(D5, D14) \ \wedge \ Inv2(D6) \ \Rightarrow \ Inv2'(D7) \tag{12.17}$$

Die Aktion *AdapterDequeue* (s. Abbildung 12.16) ist bei der Invariante *Inv2* für das Disjunkt (D7) schaltbereit. Nach dem Schalten der Aktion *AdapterDequeue* gilt $Inv2'(D8)$.

$$AdapterDequeue(K1, K2, K3) \ \wedge \ Inv0(D6) \ \wedge \ Inv2(D7) \ \Rightarrow \ Inv2'(D8) \tag{12.18}$$

```
AdapterDequeue(cEO : String) ≜
          ∧ cEO = "03.actionDequeue"
          ∧ Append(currentTrace, cEO) ∈ setOfPossibleTraces
          ∧ adapterxsAdapter = "enqueued"
          ∧ adapterxsAdapter′ = "dequeued"
          ∧ adapterxx′ = Head( adapterxqu )
          ∧ adapterxqu′ = Tail( adapterxqu )
          ∧ currentTrace′ = Append(currentTrace, cEO)
          ∧ UNCHANGED ⟨proxyxss, proxyxqu,proxyxx⟩
          ∧ UNCHANGED ⟨aSxsSensor, aSxqu, aSxx⟩
          ∧ UNCHANGED hqu;
```

Abbildung 12.16: Die Aktion `AdapterDequeue`

Bei der in Abbildung 12.17 gezeigten Aktion *ValueToSensor* verursacht das Disjunkt (D8), das Schalten der Aktion. Falls (D8) vor dem Schalten gilt, gilt nach dem Schalten (D9) von *Inv2*.

$$\exists d \in Data : ValueToSensor(d)(K1, K2, K3, K4) \ \wedge \ Inv0(D8) \ \wedge \ Inv2(D8) \Rightarrow \exists d \in Data : Inv2'(D9) \tag{12.19}$$

Für alle übrigen Disjunkte (D_i) $i \in \{1, \ldots, 10\} \setminus \{8\}$ ist die linke Seite der Implikation falsch.

Die in der Abbildung 12.18 angegebene Aktion *ReturnFromAdapter* wird durch das Disjunkt (D10) von *Inv2* schaltbereit. Falls das Disjunkt (D10) von *Inv2* das Schalten ausgelöst hat, gilt nach dem Schalten das Disjunkt (D2) oder das Disjunkt (D3).

$$\exists\, d \in Data : ReturnFromAdapter(d)(K1, K2, K3, K4) \ \wedge \ Inv0(D11) \ \wedge \ Inv2(D10) \Rightarrow \exists\, d \in Data : Inv2'(D2 \ oder \ D3) \tag{12.20}$$

Für alle übrigen Disjunkte (D_i) $i \in \{1, \ldots, 10\} \setminus \{10\}$ ist die linke Seite der Implikation (12.20) falsch.

Sämtliche *SensorProxy*-Aktionen Akt_i – mit Ausnahme der Aktionen *ValueToAdapter* und *ProxyError2* – bewirken, dass $Inv2 \wedge \text{Akt}_i \Rightarrow Inv1'$, da keine der Aktionen durch *Inv2* schalten darf, weil die linke Seite der Implikation falsch ist.

Die Aktion *ValueToSensor*, die mit dem Prozess *Sensor* in Verbindung steht, wird als einzige durch die Invariante *Inv2* schaltbereit.

```
ValueToSensor(d : Data, cEO : String) ≜
                    ∧ cEO = "03.actionCall"
                    ∧ Append(currentTrace, cEO) ∈ setOfPossibleTraces
                    ∧ adapterxsAdapter = "dequeued"
                    ∧ aSxsSensor = "init"
                    ∧ aSxsSensor′ = "enqueued"
                    ∧ aSxqu′ = Append( aSxqu,d )
                    ∧ adapterxsAdapter′ = "called"
                    ∧ currentTrace′ = Append(currentTrace, cEO)
                    ∧ UNCHANGED ⟨proxyxss, proxyxqu, proxyxx⟩
                    ∧ UNCHANGED ⟨adapterxqu, adapterxx⟩
                    ∧ UNCHANGED aSxx
                    ∧ UNCHANGED hqu;
```

Abbildung 12.17: Die Aktion `ValueToSensor`

```
ReturnFromAdapter(d : Data, cE0 : String) ≜
          ∧ cE0 = "03.actionReturn"
          ∧ Append(currentTrace, cE0) ∈ setOfPossibleTraces
          ∧ d = Head( adapterxqu)
          ∧ adapterxsAdapter = "returnEnqueued"
          ∧ adapterxsAdapter′ = "init"
          ∧ adapterxx′ = 0
          ∧ adapterxqu′ = Tail(adapterxqu)
          ∧ proxyxqu′ = Append(proxyxqu, d)
          ∧ proxyxss′ = IF proxyxss = "called"
                        THEN "returnEnqueued"
                        ELSE IF proxyxss = "calledError"
                        THEN "returnEnqError"
                        ELSE "returnEnqTimeout"
          ∧ currentTrace′ = Append(currentTrace, cE0)
          ∧ UNCHANGED proxyxx
          ∧ UNCHANGED ⟨aSxqu, aSxsSensor, aSxx⟩
          ∧ UNCHANGED hqu;
```

Abbildung 12.18: Die Aktion `ReturnFromAdapter`

12.1.4 Invariante *Inv3* und deren Beweis

In diesem Abschnitt soll die Korrektheit der Invariante *Inv3* (s. Abbildung 12.20) also $\Box Inv3$ gezeigt werden. Die Invariante *Inv3* besteht aus 18 Disjunkten. Die Disjunkte (D1) bis (D11) setzen alle Zustände und Transitionen in Beziehung, die durch Aktionen des Prozesses *SensorProxy* auftreten. Durch die Disjunkte (D12) bis (D14) werden die Zustände spezifiziert, die durch die Aktionen des Prozesses *SensorAdapter* betreten werden, wenn $proxyxss =''called''$ gilt. Der Existenzquantor in (E1) beschreibt die zulässigen Werte für die Zustandsvariable *proxyxs* mittels der gebundenen Variable x. Durch das Disjunkt (D15) werden die Zustände, die vor dem Schalten der Aktion *ValueToSensor* gelten, beschrieben. Das Disjunkt (D16) beschreibt die Zustände in denen der *getValue*-Aufruf in der Warteschlange des *Sensor*-Prozesses enthalten ist, die vor dem Schalten der Aktion *SensorDequeue* gelten. Durch das Disjunkt (D17) werden die Zustände, die vor dem Schalten der Aktion *SensorProcess* gelten, spezifiziert. Durch das Disjunkt (D18), werden die möglichen Zustände behandelt, die vor dem Schalten der Aktion *ReturnFromSensor* gelten.

Für das Initialisierungsprädikat *Init* muss gelten, dass *Inv3* impliziert wird. Dies gilt trivialerweise wegen (D15) in (12.21).

$$Init \Rightarrow Inv3(D15) \tag{12.21}$$

Die Aktion *ProxyError2*, die in Abbildung 12.14 gezeigt ist, wird durch die Disjunkte (D16),

```
SensorDequeue(cE0 : String) ≜
          ∧ cE0 = "04.actionDequeue"
          ∧ Append(currentTrace, cE0) ∈ setOfPossibleTraces
          ∧ aSxsSensor = "enqueued"
          ∧ aSxsSensor′ = "dequeued"
          ∧ aSxqu′ = Tail(aSxqu)
          ∧ currentTrace′ = Append(currentTrace, cE0)
          ∧ UNCHANGED ⟨proxyxqu, proxyxss, proxyxx⟩
          ∧ UNCHANGED ⟨adapterxsAdapter, adapterxqu, adapterxx⟩
          ∧ UNCHANGED aSxx
          ∧ UNCHANGED hqu;
```

Abbildung 12.19: Die Aktion `SensorDequeue`

```
Inv3 ≜
(D1-3)   ∨ proxyxss = "init" ∨ proxyxss = "callEnqueued" ∨ proxyxss = "callDequeued"
(D4-D6)  ∨ proxyxss = "processed" ∨ proxyxss = "readyForCall" ∨ proxyxss = "readyForError"
(D7-D8)  ∨ proxyxss = "returnEnqError" ∨ proxyxss = "returnDeqError"
(D9-D11) ∨ proxyxss = "returnDequeued" ∨ proxyxss = "Error" ∨ proxyxss = "returnEnqueued"
(D12-D13)∨ adapterxsAdapter = "enqueued" ∨ adapterxsAdapter = "dequeued"
(D14)    ∨ adapterxsAdapter = "returnEnqueued"
(E1)     ∨ ∃ x ∈ {"called", "calledError"} :
(D15)          ∨ (proxyxss = x ∧ proxyxx = 0 ∧ proxyxqu = ≪ ≫ ∧
                  adapterxsAdapter = "dequeued" ∧ adapterxx ∈ Data ∧ adapterxqu = ≪ ≫ ∧
                  aSxsSensor = "init" ∧ aSxx = 0 ∧ aSxqu = ≪ ≫))
(D16)          ∨ (proxyxss = x ∧ proxyxx = 0 ∧ proxyxqu = ≪ ≫ ∧
                  adapterxsAdapter = "called" ∧ adapterxx ∈ Data ∧ adapterxqu = ≪ ≫ ∧
                  aSxsSensor = "enqueued" ∧ aSxx = 0 ∧ oneCallInQueue(aSxqu))
(D17)          ∨ (proxyxss = x ∧ proxyxx = 0 ∧ proxyxqu = ≪ ≫ ∧
                  adapterxsAdapter = "called" ∧ adapterxx ∈ Data ∧ adapterxqu = ≪ ≫ ∧
                  aSxsSensor = "dequeued" ∧ aSxx ∈ Data ∧ aSxqu = ≪ ≫)
(D18)          ∨ (proxyxss = x ∧ proxyxx = 0 ∧ proxyxqu = ≪ ≫ ∧
                  adapterxsAdapter = "called" ∧ adapterxx ∈ Data ∧ adapterxqu = ≪ ≫ ∧
                  aSxsSensor = "processed" ∧ aSxx ∈ Data ∧ aSxqu = ≪ ≫ ))
```

Abbildung 12.20: Die Invariante `Inv3`

D(17), (D18) schaltbereit. In allen Fällen gilt nach dem Schalten das Disjunkt (D10) von *Inv2*.

$$ProxyError2(K1, K2, K3) \ \wedge \ Inv0(D8) \ \wedge \ Inv3(D16) \ \Rightarrow \ Inv3'(D10) \tag{12.22}$$

$$ProxyError2(K1, K2, K3) \ \wedge \ Inv0(D9) \ \wedge \ Inv3(D17) \ \Rightarrow \ Inv3'(D10) \tag{12.23}$$

$$ProxyError2(K1, K2, K3) \ \wedge \ Inv0(D10) \ \wedge \ Inv3(D18) \ \Rightarrow \ Inv3'(D10) \tag{12.24}$$

Für alle übrigen Disjunkte (D_i) $i \ \in \{1, \ldots, 18\} \setminus \{16, 17, 18\}$ ist die linke Seite der Implikationen (12.22), (12.35), (12.36) falsch.

Die in Abbildung 12.19 dargestellte Aktion *SensorDequeue* kann schalten, sobald das Disjunkt (D16) von *Inv3* gilt. Nach dem Schalten von *SensorProcess* gilt das Disjunkt (D17) von *Inv3*.

$$SensorDequeue(K1, K2, K3) \ \wedge \ Inv0(D8) \ \wedge \ Inv3(D16) \ \Rightarrow Inv3'(D17) \tag{12.25}$$

Mit Ausnahme des Disjunktes (D16) von *Inv3* ist die linke Seite der Implikation (12.25) nicht erfüllbar.

Die in Abbildung 12.21 dargestellte Aktion *SensorProcess* wird schaltbereit, sobald das Disjunkt (D17) von *Inv3* gilt. Nach dem Schalten von *SensorProcess* gilt das Disjunkt (D18) von *Inv3*.

$$SensorProcess(K1, K2, K3) \ \wedge \ Inv0(D9) \ \wedge \ Inv3(D17) \ \Rightarrow Inv3'(D18) \tag{12.26}$$

```
SensorProcess(cE0 : String) ≜
          ∧ cE0 = "04.actionProcess"
          ∧ Append(currentTrace, cE0) ∈ setOfPossibleTraces
          ∧ aSxsSensor = "dequeued"
          ∧ aSxsSensor' = "processed"
          ∧ currentTrace' = Append(currentTrace, cE0)
          ∧ UNCHANGED ⟨proxyxss, proxyxqu, proxyxx⟩
          ∧ UNCHANGED ⟨adapterxsAdapter, adapterxqu, adapterxx⟩
          ∧ UNCHANGED ⟨aSxqu, aSxx⟩
          ∧ UNCHANGED hqu;
```

Abbildung 12.21: Die Aktion `SensorProcess`

Für alle übrigen Disjunkte – außer dem Disjunkt (D17) – ist die linke Seite der Implikation (12.26) falsch.

```
ReturnFromSensor(d : Data, cE0 : String) ≜
          ∧ cE0 = "04.actionReturn"
          ∧ d = aSxx
          ∧ Append(currentTrace, cE0) ∈ setOfPossibleTraces
          ∧ aSxsSensor = "processed"
          ∧ adapterxsAdapter = "called"
          ∧ aSxsSensor′ = "init"
          ∧ aSxx′ = 0
          ∧ adapterxqu′ = Append( adapterxqu, d)
          ∧ adapterxsAdapter′ = "returnEnqueued"
          ∧ currentTrace′ = Append(currentTrace, cE0)
          ∧ UNCHANGED ⟨proxyxss, proxyxqu, proxyxx⟩
          ∧ UNCHANGED ⟨adapterxx, aSxqu⟩;
          ∧ UNCHANGED hqu;
```

Abbildung 12.22: Die Aktion `ValueFromSensor`

Bei der in Abbildung 12.22 gezeigten Aktion *ReturnFromSensor* führt das Disjunkt (D18) von *Inv3* zur Schaltbereitschaft. Nach dem Schalten der Aktion gilt dann das Disjunkt (D14) von *Inv3*:

$$ReturnFromSensor(K1, K2, K3, K4, K5) \wedge Inv0(D10) \wedge Inv3(D18) \Rightarrow Inv3'(D14) \quad (12.27)$$

Die Invariante bewirkt durch die Disjunkte (D1) bis D(11) von *Inv3*, dass sämtliche *SensorProxy*-Aktionen schaltbereit werden können.

Alle Aktionen des Prozesses *SensorAdapter* werden durch die Invariante *Inv3* wegen der Disjunkte (D12) bis (D14) schaltbereit.

12.1.5 Beweis der Verfeinerung für *SubsystemSensor*

Nun soll die Verfeinerung des Feinsystems *SubsystemSensor* mit den Prozessen *SensorProxy*, *SensorAdapter* und *Sensor* aus Abschnitt 11.2 und dem Grobsystem – bestehend aus dem Prozess *abstractSensor* aus Abschnitt 10.2 – betrachtet werden.

Um zu zeigen, dass ein Feinsystem FS ein Grobsystem GS implementiert, muss $FS \Rightarrow \overline{GS}$ gezeigt werden (s. Kapitel 5), wobei $\overline{GS}$ die Formel von GS beschreibt, in der alle Zustandsvariablen durch entsprechende Zustandsfunktionen auf den Zustandsvariablen von FS substituiert werden. Dies beweist die Schrittsimulation $InitFS \Rightarrow \overline{InitGS}$ und $NextFS \Rightarrow \overline{NextGS}$. Für jede Aktion des Feinsystems muss gezeigt werden, dass sie eine Aktion des Grobsystems impliziert bzw. zu einem Stotterschritt führt. Die verwendeten Zustandsfunktionen nennt man – wie in Abschnitt 5.5 angegeben – ein Refinement Mapping (Verfeinerungsabbildung).

Im Folgenden ist die Verfeinerungsabbildung *RM1* zwischen den Zustandsvariablen des *SubsystemSensor* und dem Prozess *abstractSensor* durch die Zustandsfunktionen *ox*, *ostate*, *oqu* und *ocurrentTrace* angegeben. Bei der Zustandsfunktion *oqu* wird die Historienvariable *hqu*, die den ursprünglich an den Prozess *SensorProxy* übergebenen Aufruf enthält, für die spätere Verarbeitung zwischengespeichert. Die Zustandsfunktion *oqu* erhält in Abhängigkeit vom Zustand den Wert *hqu* bzw. << >>.

```
ox ≜ proxyxx

ostate ≜
(R1)CASE proxyxss = "init" ∧ aSxsSensor ="init" ∧ aSxsSensor = "init"
         → "init"
(R2)CASE proxyxss = "callEnqueued" ∧ adapterxsAdapter = "init" ∧ aSxsSensor = "init"
         → "enqueued"
```

```
(R3)CASE proxyxss = "callDequeued" ∧ adapterxsAdapter = "init" ∧ aSxsSensor = "init"
        → "enqueued"
(R4)CASE proxyxss = "processed" ∧ adapterxsAdapter = "init" ∧ aSxsSensor = "init"
        → "enqueued"
(R5)CASE proxyxss = "readyForCall" ∧ adapterxsAdapter = "init" ∧ aSxsSensor = "init"
        → "enqueued"
(R6)CASE proxyxss = "called" ∧ adapterxsAdapter = "enqueued" ∧ aSxsSensor = "init"
        → "enqueued"
(R7)CASE proxyxss = "called" ∧ adapterxsAdapter = "dequeued" ∧ aSxsSensor = "init"
        → "enqueued"
(R8)CASE proxyxss = "called" ∧ adapterxsAdapter = "called" ∧ aSxsSensor = "enqueued"
        → "enqueued"
(R9)CASE proxyxss = "called" ∧ adapterxsAdapter = "called" ∧ aSxsSensor = "dequeued"
        → "dequeued"
(R10)CASE proxyxss = "called" ∧ adapterxsAdapter = "called" ∧ aSxsSensor = "processed"
        → "processed"
(R11)CASE proxyxss = "called" ∧ adapterxsAdapter = "returnEnqueued" ∧ aSxsSensor = "init"
        → "processed"
(R12)CASE proxyxss = "returnEnqueued" ∧ adapterxsAdapter = "init" ∧ aSxsSensor = "init"
        → "processed"
(R13)CASE proxyxss = "returnDequeued" ∧ adapterxsAdapter = "init" ∧ aSxsSensor = "init"
        → "processed"
(R14)CASE proxyxss = "readyForError" ∧ adapterxsAdapter = "init" ∧ aSxsSensor = "init"
        → "enqueued"
(R15)CASE proxyxss = "calledError" ∧ adapterxsAdapter = "enqueued" ∧ aSxsSensor = "init"
        → "enqueued"
(R16)CASE proxyxss = "calledError" ∧ adapterxsAdapter = "dequeued" ∧ aSxsSensor = "init"
        → "enqueued"
(R17)CASE proxyxss = "calledError" ∧ adapterxsAdapter = "called" ∧ aSxsSensor = "enqueued"
        → "enqueued"
(R18)CASE proxyxss = "calledError" ∧ adapterxsAdapter = "called" ∧ aSxsSensor = "dequeued"
        → "dequeued"
(R19)CASE proxyxss = "calledError" ∧ adapterxsAdapter = "called"
         ∧ aSxsSensor = "processed"
        → "processed"
(R20)CASE proxyxss = "calledError" ∧ adapterxsAdapter = "returnEnqueued"
         ∧ aSxsSensor = "init" → "processed"
(R21)CASE proxyxss = "returnDeqError" ∧ adapterxsAdapter = "init" ∧ aSxsSensor = "init"
        → "processed"
(R22)CASE proxyxss = "returnEnqError" ∧ adapterxsAdapter = "init" ∧ aSxsSensor = "init"
        → "processed"
(R23)CASE proxyxss = "Error" ∧ adapterxsAdapter = "init" ∧ aSxsSensor = "init"
        → "Error"

oqu ≜  IF(aSxsSensor = "dequeued" ∨ aSxsSensor = "processed")
       THEN ≪ ≫
       ELSE IF(proxyxss = "callEnqueued" ∨ proxyxss = "callDequeued" ∨
               proxyxss = "processed" ∨ proxyxss = "readyForCall" ∨
               proxyxss = "called" ∨ adapterxsAdapter = "dequeued" ∨
               adapterxsAdapter = "called" ∨ proxyxss = "calledError" )
       THEN hqu
       ELSE ≪ ≫

ocurrentTrace ≜ im Anhang D.6 angegeben
```

Die Tabelle 12.1 gibt die Zustände der normalen Verarbeitung an, für die $\langle N \rangle_f$ gilt. Die Stotterschritte werden separat im Text erläutert. Jede Zeile dieser Tabelle setzt einen Zustand des Feinsystems mit einem Zustand des Grobsystems in Beziehung. Dazu wird jede Zeile nummeriert. Anschließend werden die Feinsystem-Aktionen, für die der behandelte Zustand eine Vor- bzw. Nachbedingung bildet, angegeben. In der folgenden Spalte wird die Wertebelegung für die

Zustandsvariablen des betrachteten Zustandes angegeben. Die folgende Spalte gibt die Nummer des Falles für die Abbildung der Zustandsfunktion *ostate* an. Für den zugehörigen Zustand des Grobsystems werden in der folgenden Spalte die Aktionen, für die der betrachtete Zustand eine Vor- bzw. Nachbedingung ist, angegeben. In der letzten Spalte werden die Wertebelegungen der Zustandsvariablen für den zugehörigen Zustand des Grobsystems angegeben. Für die Wertebelegungen der Zustandsvariable *currentTrace* und der Zustandsfunktion *ocurrentTrace* werden Abkürzungen eingeführt, da diese Traces sehr lang sind. Für das Feinsystem werden diese Traces mit den Bezeichnern *ft1* bis *ft11* bezeichnet. Ebenso werden für das Grobsystem die Traces *gt1* bis *gt11* eingeführt.

Die Tabelle 12.2 besitzt den gleichen Aufbau wie die Tabelle 12.1, behandelt aber die fehlerbehaftete Verarbeitung. Auf der Basis der Tabellen werden die Argumentationen für den Beweis der Implikation einer Grobsystem-Aktion durch eine Feinsystem-Aktion geführt. Die Tabelle 12.2

Nr.	Bed. f. FS-Aktion	Wertebelegung der Zustandsvariablen des FS	ostate	Bed. f. GS-Aktion	Wertebelegung der Zustandsvariablen des GS
1	VB *Proxy-Enqueue*	$\left[\begin{array}{l} proxyxss ='' init'' \\ proxyxqu =<<>> \\ proxyxx = 0 \\ adapterxqu =<<>> \\ adapterxsAdapter ='' init'' \\ aSxsSensor ='' init'' \\ aSxqu =<<>> \\ currentTrace = ft_1 \end{array}\right]$	(R1)	VB *Enqueue*	$\left[\begin{array}{l} ostate ='' init'' \\ oqu =<<>> \\ ox = rand(1) \\ ocurrentTrace = gt_1 \end{array}\right]$
2	NB *Proxy-Enqueue* VB *Proxy-Dequeue*	$\left[\begin{array}{l} proxyxss ='' enqueued'' \\ proxyxqu =<< d >> \\ proxyxx = 0 \\ adapterxqu =<<>> \\ adapterxsAdapter ='' init'' \\ aSxsSensor ='' init'' \\ aSxqu =<<>> \\ currentTrace = ft_2 \end{array}\right]$	(R2)	NB *Enqueue* VB *Dequeue*	$\left[\begin{array}{l} ostate ='' enqueued'' \\ oqu =<< d >> \\ ox = rand(1) \\ ocurrentTrace = gt_2 \end{array}\right]$
3	NB *Sensor-Dequeue* VB *Sensor-Process*	$\left[\begin{array}{l} proxyxss ='' called'' \\ proxyxqu =<<>> \\ proxyxx = 0 \\ adapterxqu =<<>> \\ adapterxsAdapter ='' init'' \\ aSxsSensor ='' init'' \\ aSxqu =<<>> \\ currentTrace = ft_3 \end{array}\right]$	(R3)	VB *Process* NB *Dequeue*	$\left[\begin{array}{l} ostate ='' dequeued'' \\ oqu =<< >> \\ ox = rand(1) \\ ocurrentTrace = gt_3 \end{array}\right]$
4	NB *Sensor-Process*	$\left[\begin{array}{l} proxyxss ='' called'' \\ proxyxqu =<<>> \\ proxyxx = 0 \\ adapterxqu =<<>> \\ adapterxsAdapter ='' called'' \\ aSxsSensor ='' processed'' \\ aSxqu =<<>> \\ currentTrace = ft_4 \end{array}\right]$	(R10)	NB *Process*	$\left[\begin{array}{l} ostate ='' processed'' \\ oqu =<< d >> \\ ox = d \\ ocurrentTrace = gt_4 \end{array}\right]$

Tabelle 12.1: Die wesentlichen Zustände der normalen Verarbeitung und deren Abbildung

Nr.	Bed. f. FS-Aktion	Wertebelegung der Zustandsvariablen des FS	*ostate*	Bed. f. GS-Aktion	Wertebelegung der Zustandsvariablen des GS
1	VB *Proxy-Error1*	$proxyxss ='' readyForError''$ $proxyxqu =<<>>$ $proxyxx = 0$ $adapterxqu =<<>>$ $adapterxsAdapter ='' init''$ $aSxsSensor ='' init''$ $aSxqu =<<>>$ $currentTrace = ft_6$	(R14)	VB *Enqueue-Error*	$ostate ='' enqueued''$ $oqu =<< >>$ $ox = rand(1)$ $ocurrentTrace = gt_6$
2	VB *Proxy-Error2*	$proxyxss ='' calledError''$ $proxyxqu =<<>>$ $proxyxx = 0$ $adapterxqu =<<>> \vee$ $<< d >>$ $adapterxsAdapter = s_1$ $aSxsSensor = s_2$ $aSxqu =<<>> \vee << d >>$ $currentTrace = ft_7$	(R15)-(R20)		$ostate = s_3$ $oqu =<<>>$ $ox = rand(1)$ $ocurrentTrace = gt_7$
3	VB *Proxy-Error3*	$proxyxss ='' returnEnqError''$ $proxyxqu =<< d >>$ $proxyxx = 0$ $adapterxqu =<<>>$ $adapterxsAdapter ='' init''$ $aSxsSensor ='' init''$ $aSxqu =<<>>$ $currentTrace = ft_8$	(R22)	VB *Process-Error*	$ostate ='' processed''$ $oqu =<<>>$ $ox = rand(1)$ $ocurrentTrace = gt_8$
4	VB *Proxy-Error4*	$proxyxss ='' returnDeqError''$ $proxyxqu =<<>>$ $proxyxx = 0$ $adapterxqu =<<>>$ $adapterxsAdapter ='' init''$ $aSxsSensor ='' init''$ $aSxqu =<<>>$ $currentTrace = ft_9$	(R21)	VB *Process-Error*	$ostate ='' processed''$ $oqu =<<>>$ $ox = rand(1)$ $ocurrentTrace = gt_9$
5	NB *Proxy-Error1 bis Proxy-Error4*	$proxyxss ='' Error''$ $proxyxqu =<<>>$ $proxyxx = 0$ $adapterxqu =<<>>$ $adapterxsAdapter ='' init''$ $aSxsSensor ='' init''$ $aSxqu =<<>>$ $currentTrace = ft_{11}$	(R23)	NB *Enqueue-Error*, *Dequeue-Error*, *Process-Error*	$ostate ='' Error''$ $oqu =<<>>$ $ox = 0$ $ocurrentTrace = gt_{11}$

Tabelle 12.2: Die wesentlichen Zustände der fehlerbehafteten Verarbeitung und deren Abbildung

enthält die Mengen s_1, s_2 und s_3. Mit s_1 wird die Menge aller Werte der Zustandsvariable *adapterxsAdapter* mit Ausnahme von "init" angegeben. Die Menge s_2 enthält alle Werte der Zustandsvariable *aSxsSensor*. Weiterhin wird die Menge s_3 durch $s_3 = \{''enqueued'','' dequeued'','' processed''\}$

definiert.

Die Korrektheit der Verfeinerungsabbildung für die Initialzustände des Feinsystems und des Grobsystems $InitFS \Rightarrow \overline{InitGS}$ ergibt sich aus der ersten Zeile der Tabelle 12.1. Zunächst soll die korrekte Verfeinerung durch die Aktionen des Prozesses *SensorProxy* bewiesen werden.

Zunächst soll an der Aktion *ValueToProxy* des *SubsystemSensor* ein Beweisschritt eines Verfeinerungsbeweises demonstriert werden. Dabei muss gezeigt werden, dass ein Schalten der Aktion *ValueToProxy* die Aktion *enqueue* des Grobsystems mit dem definierten Refinement Mapping *RM1* impliziert. Für den Beweisschritt werden die bereits bewiesenen Invarianten *Inv0* und *Inv1* aus den Abschnitten 12.1.1 und 12.1.2 herangezogen. Die Aktion *ValueToProxy* wird durch das Disjunkt (D3) von *Inv1* schaltbereit. Bei (D3) von *Inv1* gilt aber $proxyxss ='' init'' \wedge aSxsSensor ='' init'' \wedge aSxsSensor ='' init''$. Mit dem Fall (R1) der Zustandsfunktion *ostate* ergibt sich trivialerweise für *ostate* der Wert $''init''$, damit wird die Aktion *Enqueue* schaltbereit. Die übrigen Zustandsfunktionen *oqu* und *ox* sind für das Schalten nicht relevant. Nach dem Schalten der Aktion *ValueToProxy* gilt das Disjunkt (D4) der Invariante *Inv1* bei dem $proxyxss ='' callEnqueued''$ ist. Für dieses Disjunkt gilt der Fall (R2) der Zustandsfunktion *ostate*, womit sich $ostate ='' enqueued''$ ergibt, was auch nach dem Schalten von *Enqueue* gilt. Weiterhin ist nach den Schalten von *Enqueue* $oqu =<< d >>$, was für $proxyxss ='' callEnqueued''$ durch die Zustandsfunktion *oqu* gegeben ist. Die Zustandsfunktion *ox* ist für diesen Beweisschritt nicht relevant, da die Zustandsvariablen *proxyx* und x nicht durch die beteiligten Aktionen geändert werden. Damit ist die Gültigkeit der Implikation gezeigt. Die Implikation (12.28) entspricht einem Zustandsübergang von Zeile 1 in Zeile 2 der Tabelle 12.1.

$$\exists\, d \in Data : ValueToProxy(d) \wedge Inv0(D1) \wedge Inv1(D3) \Rightarrow \exists\, d \in Data : Enqueue(d) \tag{12.28}$$

Nun soll die Aktion *ProxyDequeue* betrachtet werden. Die Aktion wird durch die Disjunkte (D4), (D9) und (D11) von *Inv1* schaltbereit. Nach dem Schalten gilt dann eines der Disjunkte (D5), (D10) oder (D14) von *Inv1*. Die Implikation (12.29) gibt einen Stotterschritt an:

$$\begin{bmatrix} ostate ='' enqueued'' \\ oqu =<< d >> \\ ox = 0 \\ ocurrentTrace = t_1 \end{bmatrix} \Rightarrow \begin{bmatrix} ostate ='' enqueued'' \\ oqu =<< d >> \\ ox = 0 \\ ocurrentTrace = t_1 \end{bmatrix} \text{ womit (12.29) erfüllt ist.}$$

$$ProxyDequeue \wedge Inv0(D2, D12, D15) \wedge Inv1(D4, D9, D11) \Rightarrow [\overline{NextGS}]_{\overline{vars}} \tag{12.29}$$

Vor dem Schalten der Aktion *ProxyProcess* muss (D3) von *Inv0* und (D5) von *Inv1* erfüllt sein. Wie in Implikation (12.30) angegeben, handelt es sich im Grobsystem um einen Stotterschritt, wodurch die Implikation (12.30) gilt.

$$\exists\, cEO \in EO : ProxyProcess(cEO) \wedge Inv0(D3) \wedge Inv1(D5) \Rightarrow [\overline{NextGS}]_{\overline{vars}} \tag{12.30}$$

Die Aktion *setCall1* wird durch das Disjunkt (D6) von *Inv1* schaltbereit und wird auf einen Stotterschritt im Grobsystem, wie in der Implikation (12.31) gezeigt, abgebildet.

$$\exists\, cEO \in EO : setCall1(cEO) \wedge Inv0(D4) \wedge Inv1(D6) \Rightarrow [\overline{NextGS}]_{\overline{vars}} \tag{12.31}$$

Jetzt werden die Aktionen des Fehlerzweiges betrachtet, deren Zustände in Tabelle 12.2 angegeben sind. Die gleiche Argumentation wie für die Aktion *setCall1* gilt für die Aktion *setCall2*, wie in (12.32) angegeben. Die Vorbedingungen der Aktion *ProxyError1* bilden das Disjunkt (D14) von *Inv0* und das Disjunkt (D8) von *Inv1*. Durch die Implikation (12.33) wird der Übergang von Zeile 1 in Zeile 5 der Tabelle 12.2 angegeben, der durch die Aktion *EnqueueError* des Grobsystems ausgelöst wird. Vor dem Schalten ist (D8) von *Inv1* gültig, wodurch die Implikation (12.33) wegen (R23) gilt.

$$\exists\, cEO \in EO : setCall2(cEO) \wedge Inv0(D4) \wedge Inv1(D6) \Rightarrow [\overline{NextGS}]_{\overline{vars}} \tag{12.32}$$

$$\exists\, cEO \in EO : ProxyError1(cEO) \wedge Inv0(D14) \wedge Inv1(D8) \Rightarrow EnqueueError \tag{12.33}$$

Die Aktion *ProxyError2* wird sowohl durch Disjunkte der Invariante *Inv1* als auch der Invariante *Inv2* schaltbereit. Für die Invariante *Inv2* sind das die Disjunkte (D7), (D8), (D9) und (D10), wenn gleichzeitig eines der Disjunkte (D6), (D7) bzw. (D8) von *Inv0* erfüllt ist. Für diesen Fall impliziert die Aktion *ProxyError2* die Aktion *EnqueueError* des Grobsystems, wie in Implikation (12.34) angegeben. Die Implikation (12.34) beschreibt einen Übergang von Zeile 2 in Zeile 5 der Tabelle 12.2.

$$\exists\, cEO \in EO : ProxyError2(cEO) \wedge Inv0(D6, D7, D8, D11) \wedge Inv2(D7, D8, D9, D10) \Rightarrow EnqueueError \tag{12.34}$$

Für die Disjunkte der Invariante *Inv3* werden mehrere Grobsystemaktionen – nämlich *DequeueError* und *ProcessError* – impliziert, wenn eines der Disjunkte (D9) und (D10) von *Inv0* erfüllt ist.

Weiterhin beschreibt die Implikation (12.35) ebenso einen Übergang von Zeile 2 in Zeile 5 der Tabelle 12.2.

$$\exists\, cEO \in EO : ProxyError2(cEO) \wedge Inv0(D9) \wedge Inv3(D17) \Rightarrow DequeueError \tag{12.35}$$

Abschließend wird durch die Implikation (12.36) auch ein Übergang von Zeile 2 in Zeile 5 der Tabelle 12.2 angegeben.

$$\exists\, cEO \in EO : ProxyError2(cEO) \wedge Inv0(D10) \wedge Inv3(D18) \Rightarrow ProcessError \tag{12.36}$$

Die Aktion *ProxyError3* wird durch das Disjunkt (D15) von *Inv0* und das Disjunkt (D11) von *Inv1*, wie in Implikation (12.37) gezeigt, schaltbereit. Somit impliziert diese Aktion die Grobsystemaktion *ProcessError*. Die Implikation gibt einen Zustandsübergang von Zeile 3 in Zeile 5 der Tabelle 12.2 an.

$$\exists\, cEO \in EO : ProxyError3(cEO) \wedge Inv0(D15) \wedge Inv1(D11) \Rightarrow ProcessError \tag{12.37}$$

Mit der Implikation (12.38) wird der Zustandsübergang von Zeile 4 in Zeile 5 der Tabelle 12.2 beschrieben.

$$\exists\, cEO \in EO : ProxyError4(cEO) \wedge Inv0(D16) \wedge Inv1(D10) \Rightarrow ProcessError \tag{12.38}$$

Jetzt wird die Korrektheit der Verfeinerung für die Aktionen des Prozesses *SensorAdapter* unter Verwendung der Invarianten gezeigt.

Die Aktionen *ValueToAdapter*, *AdapterDequeue* und *ValueToSensor* führen zu einem Stotterschritt, wie in den Implikationen (12.39), (12.40) und (12.41) angegeben, sodass die Verfeinerung erfüllt ist.

$$\exists\, cEO \in EO, \exists\, d \in Data : ValueToAdapter(cEO, d) \wedge Inv0(D5, D14) \wedge Inv2(D6) \Rightarrow [\overline{NextGS}]_{\overline{vars}} \tag{12.39}$$

$$\exists\, cEO \in EO : AdapterDequeue(cEO) \wedge Inv0(D6) \wedge Inv2(D7) \Rightarrow [\overline{NextGS}]_{\overline{vars}} \quad (12.40)$$

$$\exists\, cEO \in EO, \exists\, d \in Data : ValueToSensor(cEO, d) \wedge Inv0(D8) \wedge Inv2(D8) \Rightarrow [\overline{NextGS}]_{\overline{vars}} \quad (12.41)$$

Mit der Vorbedingung $Inv0(D10)$ und $Inv2(D11)$ wird die Aktion *ReturnFromAdapter* schaltbereit und führt zu einem Stotterschritt im Grobsystem.

$$\exists\, cEO \in EO, \exists\, d \in Data : ReturnFromAdapter(cEO, d) \wedge Inv0(D11) \wedge Inv2(D10) \Rightarrow [\overline{NextGS}]_{\overline{vars}} \quad (12.42)$$

Nun soll für die Aktionen des Prozesses *Sensor* die korrekte Verfeinerung unter Verwendung der Invarianten *Inv0* und *Inv3* nachgewiesen werden.

Die Aktion *SensorDequeue* wird durch das Disjunkt (D16) der Invariante *Inv3* schaltbereit. Die Implikation (12.43) gibt einen Übergang von Zeile 2 in Zeile 3 aus der Tabelle 12.1 an.

$$\exists\, cEO \in EO : SensorDequeue(cEO) \wedge Inv0(D8) \wedge Inv3(D16) \Rightarrow Dequeue \quad (12.43)$$

Die Aktion *SensorProcess* impliziert die Aktion *Process* des Grobsystems. Die Implikation (12.44) gibt einen Zustandsübergang von Zeile 3 in Zeile 4 der Tabelle 12.1 an.

$$\exists\, cEO \in EO : SensorProcess(cEO) \wedge Inv0(D9) \wedge Inv3(D17) \Rightarrow Process \quad (12.44)$$

Durch die Implikation (12.45) wird ein Stotterschritt angegeben.

$$\exists\, cEO \in EO, \exists\, d \in Data : ReturnFromSensor(cEO, d) \wedge Inv0(D10) \wedge Inv3(D18) \Rightarrow [\overline{NextGS}]_{\overline{vars}} \quad (12.45)$$

12.2 Nachweis der Verfeinerung des *SubsystemController*

Für dieses Subsystem muss nachgewiesen werden, dass der in Abschnitt 11.2.1 spezifizierte cTLA-Prozess eine korrekte Verfeinerung des Prozesses *abstractController*, der in Kapitel 10 erläutert und in Anhang B.1 gezeigt wird, ist. Es muss also $abstractController \Rightarrow concreteController$ gelten.

Hier soll kurz die Idee der Verfeinerungsabbildung, die die Grundlage für die maschinelle Verifikation durch den TLC gebildet hat, angeben werden. Die Abbildung 12.23 zeigt zwölf Zustandsfunktionen. Die Zustandsvariablen *x*, *y* *sync* und *qu* werden auf sich selbst abgebildet. Die Zustandsvariablen, die für Kontrollzustände und Objektflüsse eingeführt worden sind, werden ebenfalls auf sich selbst abgebildet. Wesentliche Unterschiede gibt es nur bei der Zustandsfunktion *ostate*, da die Zustände "s1" bis "s12" zusätzlich aufgenommen worden sind. Die Zustandsfunktion bildet Werte der Zustandsvariablen *state* des Feinsystems auf Werte der Zustandsvariablen *state* des Grobsystems ab. Alle Werte, die sich um den Wert "waitSensor" der Zustandsvariable *state* gruppieren – nämlich "s1" bis "s4" – werden auf diesen Wert abgebildet. Für den Fall der Werte "s5" bis "s8" werden diese auf den Wert "valuesComputed" der Zustandsvariablen *state* abgebildet. Schließlich werden die Werte "s9" bis "s12" der Zustandsvariable *state* auf den Wert "waitActuator" des Grobsystems abgebildet.

```
ox = x
oy = y
oqu =  IF state = "s1" ∨ state = "s2" ... state = "s12"
       THEN ≪ ≫
       ELSE qu
oCtrlStartActReadx ≜ ctrlStartActReadx
oCtrlActReadxActReady ≜ ctrlActReadxActReady
oExists01 ≜ exists01
oValue01 ≜ value01
oExists02 ≜ exists02
oValue02 ≜ value02
oExists03 ≜ exists03
oValue03 ≜ exists03
ostate ≜
      CASE state = "init"
      CASE state = "init" → "init"
      CASE state = "stopped" → "stopped"
      CASE state = "s1" → "waitSensor"
      CASE state = "s2" → "waitSensor"
      CASE state = "s3" → "waitSensor"
      CASE state = "s4" → "waitSensor"
      CASE state = "s5" → "valuesComputed"
      CASE state = "s6" → "valuesComputed"
      CASE state = "s7" → "valuesComputed"
      CASE state = "s8" → "valuesComputed"
      CASE state = "s9" → "waitActuator"
      CASE state = "s10" → "waitActuator"
      CASE state = "s11" → "waitActuator"
      CASE state = "s12" → "waitActuator"
      CASE state = "wait" → "wait"
      CASE state = "waitSensor" → "waitSensor"
      CASE state = "getEnqueued" → "getEnqueued"
      CASE state = "valueDequeued" → "valueDequeued"
      CASE state = "valuesComputed" → "valuesComputed"
      CASE state = "waitActuator" → "waitActuator"
      CASE state = "returnEnqueued" → "returnEnqueued"
```

Abbildung 12.23: Die Zustandsfunktionen für die Zustandsabbildung `RM2` zwischen dem `SubsystemSensor` und dem `abstractController`

Die Bildschirmausgabe von TLC nach der Durchführung des Verfeinerungsbeweises für das Verfeinerungsmuster ist im Folgenden aufgeführt:

> Model checking completed. No error has been found. Estimates of the probability that TLC did not check all reachable states because two distinct states had the same fingerprint: calculated (optimistic): 2.282462413516484E-15 based on the actual fingerprints: 9.967922017819474E-15 429 states generated, 152 distinct states found, 0 states left on queue. The depth of the complete state graph search is 14.

Dies ist unter Verwendung von TLC für drei Datenwerte auf einem Rechner mit einem AMD Athlon 64 X2 3800+ Prozessor und 1.0 GB Arbeitsspeicher für die Sun JVM der Version 1.5 in 30 Sekunden vorgenommen worden. Insgesamt wurden für den Verfeinerungsbeweis durch TLC 429 Zustände erzeugt, von denen 152 Zustände unterscheidbar waren. Die Tiefe bei der Suche im Zustandsgraphen beträgt 14.

Die vollständige Spezifikation dieses Subsystems ist, wie sie für TLC verwendet worden ist, im Anhang D.5 angegeben. Zur Verifikation einer Verfeinerung wird das **INSTANCE** Schlüsselwort

aus TLC verwendet, um das Grobsystem, das hier durch den Prozess *abstractController* gebildet wird, zu instantiieren. Durch AT wird somit eine Instanz des Prozesses *abstractController* beschrieben, in der die Zustandsvariablen x, y, *state*, *qu*, *sync* durch die Zustandsfunktionen *ox*, *oy*, *ostate*, *oqu* und *osync* substituiert werden. Die Zustandsfunktionen aus Abbildung 12.23 werden dazu in die Spezifikation des Feinsystems aufgenommen. Um zu prüfen, ob die kanonische Formel *Spec* des Feinsystems die kanonische Formel *GSpec* des Grobsystems impliziert, muss eine Property in der Spezifikation aufgenommen werden, die anschließend durch Eintrag in die Konfigurationsdatei mittels TLC verifiziert wird.

12.3 Verfeinerungsbeweis der *Tick*-Aktionen des Grob- und des Feinsystems

In diesem Abschnitt wird die Verfeinerung der *Tick*-Aktionen, die durch die *Tick*-Aktionen der einzelnen Prozesses gebildet werden, betrachtet. Um die Korrektheit der Verfeinerung unter Einbeziehung der Wartezeiten von Aktionen zu beweisen, ist zu zeigen, dass *TickFS* eine korrekte Verfeinerung von *TickGS* ist. Dieser Beweis wird nicht mit der Unterstützung des Model-Checkers TLC durchgeführt, sondern wird manuell vorgenommen, weil die vorliegende TLC Version einen Ausdruck der Form:

$$enabled(TimerAktion_i) \Rightarrow now' \leq now + (gt_i - TimerAktion_i)$$

nicht unterstützt. Allerdings wird in [Lam05] beschrieben, wie man Realzeitbeweise mit TLC durchführt, wobei Lamport die Tauglichkeit von TLC für Realzeitbeweise beschreibt. Die Aktion *TickGS* des Grobsystems ist in Abbildung 10.18 aufgeführt. Die Aktion erhöht die Uhrenvariable *now* mit jedem Schalten um höchstens ϵ, das die Auflösung zur Erhöhung von *now* angibt. Zusätzlich wird die Hilfsvariable z eingeführt, die später verwendet wird, um zu beweisen, dass kein Zeno-Verhalten vorliegt. Weiterhin erhöht die Aktion *TickGS* für jede Aktion mit einer persistenten, maximalen Wartezeit des Grobsystems eine eigenständige Timervariable, die nur bis zu dem Wert des jeweiligen P MAX TIME Ausdrucks, der hier jeweils für jede Aktion durch eine Konstante gt_i angegeben wird, ansteigen darf.

Um zu zeigen, dass $TickFS \Rightarrow TickGS$ gilt, muss eine Verfeinerungsabbildung für den Implikationsbeweis konstruiert werden. Diese wird auf der Basis von zusätzlichen Hilfsvariablen – nämlich $h_1, ...h_{10}$ – , die in der Aktion *TickFS* des Feinsystems aus Abbildung 11.25 berechnet werden, definiert. In einer Hilfsvariablen werden die Zeiten, in denen mehrere Feinsystem-Aktionen schaltbereit sind, aufsummiert. Die Abbildung 12.24 zeigt exemplarisch die Hilfsvariablen h_1 und h_2. Die übrigen Hilfsvariablen werden im Anhang E.1 angegeben. Eine Hilfsvariable liefert somit einen summierten Beitrag zur Wartezeit des Timers einer Grobsystemaktion, wie man am etwa Beispiel der Hilfsvariable h_2 sieht, die alle Zeiten für schaltbereite Aktionen der normalen Verarbeitung, bis der *getValue*-Aufruf den *Sensor*-Prozess erreicht, aufsummiert. Für diese Hilfsvariablen, die Schaltzeiten mehrerer Aktionen des Feinsystems addieren, wird im Fall von THEN, die Hilfsvariable h_i jeweils um $h_i - now + now'$ erhöht. Die Zustandsfunktionen, die die Verfeinerungsabbildung definieren, sind in der Tabelle 12.3 angegeben. Die Timervariablen des Feinsystems werden als interne Variablen betrachtet, die keinen direkten Einfluss auf die Verfeinerung haben.

```
h1' = IF(enabled(ProxyEnqueue(d)) ∨ enabled(ProxyDequeue)
         ∨ enabled(ProxyProcess) ∨ enabled(setCall1) ∨
         ∨ enabled(getValueCall(d)) ∨ enabled(AdapterDequeue) ∨
         ∨ enabled(AdapterProcess) ∨ enabled(getValueCall(d) ∨
         ∨ SensorDequeue))
    THEN h1 + (now' - now)
∧   ELSE h1
h2' = IF(enabled(SensorDequeue))
    THEN h2 + (now' - now)
∧   ELSE h2
...
```

Abbildung 12.24: Die Hilfsvariablen h1 und h2 für Timervariablen

Im Folgenden werden die Implikationen einiger ausgewählter Konjunkte bewiesen. Die ersten vier Konjunkte der Aktionen *TickFS* und *TickGS* stimmen überein, womit die Implikation trivialerweise gilt. Die **IF-THEN-ELSE** Ausdrücke, die die zusätzlich eingeführten Hilfsvariablen erhöhen, implizieren **IF-THEN-ELSE** Ausdrücke, die zur Erhöhung der Timervariablen des Grobsystems führen. Die Timervariablen des Feinsystems, die zu schaltbereiten Feinsystem-Aktionen gehören, liefern einen Beitrag zu den Werten der Hilfsvariablen. Sie führen zu Stotterschritten

Urbild der Zustandsfunktion aus TickFS	*Bild der Zustandsfunktion aus TickGS*
now	now
z	z
h_1	timerEnqueue[d]
h_2	timerDequeue
h_3	timerProcess
h_4	timerDequeueError
h_5	timerEnqueueError
h_6	timerProcessError
h_7	timerDequeueTimeout
h_8	timerEnqueueTimeout
h_9	timerProcessTimeout
h_{10}	timerReturn[d]

Tabelle 12.3: Die Zustandsfunktionen der zeitbehafteten Variablen des Feinsystems auf die zeitbehafteten Variablen des Grobsystems

des Grobsystems, da zwar die Zeit voranschreitet, aber im Grobsystem keine Aktion schaltet. Für Ausdrücke der Form $enabled(TimerAktion_i) \Rightarrow now' \leq now + (gt_i - TimerAktion_i)$ gilt die im folgenden Beispiel aufgeführte Beweisidee unter der Voraussetzung (12.46), wobei die Summe alle Zeiten der schaltbereiten Aktionen des Feinsystems bis der *getValueCall* den *Sensor*-Prozess erreicht, angibt:

$$gt_1 \geq \sum_{i=0}^{7} ft_i \tag{12.46}$$

Weiterhin gilt trivialerweise (12.47), da jede Feinsystem Aktion einen positiven Beitrag oder null liefert:

$$gt_1 \geq \sum_{i=0}^{7} ft_i \geq \sum_{i=0}^{6} ft_i \geq \sum_{i=0}^{5} ft_i \ldots \geq \sum_{i=0}^{1} ft_i \tag{12.47}$$

Für den Ausdruck $enabled(ValueToProxy)$ gilt:

$$h_1 \leq ft_1 oder - h_1 \geq -ft_1 \tag{12.48}$$

Der Spezialfall für die erste Aktion ist:

$$h_1 \leq \sum_{i=0}^{7} ft_i \Leftrightarrow -h_1 \geq -\sum_{i=0}^{7} ft_i \tag{12.49}$$

Um zu vermeiden, dass *TickFS* den Wert von *now* über einen Wert erhöht, der die maximale Verweildauer von *ProxyDequeue* übersteigt, ist in *TickFS* das folgende Konjunkt spezifiziert:

$$enabled(ProxyDequeue) \Rightarrow now' \leq now + (ft_2 - TimerProxyDequeue) \tag{12.50}$$

Wird zusätzlich $ft1 \geq t_{ValueToProxy}$ addiert und wieder subtrahiert gilt:

$$now' \leq now + (ft_1 + ft_2 - (TimerProxyDequeue + t_{ValueToProxy})) \tag{12.51}$$

Für h_1 gilt aber:

$$\begin{aligned} h_1 \leq TimerProxyDequeue + t_{ValueToProxy} \quad \Leftrightarrow \\ -h_1 \geq -(TimerProxyDequeue + t_{ValueToProxy}) \end{aligned} \tag{12.52}$$

Wird h_1 in (12.50) eingesetzt, ergibt sich für das Feinsystem:

$$now' \leq now + (ft_1 + ft_2 - h_1) \tag{12.53}$$

Die Ungleichung (12.53) impliziert mit (12.47)

$$enabled(Enqueue(d)) \Rightarrow now' \leq now + (gt_1 - timerEnqueue[d]), \tag{12.54}$$

in *TickGS*, da h_1 auf *TimerEnqueue* durch eine Zustandsfunktion abgebildet wird, wie sie in Tabelle 12.3 gezeigt ist. Die übrigen Hilfsvariablen werden nach dem selben Prinzip bewiesen.

12.4 Ausschluss des Zeno-Verhaltens von *Tick*

In diesem Abschnitt soll gezeigt werden, dass die Uhrenvariable *now* über alle Grenzen steigen kann und kein Zeno-Verhalten, wie in Kapitel 6.3.3 beschrieben, auftritt. Hierfür sind Beweise mit der Lattice Regel erforderlich. Um diese durchzuführen werden in der Regel auch SF1 bzw. WF1 Beweise durchgeführt, um zu zeigen, dass man sich in einer Zustandsfolge wegen starker bzw. schwacher Fairnessannahmen weiterbewegt. Dadurch entsteht eine Verschachtelung von Lattice und WF1 bzw. SF1 Beweisen. Häufig werden solche Beweise rekursiv geführt. Zunächst wird jetzt das Diskretisierungsintervall H_i definiert (12.55):

$$H_i \equiv z + (i-1) * \epsilon > t \geq z + (i-2) * \epsilon \tag{12.55}$$

Dann wird ein rekursiv angesetzter Lattice-Beweis verwendet (12.56).

$$\frac{c \in N \Rightarrow (H_c \rightsquigarrow (now > t \vee \exists d \in N : (c > d) \wedge H_d))}{\exists c \in N : H_c \rightsquigarrow now > t} \tag{12.56}$$

Die SF1-Regel wird benutzt, um den Lattice zu beweisen. Im Folgenden sei $P = H_i$ und $Q = H_{i-1} \vee H_{i-2}$

$$\frac{\begin{array}{l} H_i \wedge [N]_f \;\Rightarrow\; (H_i' \vee (H_{i-1}' \vee H_{i-2}')) \\ H_i \wedge \langle N \wedge Tick(z)\rangle_f \;\Rightarrow\; H_{i-1}' \vee H_{i-2}' \\ H_i \wedge [N]_f \;\Rightarrow\; \Diamond Enabled\ \langle Tick(z)\rangle_f \end{array}}{\Box[N]_f \wedge \mathrm{SF}(Tick(z)) \;\Rightarrow\; (H_i \rightsquigarrow (H_{i-1} \vee H_{i-2}))} \tag{12.57}$$

Die erste Prämisse folgt aus den Konjunkten von Tick $now' = t$, $now < z$, $z' = t + \epsilon$ und $now' \leq now + \epsilon$, damit ergibt sich $z' < z + 2 * \epsilon$. Die zweite Prämisse ist trivial, da ein Schalten von Tick(z) zu $H_{i-1} \vee H_{i-2}$ führt. Für die dritte Prämisse werden die Definitionen (12.58), (12.59) und (12.60) benötigt. Mit (12.58) wird die Menge aller Aktionen definiert, die im gegenwärtig betrachteten Diskretisierungsintervall liegen. Hierbei bezeichnet M_{Ak} die persistente, maximale Wartezeit der Aktion A_k. Mit τ_{Ak} wird die Timervariable der Aktion A_k bezeichnet.

$$ST = \{A_k |\ k \in Aktionenindex :: now + M_{Ak} - \tau_{Ak} < z\} \tag{12.58}$$

$$Enabled\langle Tick(z)\rangle \Leftrightarrow \forall A_i \in ST :: \neg Enabled(A_i) \vee ST = \emptyset \tag{12.59}$$

$$\neg Enabled\langle Tick(z)\rangle \Leftrightarrow \exists A_i \in ST :: Enabled(A_i) \wedge ST \neq \emptyset \tag{12.60}$$

Die dritte Prämisse wird mit der Lattice-Regel unter Verwendung von (12.58), (12.59) und (12.60) bewiesen. Mit der Lattice-Regel ergibt sich:

$$\frac{c \in N \Rightarrow (|ST| = c \rightsquigarrow Enabled\langle Tick(z)\rangle \vee \exists d \in N : (c > d) \wedge |ST| = d)}{\exists c \in N : |ST| = c \rightsquigarrow Enabled\langle Tick(z)\rangle} \tag{12.61}$$

Zum Beweis der Lattice-Regel wird die WF1-Regel herangezogen. Auch hierfür werden Definitionen für P und Q, die in (12.62) und (12.63) angegeben sind, benötigt.

$$P = |ST| = c \ \wedge\ \neg Enabled\langle Tick(z)\rangle \tag{12.62}$$

$$Q = |ST| < c \ \vee \ Enabled\langle Tick(z)\rangle \tag{12.63}$$

Damit ergibt sich (12.64)

$$\frac{\begin{array}{l}|ST| = c \wedge [N]_w \ \Rightarrow \ (P' \vee Q') \\ P \wedge \langle N \wedge A_k\rangle_f \ \Rightarrow \ Q' \\ P \ \Rightarrow \ \Diamond Enabled \ \langle A_k\rangle_f\end{array}}{\Box[N]_f \wedge \mathrm{WF}_f(A_k) \Rightarrow (P \rightsquigarrow Q)} \tag{12.64}$$

Die erste Prämisse ergibt sich aus der Invariante (12.65):

$$\neg Enabled\langle Tick(z)\rangle \Rightarrow \forall \overline{z} > z :: \neg Enabled\langle Tick(\overline{z})\rangle \tag{12.65}$$

Die zweite Prämisse folgt aus der Definition von P und Q. Die dritte Prämisse ergibt sich aus der Definition von ST. Damit ist Zeno-Verhalten für *Tick* ausgeschlossen.

Kapitel 13

Werkzeugunterstützung zur Verifikation von Verfeinerungsmustern

In diesem Kapitel wird die Werkzeugunterstützung zur Modellierung und Verifikation von Verfeinerungsmustern behandelt. Diese ist notwendig, da die manuelle Übersetzung von UML-Diagrammen in cTLA sehr zeitaufwändig ist. Hierfür ist das Werkzeug UML2cTLA entwickelt worden, das die für diese Arbeit relevanten UML-Konstrukte (s. Kapitel 10) in cTLA übersetzt. Eclipse[1] ist ein in der Industrie weit verbreitetes Entwicklungswerkzeug für verschiedene Sprachen. Eclipse-basierte UML-Editoren, die auf einem UML2 Metamodell beruhen, sind mittlerweile sehr weit verbreitet und bilden die Grundlage für eine Transformation von UML nach cTLA mit UML2cTLA. In einer früheren Version ist ein Prototyp für UML2cTLA mit dem UML Werkzeug Rational Rose und Visual Basic Skripten unter Verwendung des Rose Extensibility Interface (REI) entwickelt worden. Bereits existierende Werkzeuge – wie IBM Rational Software Architect, cTc und eTLA+ – erlauben es, eine leistungsfähige und integrierte Umgebung für die Erstellung und Verifikation von Verfeinerungsmustern zu konstruieren.

In Abschnitt 13.1 wird zunächst ein kurzer Überblick über die Gesamtarchitektur gegeben. Das Eclipse-basierte kommerzielle Werkzeug IBM Rational Software Architekt(RSA) zur Modellierung mit UML wird kurz in Abschnitt 13.2 eingeführt. Anschließend wird in Abschnitt 13.3 das Werkzeug UML2cTLA vorgestellt. Der Spezifikationstextberechner cTc wird in Abschnitt 13.4 behandelt. Die Übersetzung von cTLA nach TLA+ unter Verwendung von eTLA und Eclipse wird im letzten Abschnitt betrachtet.

13.1 Überblick über die Gesamtarchitektur

Beim Key Projekt [BHS07a, ABB+00] sind grundlegende Ziele für die Einwicklung von Werkzeugen für die formale Verifikation, die auch im industriellen Kontext verwendet werden sollen, formuliert worden:

1. Tools for formal software specification and verification must be integrated into industrial software engineering procedures.
2. User interfaces of these tools must comply with state-of-the-art software engineering tools.
3. The necessary amount of training in formal methods must be minimized. Moreover, techniques formal software specification and verification must be teachable in a structured manner ...

[1] www.eclipse.org

4. It must be possible to give realistic estimations of the cost of each step in formal software specification and verification depending on the type of software and the degree of formalization.

Die Gesamtarchitektur für die Übersetzung von Verfeinerungsmustern nach cTLA beruht auf der Integration der Werkzeuge RSA, UML2cTLA, cTc, Eclipse, eTLA und TLC mit Eclipse als Integrationsplattform und ist in Abbildung 13.1 angegeben. Gerichtete Kanten geben Informationsflüsse zwischen den Werkzeugen an. Das Werkzeug IBM RSA, das ein kommerziell verwendetes CASE-Tool ist, bildet mit seinen Editoren für die einzelnen UML-Diagramme die Benutzerschnittstelle zur Modellierung von Verfeinerungsmustern mit UML-Diagrammen und stellt die Funktionalität zur Ablage von Modellen bereit. Der Modell-Übersetzer UML2cTLA nimmt eine Übersetzung der UML-Diagramme eines Analysemusters bzw. von Entwurfsmustern in cTLA vor, indem Transformationen auf den relevanten UML Konstrukten durchgeführt werden. Als Ergebnis der Transformationen werden die erzeugten cTLA-Prozesse als Dateien ausgegeben. Weiterhin unterstützt UML2cTLA durch Ausgabe von Zustandsvariablen und deren Typen einen Zustandsfunktions-Editor und einen Regeleditor, die beide unter Verwendung des eTLA-Plugins für Eclipse [GVZ05] realisiert werden. Dies erlaubt es TLA+-Spezifikationen zu erstellen und durch TLC zu verifizieren. Mit dem Zustandsfunktions-Editor lassen sich Zustandsfunktionen erstellen, die Abbildungen zwischen den Zustandsvariablen eines Analysemusters und der Entwurfsmuster beschreiben. Weiterhin unterstützt der Regeleditor die Erstellung von Regeln, die als Gleichungen bzw. Ungleichungen auf der Basis der Parameterbedingungen für die Realzeitanforderungen bzw. die Realzeitanforderungen der Muster formuliert werden. Der Spezifikationstextberechner cTc erlaubt die Umformung eines cTLA-Subsystems mit mehreren cTLA-Prozessen in einen flachen cTLA-Prozess, der in einer Datei bereitgestellt wird. Dieser cTLA-Prozess wird nach einigen Modifikationen mit Eclipse und dem eTLA-Plugin, die den cTLA-Prozess in ein TLA+-Modul überführen, als Eingabe für den Model-Checker TLC verwendet. In den folgenden Abschnitten werden die einzelnen Werkzeuge detaillierter vorgestellt.

13.2 IBM Rational Software Architect

RSA, das auf Eclipse basiert, ist ein von IBM entwickeltes Modellierungswerkzeug für UML mit großem Funktionsumfang. Von besonderem Interesse sind die Diagrammeditoren für Klassen-, Statechart-, Aktivitäts- und Interaktionsdiagramme, die umfangreich alle UML-Konstrukte unterstützen. In Abbildung 13.2 ist gezeigt, wie ein Aktivitätsdiagramm mit RSA erstellt wird. In der rechten Tool-Bar erkennt man die Bedienelemente zur Erstellung, der vom Standard unterstützten UML-Actions, wie sie in Kapitel 3 vorgestellt worden sind. Die erstellten Modelle können in einem Repository abgelegt und in unterschiedlichen Formaten – etwa XMI (XML Metadata Interface) – abgelegt werden. Weiterhin bietet RSA Java Bibliotheken zur Traversierung und Transformation der Modelle an.

Mit dem Eclipse Modelling Projekt liegt ein ähnliches Open-Source Tool der Eclipse Foundation vor, das im Moment aber noch relativ instabil ist, da es sich in der Inkubationsphase befindet. Beide Werkzeuge verwenden die gleichen objektorientierten Frameworks.

13.3 Modell-Übersetzer UML2cTLA

Der Modell-Übersetzer unterstützt den Modellierer bei der Umwandlung von UML-Diagrammen in cTLA-Prozesse und vermeidet die aufwändige und fehleranfällige manuelle Transformation von UML-Diagrammen in cTLA. Der Modellübersetzer (in [MB02] wird ein solches Werkzeug auch Modell-Compiler genannt) greift zur Generierung auf das UML-Repository von RSA zu, um aus den UML-Diagrammen die cTLA-Prozesse zu erzeugen. UML2cTLA ist mit Java entwickelt worden und macht intensiven Gebrauch von den Eclipse Projekten EMF (Eclipse Modelling Framework) und UML2, die es gestatten UML-Modelle als Datenstrukturen in Eclipse bereitzustellen und zu traversieren. Weiterhin können UML-Modelle auf der Basis von XMI als Dateien eingelesen und gespeichert werden. Für die Transformation wird im Model Repository nach Klassen,

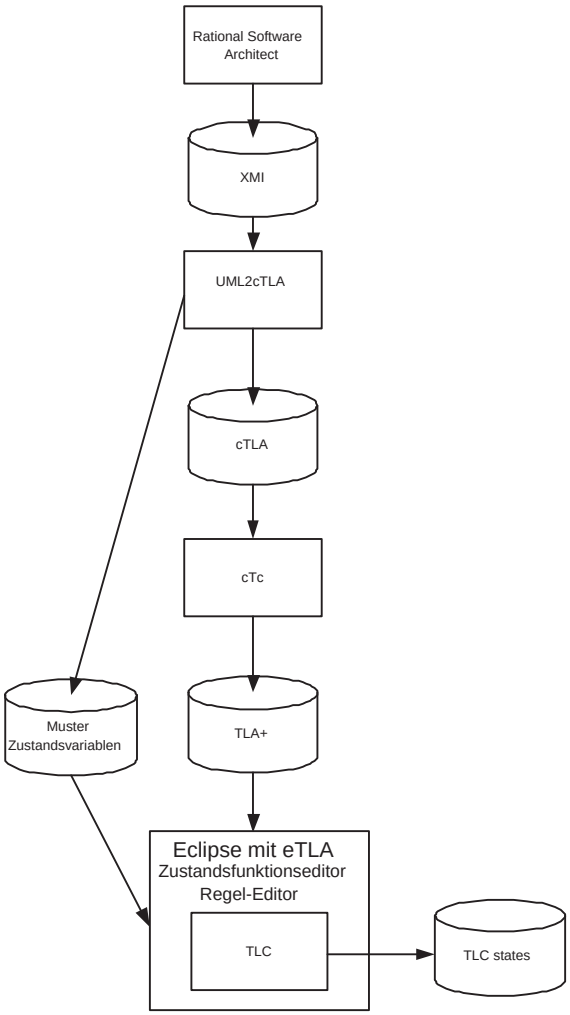


Abbildung 13.1: Die Werkzeuge zur Verifikation

Statechart-Diagrammen, Aktivitätsdiagrammen und Sequenzdiagrammen gesucht. Bei der Transformation muss zwischen einer strukturellen und einer Instantiierungsphase unterschieden werden. In der strukturellen Phase wird jede Klasse mit ihrem Statechart-Diagramm und ihren Aktivitäten übersetzt, wobei jeweils jede Klasse eines Musters einen Ausgangspunkt für die Übersetzung nach cTLA bildet. Dabei wird für jede Klasse ein cTLA-Prozess mit dem Namen der Klasse erzeugt. Weiterhin werden die Klassenattribute in Zustandsvariablen übersetzt. Anschließend wird für jede Klasse das zugehörige Statechart-Diagramm ermittelt. Dieses wird nach Zuständen und Transitionen durchsucht. Dies hat den Zweck die Kontrollzustände und die Aktionen für Transitionen zwischen den Kontrollzuständen für den zugehörigen cTLA-Prozess zu erzeugen. Bei der Übersetzung einer Transition ist eine vorhandene Aktivität zu berücksichtigen. In diesen Aktivitäten werden die Aktionen und Flüsse ermittelt. Für die Flüsse werden entsprechende Zustandsvariablen in den cTLA-Prozess der zugehörigen Klasse aufgenommen. Jede Aktion wird entsprechend ihres Typs und ihrer ein- und ausgehenden Pins – wie in Kapitel beschrieben 10 – in eine cTLA-Aktion übersetzt. Im Rahmen der Instantiierungsphase, deren Ausgangspunkt das Sequenzdiagramm eines Musters ist, wird der cTLA-Systemprozess eines Musters erzeugt. Dazu wird nach dem Sequenzdiagramm gesucht, das die Instantiierung und die Interaktion der Objekte spezifiziert. Für jedes Objekt wird ein Prozess des Prozesstyps der Klasse instantiiert. Die Nachrichten und Aktivierungszonen aus dem Sequenzdiagramm werden ermittelt und verwendet, um in den Systemaktionen die einzelnen Prozessaktionen miteinander zu koppeln.

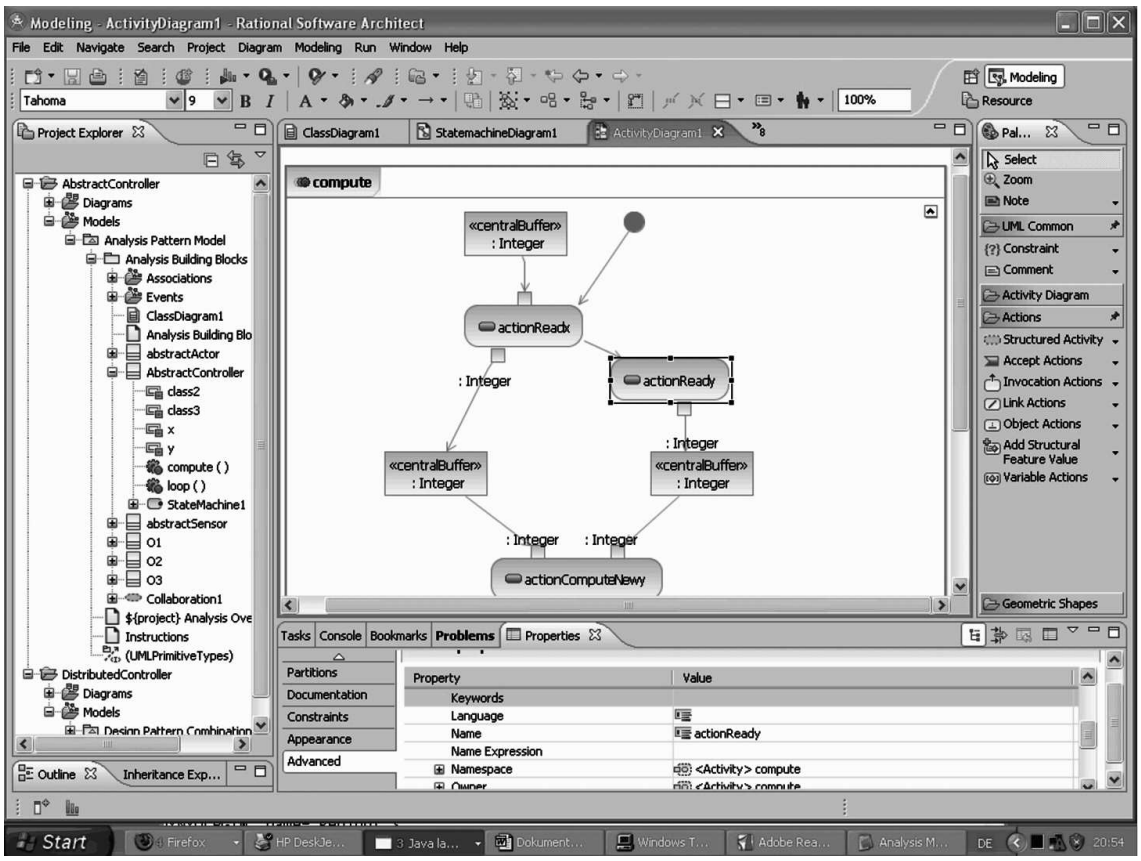


Abbildung 13.2: Ein Editor des IBM Rational Software Architect

Weiterhin kann vor der Übersetzung angegeben werden, ob die Realzeit- und Lebendigkeitseigenschaften einer Aktion berücksichtigt werden sollen. Für jede Aktion kann angegeben werden, ob Einträge in eine durch das Werkzeug generierte *Tick*-Aktion eingefügt und bestimmte cTLA-spezifische Spezifikations-Schlüsselwörter erzeugt werden. In der Dialogbox zur Spezifikation von Lebendigkeits- und Realzeiteigenschaften ist es möglich vorzugeben, ob die generierte cTLA-Aktion Realzeit- oder Lebendigkeitseigenschaften besitzt. Es kann dabei spezifiziert werden, ob für die generierte Aktion persistente oder volatile Wartezeiten vorliegen und ob es sich um maximale bzw. minimale Wartezeiten handelt. Soll die generierte cTLA-Aktion Fairnessanforderungen besitzen, ist es möglich zwischen starker und schwacher Fairness auszuwählen. Außerdem wird für eine generierte cTLA-Aktion spezifiziert, ob bereits Konjunkte in der cTLA-Aktion *Tick* erzeugt werden.

Weiterhin stellt UML2cTLA einen Zustandsfunktions-Editor sowie einen Regeleditor bereit, um die Spezifikation von Verfeinerungsmustern zu erleichtern.

13.4 Spezifikationstextberechner cTc

Der Spezifikationstextberechner wird verwendet, um die vom Modell-Übersetzer UML2cTLA generierten cTLA-Prozesse eines cTLA-Subsystems zu einem cTLA-Prozess für die Verifikation zusammenzufassen. Die wesentliche Funktionalität des Werkzeuges cTc besteht nach [Hey95, Mes02] in der direkten Überführung des Ergebnisses der Anwendung einer cTLA-Spezifikationsoperation in eine flache cTLA-Spezifikation. Dies entbindet den Entwickler von der zeitaufwendigen und ggf. fehlerträchtigen Aufgabe der manuellen Umformung der cTLA-Spezifikationstexte. Das Werkzeug cTc unterstützt cTLA-Prozessoperationen – wie etwa die Prozesskomposition und die Prozessverfeinerung –, die ausführlich in [Mes02] vorgestellt werden. Syntax- und Typprüfungen aller

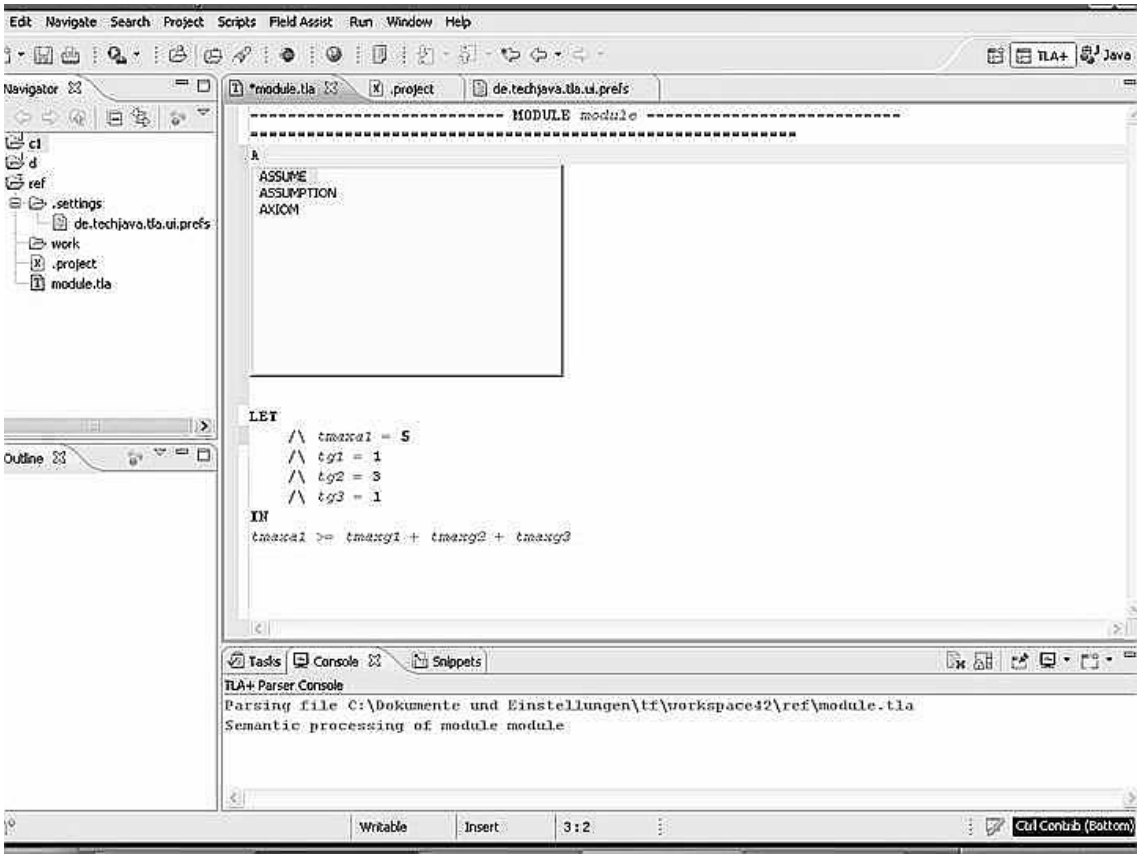


Abbildung 13.3: Der Regeleditor

Elemente (flache Prozesse, Spezifikationsoperationen, Datentypmodule) und die Prüfung der statischen Semantik von Spezifikationsoperationen (etwa Existenz von Verfeinerungvariablen) werden unterstützt. Da das Werkzeug batchorientiert ist, eignet es sich sehr gut als Backendwerkzeug. Das Werkzeug erhält als Eingabe eine oder mehrere Dateien mit cTLA-Subsystemen und cTLA-Prozessen und erzeugt daraus eine Datei mit einem flachen cTLA-Prozess.

13.5 Editoren für Zustandsfunktionen und Regeln

Der Zustandsfunktions-Editor dient zur Angabe von Zustandsfunktionen für Verfeinerungsmuster, in denen Zustandsvariablen der Muster auftreten. Der Zustandsfunktions-Editor erlaubt die Spezifikation von Zustandsfunktionen durch die Angabe von IF-THEN-ELSE- und CASE-Konstrukten, um zwischen den Zustandsvariablen eines Analysemusters und den Zustandsvariablen von Entwurfsmustern Abbildungen anzugeben. Um die Fallunterscheidungen der Zustandsfunktionen anzugeben, werden Boolesche Operatoren und prädikatenlogische Formeln verwendet. Der Editor beruht auf dem eTLA+-Plugin für Eclipse und verwendet die von UML2cTLA bereitgestellten Zustandsvariablen.

Der Regeleditor erlaubt die Spezifikation von zeitbehafteten Regeln eines Verfeinerungsmusters, um maximale und minimale Wartezeiten von Transitionen bzw. Actions anzugeben. Die Regeln können unter Anwendung von Booleschen sowie arithmetischen Operatoren, Relationszeichen und Variablen formuliert werden. Bei der Verwendung eines Verfeinerungsmusters wird TLC als prädikatenlogischer Taschenrechner eingesetzt, um die Regeln auszuwerten. Dies ist durch Angabe des ASSUME Statements [Lam03] möglich, das die Auswertung von Formeln unterstützt. Die Benutzerschnittstelle des Regeleditors ist in Abbildung 13.3 angegeben.

13.6 Übersetzung von cTLA nach TLA+ mit Eclipse und dem TLA-Editor eTLA

Bei der Übersetzung von cTLA nach TLA+ ist zu beachten, dass TLA+ nicht typisiert ist. Deshalb müssen alle Typangaben bei der Definition von Zustandsvariablen und Aktionenparametern entfernt werden. Weiterhin ist zu beachten, dass die Prozessheader von cTLA in TLA+ nicht erlaubt sind. Deshalb müssen diese entfernt und durch das Schlüsselwort **MODULE** ersetzt werden. Ebenso muss das Schlüsselwort **END** am Ende eines cTLA-Prozesses entfernt und stattdessen Bindestriche eingefügt werden. Das Semikolon, das in cTLA am Ende der Definition einer Zustandsvariablen bzw. Aktionendefinition verwendet wird, muss ebenfalls entfernt werden. Schließlich muss eine kanonische Formel in das TLA+-Modul aufgenommen werden. Für diese einfachen syntaktischen Ersetzungen ist das Eclipse Monkey Projekt ausreichend. Dieses Projekt unterstützt die Erstellung und die Ausführung von einfachen Skripten zur Bearbeitung von Spezifikationstexten[2].

[2] http://wiki.eclipse.org/Eclipse_Monkey

Kapitel 14

Anwendung der Verfeinerungsmuster auf Steuerungssoftware

In diesem Kapitel wird die Anwendung der Verfeinerungsmuster, die in Kapitel 9 vorgestellt worden sind, für die Konstruktion der Steuerungssoftware der Beispielanlage aus Kapitel 2 betrachtet. Dazu werden in Abschnitt 14.1 zunächst die Analysemuster betrachtet, die die Anforderungen für Steuerungssoftware reflektieren. In Orientierung an der Struktur des Ebenenmodells aus Abschnitt 2.1 werden dann die verwendeten Verfeinerungsmuster betrachtet. Der Abschnitt 14.1.1 behandelt die Verfeinerungsmuster, die für das Zusammenwirken der Prozessleitebene und der Prozessführungsebene ausgewählt werden. In Abschnitt 14.1.2 wird auf die Verfeinerungsmuster für die Kooperation der Prozessführungsebene und der Prozesssicherungsebene eingegangen. Abschließend wird in Abschnitt 14.1.3 auf die Verfeinerungsmuster für die Kooperation der Prozesssicherungsebene und der Feldebene eingegangen.

14.1 Anwendung der Verfeinerungsmuster auf die Beispielanlage

In Abbildung 14.1 wird das vollständige ASM der Steuerungssoftware mit den Analysemustern für die Steuerungssoftware der Beispielanlage gezeigt. Dadurch werden die funktionalen Anforderungen an die Steuerungssoftware spezifiziert. In der Abbildung wird der Begriff Analysemuster durch die Bezeichnung *AM* abgekürzt. Die Analysemuster sind von *AM1* bis *AM7* nummeriert. Mit dem Analysemuster *AM1* wird ein Bild/Funktionsbaustein-Muster (s. Abschnitt 8.4.7) angegeben. Das Analysemuster *AM2* ist ein Teilanlagensteuerungen/Anlagensteuerungs-Muster, das in Abschnitt 8.4.2 vorgestellt worden ist. Für das Analysemuster *AM3* wird das Gruppensteuerungen/Teilanlagensteuerungs-Muster aus Abschnitt 8.4.3 drei Mal angewendet, da der Teilanlagensteuerung zwei Gruppensteuerungen und ein Regler untergeordnet sind. Das Analysemuster *AM4* ist ein Einzelgerätesteuerungen/Gruppensteuerungs-Muster, das für die Steuerung des Einfüllvorgangs verwendet wird. Auch bei dem Analysemuster *AM5* wird ein Einzelgerätesteuerungen/Gruppensteuerungs-Muster verwendet, um das Abpumpen der Lösung zu steuern. Um die Membranpumpe vor Beschädigungen zu bewahren, findet für das Analysemuster *AM6* das Funktionsbaustein/Verriegelungs-Muster Verwendung, das bereits aus Abschnitt 8.4.5 bekannt ist. An dieses Muster werden Fehlertoleranzanforderungen gestellt. Das Analysemuster *AM7* entspricht dem wohlbekanntem Reglermuster aus Abschnitt 8.4.6, das verwendet wird, um die Temperatur der Gelatine-Lösung zu stabilisieren. In den folgenden drei Abschnitten wird für jede Ebene der Steuerungssoftware erläutert, wie auf der Basis der nicht-funktionalen Anforderungen an die Analysemuster Verfeinerungsmuster ausgewählt werden, um zu einem korrekten Entwurf der Steuerungssoftware zu gelangen. An alle Analysemuster bestehen nicht-funktionale Anforderungen bzgl. der Verteilung und der Realzeitfähigkeit.

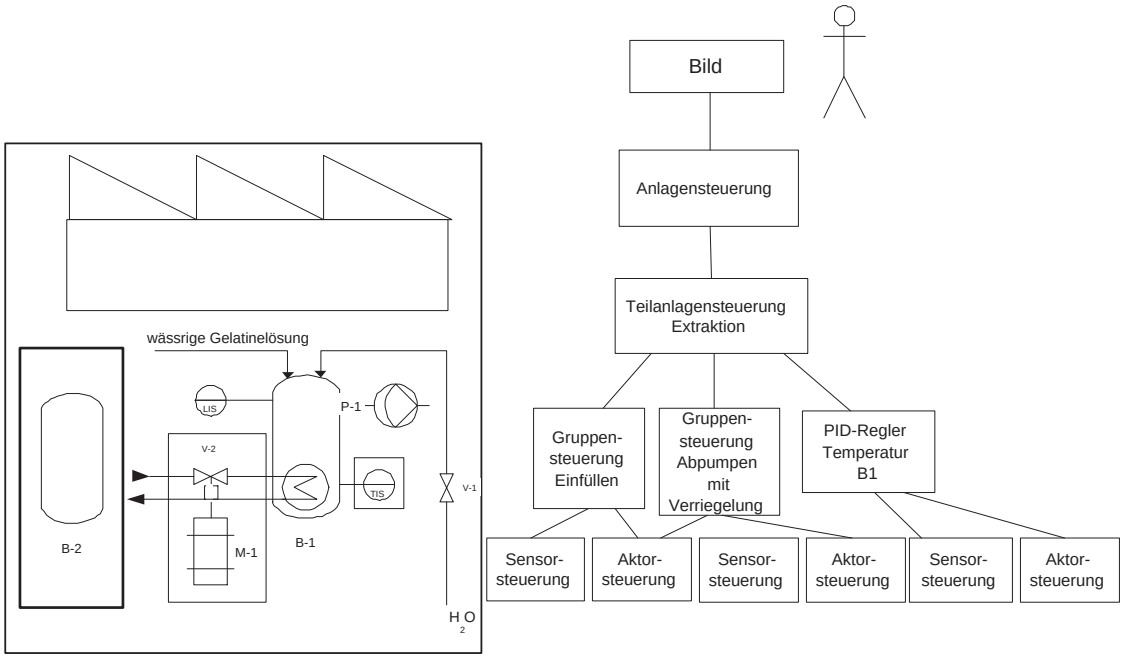


Abbildung 14.1: Die Beispielanlage mit Steuerungssoftware

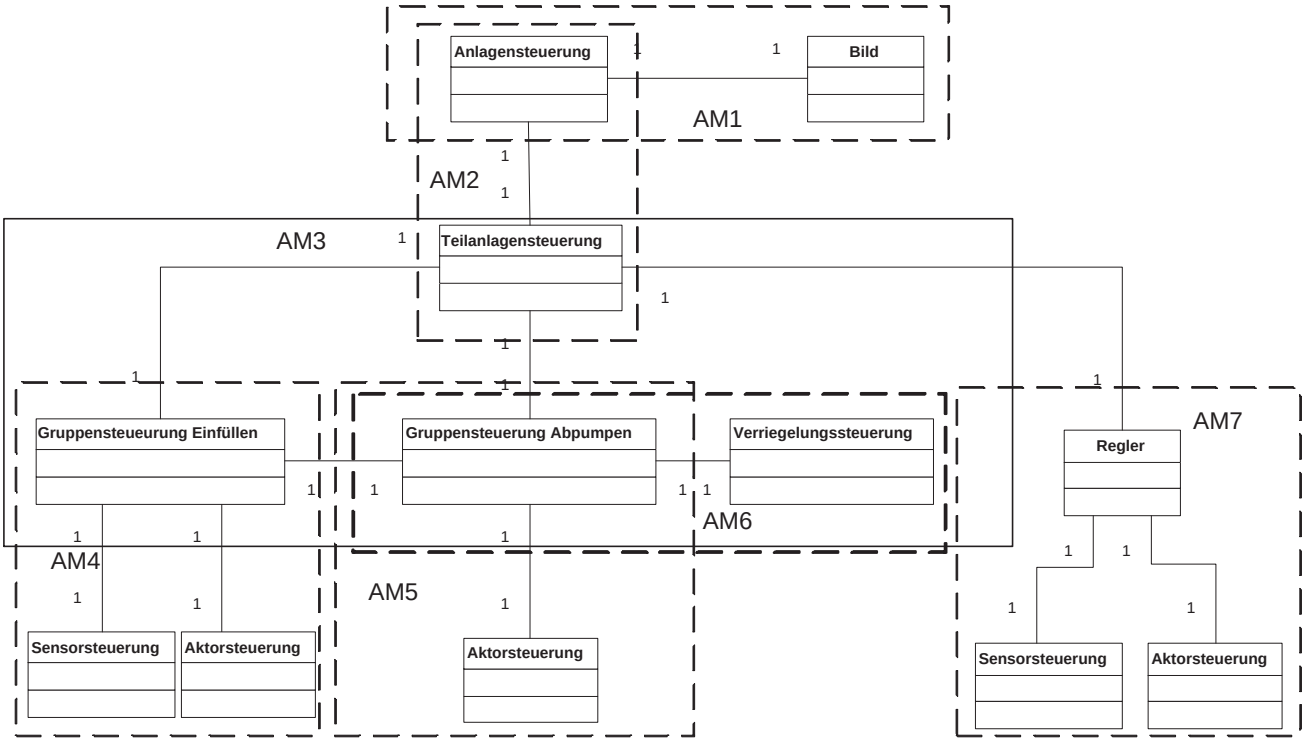


Abbildung 14.2: Das ASM der Beispielanlage mit den Analysemustern

14.1.1 Anwendung der Verfeinerungsmuster in der Prozessführungsebene

Zunächst wird das verteilte Bild/Funktionsbaustein-Muster zur Verfeinerung des Analysemusters verwendet, um sicherzustellen, dass relevante Informationen zeitnah beim Bediener der Anlage eintreffen. Dies ist darauf zurückzuführen, dass Verteilungs- und Realzeitanforderungen bestehen. Durch die Anwendung dieses Verfeinerungsmusters ergibt sich das in der Abbildung 14.3 angegebene Design. Um die Realzeitanforderungen für das Bild und den Funktionsbaustein, der in diesem Fall eine Teilanlagensteuerung repräsentiert, zu erfüllen, werden zwei sporadische Tasks verwendet. Da Transaktionssicherheit gefordert ist, werden entsprechend dem Verfeinerungsmuster ein Transaktionskoordinator und zwei Wrapper eingesetzt. Realzeitbedingungen, die sich aus den Anforderungen ergeben, sind zu prüfen, um sicherzustellen, dass das Verfeinerungsmuster anwendbar ist.

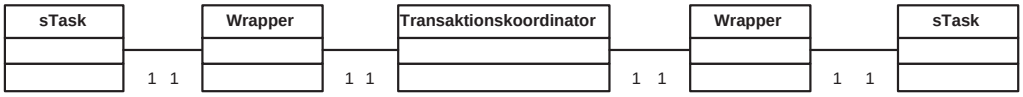


Abbildung 14.3: Das Design des Bild/Funktionsbaustein-Musters

14.1.2 Anwendung der Verfeinerungsmuster für die Kooperation der Prozessführungsebene und der Prozessstabilisierungsebene

An das Analysemuster *AM2* aus der Abbildung 14.2, bei dem eine Anlagensteuerung mit einer untergeordneten Teilanlagensteuerung kooperiert, werden Realzeit- und Verteilungsanforderungen gestellt. Daher wird das verteilte Teilanlagensteuerungen/Anlagensteuerungs-Musters zur Verfeinerung von *AM2* herangezogen. In der Abbildung 14.4 wird die Entwurfsmuster-Kombination angegeben, die sich durch die Anwendung dieses Verfeinerungsmusters ergibt. Für jede Steuerung wird jeweils eine sporadische Task eingesetzt. Um die Verteilungsanforderungen zu gewährleisten, wird das Proxy-Muster verwendet. Selbstverständlich sind die Realzeitbedingungen gemäß den Anforderungen zu prüfen. Es besteht eine Überlappung bei der sporadischen Task für die Anlagensteuerung, wie in den Abbildungen 14.3 und 14.4 zu erkennen ist.

Bei der korrekten Umsetzung der Steuerungssoftware für die Teilanlagensteuerung mit ihren Gruppensteuerungen, die mit *AM3* in Abbildung 14.2 bezeichnet wird, hilft das verteilte Gruppensteuerungen/Teilanlagensteuerungs-Muster, an das nach Abschnitt 2.5 wiederum Realzeit- und Verteilungsanforderungen bestehen, weil sämtliche Steuerungen räumlich voneinander entfernt sind. Dabei kooperiert die Teilanlagensteuerung mit zwei untergeordneten Gruppensteuerungen und dem Regler. In der Abbildung 14.5 wird die Entwurfsmuster-Kombination gezeigt, die aus vier Tasks und drei Proxy Klassen besteht. Um diese Entwurfsmuster-Kombination zu erhalten, wird das Verfeinerungsmuster wiederholt angewendet. Wegen der hohen Realzeitanforderungen an die Gruppensteuerungen, die prozessnäher sind als die bisher behandelten Steuerungen der höheren Ebenen, werden periodische Tasks verwendet, die in fest vorgegebenen Zeitintervallen Verarbeitungen durchführen. Da die Gruppensteuerungen räumlich entfernt von ihrer Teilanlagensteuerung positioniert sind, wird mehrfach das Proxy-Muster herangezogen, das für die Kommunikation zwischen der Teilanlagensteuerung und den Gruppensteuerungen gebraucht wird. Natürlich ist die Erfüllung der Realzeiteigenschaften durch Prüfung von Bedingungen nachzuweisen. Eine Überlappung zwischen den Ebenen ergibt sich für die sporadische Task, die für die Teilanlagensteuerung

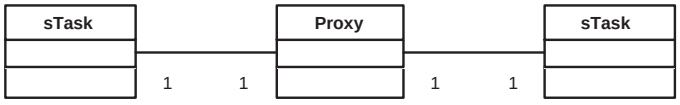


Abbildung 14.4: Das Design der Anlagensteuerung mit ihrer Teilanlagensteuerung

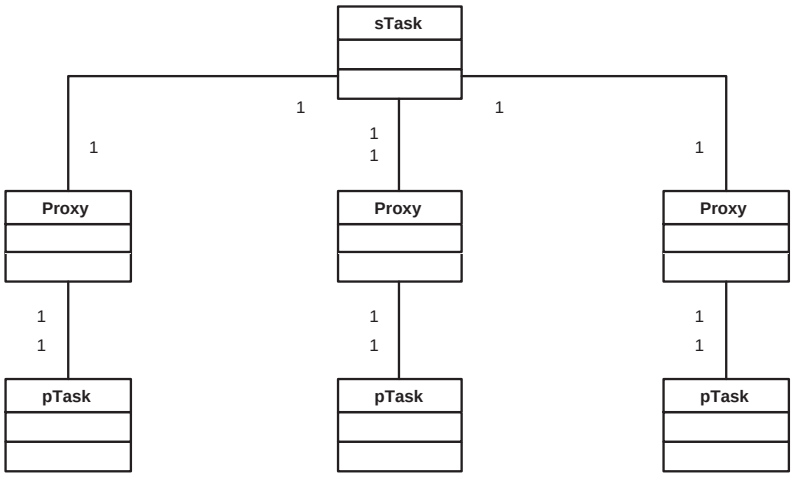


Abbildung 14.5: Das Design der Teilananlagensteuerung mit ihren Gruppensteuerungen

eingeführt worden ist. Dies erkennt man anhand der Abbildungen 14.4 und 14.5.

14.1.3 Anwendung der Verfeinerungsmuster für die Kooperation der Prozessstabilisierungsebene und der Feldebene

In Abschnitt 2.5 sind Anforderungen bzgl. der Verteilung und der Fehlertoleranz an die Gruppensteuerungen, den Regler und die Einzelgerätesteuerungen dieser beiden Ebenen formuliert worden. Die Gruppensteuerung zum Einfüllen ist von der Sensorsteuerung S_1 räumlich entfernt, da sich der Sensor direkt am Behälter *B1* befindet. Außerdem befinden sich die Gruppensteueuerung zum Abpumpen und ihre Aktorsteuerung A_1 an verschiedenen Positionen. Auch die Sensorsteuerung S_2 und die Aktorsteuerung A_2 sind nicht direkt bei dem Regler für die Temperatur positioniert, sodass für die Kommunikation eine räumliche Distanz überbrückt werden muss.

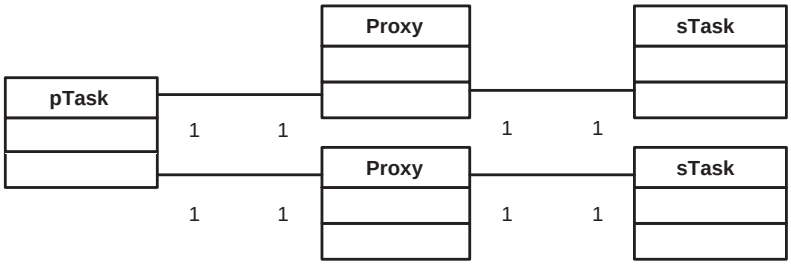


Abbildung 14.6: Das Design für die Einzelgerätesteuerung und die Gruppensteuerung zur Behälterbefüllung

Zur korrekten Realisierung des Zusammenwirkens von Einzelgeräte- und Gruppensteuerungen findet das verteilte Einzelgerätesteuerungen/Gruppensteuerungs-Muster Verfeinerungsmuster zweimalige Verwendung zur Verfeinerung der Analysemuster *AM4* und *AM5*. Die erste Anwendung dieses Verfeinerungsmusters betrifft die Gruppensteuerung für das Befüllen von *B1* und die Einzelgerätesteuerung für das zugehörige Ventil. Die zugehörige Entwurfsmuster-Kombination ist in Abbildung 14.6 gezeigt. Man erkennt, dass zwei sporadische und eine periodische Task verwendet werden. Zwei *Proxy*-Objekte dienen der Kommunikation der Steuerungen.

Weiterhin wird dasselbe Verfeinerungsmuster für die Gruppensteuerung zum Abpumpen der Gelatine und die Einzelgerätesteuerung für das Auslassventil – also *AM5* – verwendet. In Abbildung 14.7 ist die zugehörige Entwurfsmuster-Kombination angegeben. Für die Gruppensteuerung wird eine periodische Task und für die Einzelgerätesteuerung eine sporadische Task verwendet, um

die Realzeitanforderungen zu erfüllen. Zur Unterstützung der Kommunikation dient wiederum das Proxy Muster.

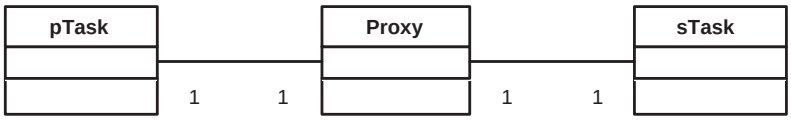


Abbildung 14.7: Das Design für die Einzelgerätesteuerung und die Gruppensteuerung zum Abpumpen

14.1.4 Anwendung der Verfeinerungsmuster innerhalb der Prozesssicherungsebene

Um sicherzustellen, dass die Membran der Pumpe keinesfalls beschädigt wird, muss die Verriegelungssteuerung, die mit der Gruppensteuerung zum Abpumpen zusammenwirkt, fehlertolerant ausgelegt werden (s. *AM6*). Das bedeutet, dass ein Ausfall der Verriegelungssteuerung mit hoher Wahrscheinlichkeit verhindert wird. Daher wird das Verfeinerungsmuster fehlertolerante Verriegelung mit einem Watchdog Muster verwendet (alternativ ist natürlich ein Recoverable Distributed Observer Muster möglich). In Abbildung 14.8 ist der zugehörige Entwurf, der einen Watchdog mit einer eigenen periodischen Task verwendet, angegeben. Die Task des Watchdogs prüft in periodischen Zeitabständen, ob die Task für die Verriegelungssteuerung noch aktiv ist. Prüfungen für die Realzeitbedingungen haben zum Gegenstand, dass eine Reaktion der Verriegelungssteuerung schnell genug erfolgen muss, damit die Zerstörung der Membranpumpe ausgeschlossen wird. Den Ausfall der Verriegelungstask muss der Watchdog erkennen und innerhalb der vorgegebenen Reaktionszeit einen Neustart vornehmen.

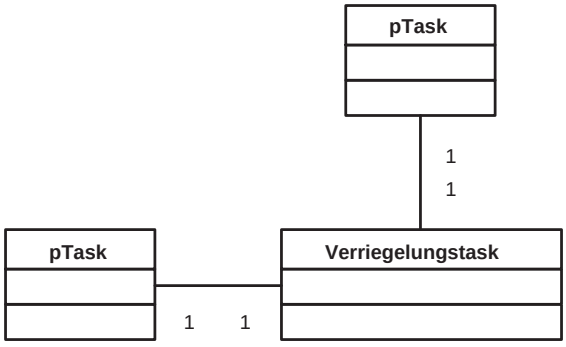


Abbildung 14.8: Das Design der Verriegelungssteuerung

Das Verfeinerungsmuster des verteilten Reglers wird zur korrekten Realisierung des Analysemusters *AM7* verwendet, das die Temperatur des Behälters *B1* stabilisiert. Das zugehörige Design des verteilten Reglers ist bereits ausführlich in den vorherigen Abschnitten vorgestellt worden.

Abschließend wird festgestellt, dass auch in dieser Ebene Überlappungen zu den Verfeinerungsmustern der Steuerungen der hierarchisch höher angeordneten Steuerungen existieren. Die periodischen Tasks aus den Abbildungen 14.6, 14.7 und 14.8 und der periodischen Task des Reglers entsprechen den in Abbildung 14.5 angegebenen periodischen Tasks.

Kapitel 15

Ausblick

In der Arbeit sind Verfeinerungsmuster für objektorientierte Steuerungssoftware vorgestellt worden. Dabei sind alle Aspekte von der Modellierung mit UML, der Übersetzung in cTLA bis zur Verifikation und dem Model-Checking der formalen Modelle untersucht worden. Ein Schwerpunkt lag auf dem Nachweis der Korrektheit von Verfeinerungsmustern auch unter Berücksichtigung von Realzeitanforderungen.

Im Bereich der Steuerungssoftware liegt mit OPC UA ein neuer Standard vor, der es ermöglicht verteilte Architekturen für Steuerungssoftware zu realisieren. Hierfür existieren seit kurzem einige frei verfügbare Implementierungen des OPC-UA-Stacks auf der Basis von Java bzw. C#. Die Informationsmodelle der OPC UA [MLD09] und das OPC-UA-Informationsmodell für IEC 61131-3 [Tec09] bilden eine interessante Ausgangsposition für die Anwendung bzw. Weiterentwicklung der Verfeinerungsmuster dieser Arbeit. Natürlich ist dabei auch die Verwendbarkeit der Analyse- und Entwurfsmuster zu betrachten. Ebenso sind die sog. Programme aus OPC UA, die mit Statechart-Diagrammen erstellt werden, für weitere Forschungen im Bereich der Steuerungssoftware von Interesse. Zusätzlich ist der Bereich der Alarme und Bedingungen (Alarms and Conditions) bedeutsam, da dort sicherheitskritische Funktionalität verifiziert werden kann. Für IEC 61131-3 werden Programme aus OPC UA erst für spätere Versionen des Standards (gegenwärtig liegt Version 0.09 vor) erwartet bzw. in Aussicht gestellt.

Sicherlich ist es sinnvoll, auch andere Domänen als den Bereich der Steuerungssoftware nach Verfeinerungsmustern zu untersuchen. Damit würde das Konzept durch die Anforderungen anderer Fachgebiete konsolidiert bzw. weiterentwickelt werden. Z. B. liegt für das ebenfalls technisch orientierte Gebiet der Energieinformatik [WSA07] eine Domäne vor, die sich mit dem Einsatz der Informationstechnik in der Energiewirtschaft auseinandersetzt. Ein UML-Profil names CIM (Common Information Infrastructure) [UG07] ist bereits standardisiert (IEC-61970) worden.

Lamport hat auch die Verwendung von TLC zur Verifikation von Realzeiteigenschaften vorgeschlagen [Lam05]. TLC ist durchaus geeignet, um Realzeiteigenschaften zu verifizieren. Von Lamport durchgeführte Messungen zeigen, dass TLC mit Model-Checkern für Realzeitanforderungen, wie etwa Uppaal, konkurrieren kann. An Stelle der in dieser Arbeit verwendeten Verifikation der Realzeiteigenschaften von Verfeinerungsmustern mittels Handbeweisen könnte die Verifikation mit dem Model-Checker TLC automatisiert werden, indem etwa bestimmte Schablonen für das Model-Checking dieser Eigenschaften bereitgestellt werden.

Natürlich sind alle Forschungsansätze, die die maschinelle Verifikation von Verfeinerungsmustern unterstützen können, von großer Bedeutung. Hierbei sind alle Ansätze, die zur Beschleunigung der Verifikation führen, besonders interessant. In den letzten Jahren haben sog. SAT Solver Interesse hervorgerufen. SAT Solver sind in der Lage, komplexe, Boolesche Gleichungen, die aus sehr vielen Booleschen Variablen bestehen und in konjunktiver Normalform (CNF) spezifiziert sind, auf ihre Erfüllbarkeit zu prüfen. Auf diesem Gebiet sind nach [MP05, Mar09] beachtliche Verbesserungen erzielt worden. Die Übersetzung von temporallogischen Konstrukten, die für cTLA-Spezifikationen eines Verfeinerungsmusters verwendet werden, in Boolesche Formeln ist hierfür notwendig. Außerdem erlangt auch die Anwendung von Mehrkernprozessoren für das Model-Checking zuneh-

mende Bedeutung [Hol08, VBBB09]. TLC unterstützt bereits Java Multi-Threading. Allerdings werden Locks in Threads eingesetzt, die zu Blockaden führen. Lock-free Algorithmen bieten einen interessanten Ausweg. Da TLC mit Java realisiert worden ist, sind auch Neuerungen der Java Technologie, wie das ForkJoin Framework aus Java 7 [DME09] von Interesse.

Auch Erweiterungen der Spezifikationssprache cTLA können für die Weiterentwicklung der Konzepte dieser Arbeit von Bedeutung sein. Die cTLA 2003 Version unterstützt die Vererbung von Prozesstypen mit dem Schlüsselwort **EXTENDS** [RK03]. Für Verfeinerungsmuster, die objektorientierte Vererbung verwenden, entsteht dadurch eine interessante Erweiterungsmöglichkeit. Auch die Anwendung der in [Mes02] vorgestellten Operatoren zur Verfeinerung von Prozessen ist von Interesse.

Anhang A

cTLA-Spezifikationen der Analysemusterkombination

A.1 cTLA-Spezifikationen des Prozesses *abstractController*

```
PROCESS abstractController

IMPORT
      Sequence;

CONSTANT
 k : Integer;

VARIABLES
    x,y : Real; ! y für den Stellwert und x für den Istwert
    qu : queue; ! Zustand der Ereigniswarteschlange
    sync : Boolean; ! Synchronisationsvariable für nebenläufigen Zugriff
    ctrlStartActReadx, ctrlActReadxActReady : Boolean; ! ZV für eingeh. KF
    state : {"init", "stopped", "wait", "waitSensor", "getReturnReceived", "getReturnDequeued",
    "valuesComputed", "waitActuator","setReturnReceived"}; ! Kontrollzustand
    O1_Exists, existsO2, existsO3 : Boolean; ! Existenz der Objektflüsse O1, O2 und O3
    valueO1, valueO2, valueO3 : Real; ! Übersetzung der Objektknoten O1, O2, O3 zur Wertablage
INIT ≜ ... s. Text

ACTIONS

! Die Übersetzung der Transitionen

loop ≜ ...

waitSensorToValuesComputed ≜ ... ! Transition wait waitSensor→ ValuesComputed Aktion

wait waitSensor → ValuesComputed Aktion s. waitToStopped ≜ !

!Transition wait → stopped Aktion s. Text stoppedToWait ≜  !

Transition stopped → wait Aktion s. Text
    ∧ state = ''stopped''
    ∧ state′ = ''init''
    ∧ UNCHANGED ⟨x, y, qu, sync, ctrlStartActReadx, ctrlActReadxActReady⟩
    ∧ UNCHANGED ⟨ existsO1, existsO2, existsO3, valueO1, valueO2, valueO3⟩;

! Die Übersetzung der Actions

actionReadx(currentEventOccurrence : String) ≜ ... ! Aktion s.Text

actionWritey(currentEventOccurrence : String) ≜ ... ! Aktion s. Text

actionComputeNewy(currentEventOccurence : String) ≜ ... ! Aktion s. Text
```

```
actionGetValueCall(currentEventOccurence : String) ≜ ... ! Aktion s. Text

actionReady(currentEventOccurence : String) ≜ ! Übersetzung der Action Ready
    ∧ ¬sync
    ∧ ctrlStartActReadx = TRUE ! Prüfung der eingehenden ZV
    ∧ currentEventOccurence = "O1.actionReadx"
    ∧ value01′ = x ∧ ! Setzen der ZV eines ausgehenden OF
    ∧ ctrlStartActReadx′ = FALSE
    ∧ exists01′ = TRUE ! Setzen der ausgehenden ZV
    ∧ ctrlActReadxActReady′ = TRUE ! Setzen der ZV des ausgehenden KF
    ∧ UNCHANGED ⟨ x,y, qu, sync, state, exists02, exists03, value02, value03⟩;

computeNewy ≜ [p1 ∈ Real, p2 ∈ Real ↦ k*p1]

actionSetValueCall(val : Real; currentEventOccurence : String) = ! Action setValueCall
    ∧ state = ''valuesComputed''
    ∧ state′ = ''waitActuator''
    ∧ currentEventOccurence = ''O1.actionSetValueCall''
    ∧ val = x
    ∧ sync′ = TRUE
    ∧ UNCHANGED ⟨qu, x, y, sync, ctrlStartActReadx, ctrlActReadxActReady⟩
    ∧ UNCHANGED ⟨exists01, exists02, exists03, value01, value02, value03⟩;

enqueue(val : Real) ≜ ...

dequeue ≜ ...

END
```

Anhang B

TLA-Spezifikationen der Analysemusterkombination

B.1 TLC-Spezifikation des Moduls *abstractController*

```
MODULE abstractContr

EXTENDS Naturals, Sequences, TLC, Reals

VARIABLES ostate,
          oqu,
          ox,
          oy,
          osync,
          oCtrlStartActReadx,
          oCtrlActReadxActReady,
          oValue01,
          oValue02,
          oValue03,
          oExists01,
          oExists02,
          oExists03

Init ≜ ∧ ostate = "init"
       ∧ ox = 0
       ∧ oy = 0
       ∧ osync = FALSE
       ∧ oqu = ≪ ≫
       ∧ oCtrlActReadxActReady = FALSE
       ∧ oCtrlStartActReadx = FALSE
       ∧ oValue01 = 0
       ∧ oExists01 = FALSE
       ∧ oValue02 = 0
       ∧ oExists02 = FALSE
       ∧ oValue03 = 0
       ∧ oExists03  = FALSE


stop ≜
      ∧ ostate = "init"
      ∧ ostate′ = "stopped"
      ∧ UNCHANGED ⟨ox, oy, oqu, osync⟩
      ∧ UNCHANGED ⟨oCtrlActReadxActReady, oCtrlStartActReadx⟩
```

```
      ∧ UNCHANGED ⟨oExists01, oExists02, oExists03⟩
      ∧ UNCHANGED ⟨oValue01, oValue02, oValue03⟩

restart ≜
      ∧ ostate = "stopped"
      ∧ ostate′ = "init"
      ∧ UNCHANGED ⟨ox, oy, oqu, osync⟩
      ∧ UNCHANGED ⟨oCtrlActReadxActReady, oCtrlStartActReadx⟩
      ∧ UNCHANGED ⟨oExists01, oExists02, oExists03⟩
      ∧ UNCHANGED ⟨oValue01, oValue02, oValue03⟩
loop ≜
      ∧ ostate = "init"
      ∧ ostate′ = "wait"
      ∧ UNCHANGED ⟨ox, oy, oqu, osync⟩
      ∧ UNCHANGED ⟨oCtrlActReadxActReady, oCtrlStartActReadx⟩
      ∧ UNCHANGED ⟨oExists01, oExists02, oExists03⟩
      ∧ UNCHANGED ⟨oValue01, oValue02, oValue03⟩

getValueCall ≜
    ∧ ostate = "wait"
    ∧ ostate′ = "waitSensor"
    ∧ osync′ = TRUE
    ∧ UNCHANGED ⟨ox, oy, oqu rangle:
    ∧ UNCHANGED ⟨oCtrlActReadxActReady, oCtrlStartActReadx⟩
    ∧ UNCHANGED ⟨oExists01, oExists02, oExists03⟩
    ∧ UNCHANGED ⟨oValue01, oValue02, oValue03⟩

compute ≜
    ∧ ostate = "valueDequeued"
    ∧ oCtrlStartActReadx = TRUE
    ∧ UNCHANGED ⟨ox, oqu, osync⟩
    ∧ UNCHANGED ⟨oCtrlActReadxActReady, oCtrlStartActReadx⟩
    ∧ UNCHANGED ⟨oExists01, oExists02, oExists03⟩
    ∧ UNCHANGED ⟨oValue01, oValue02, oValue03⟩

actionReadx(currentEventOccurence) ≜
    ∧ ¬osync
    ∧ oCtrlStartActReadx = TRUE
    ∧ ocurrentEventOccurence = "01.actionReadx"
    ∧ oValue01′ = ox
    ∧ oCtrlStartActReadx′ = FALSE
    ∧ oExists01′ = TRUE
    ∧ oCtrlActReadxActReady′ = TRUE
    ∧ UNCHANGED ⟨ox, oqu, osync⟩
    ∧ UNCHANGED ⟨oCtrlActReadxActReady⟩
    ∧ UNCHANGED ⟨oExists01, oExists02, oExists03⟩
    ∧ UNCHANGED ⟨oValue01, oValue02, oValue03⟩

actionReady(currentEventOccurence) ≜
    ∧ ¬osync
    ∧ oCtrlStartActReadx = TRUE
    ∧ ocurrentEventOccurence = "01.actionReady"
    ∧ oValue01′ = ox
    ∧ oCtrlStartActReadx′ = FALSE
    ∧ oExists01′ = TRUE
    ∧ oCtrlActReadxActReady′ = TRUE
    ∧ UNCHANGED ⟨ox, oy, oqu, osync⟩
    ∧ UNCHANGED ⟨oCtrlStartActReadx⟩
    ∧ UNCHANGED ⟨oExists01, oExists02, oExists03⟩
    ∧ UNCHANGED ⟨oValue01, oValue02, oValue03⟩
```

```
actionWritey(currentEventOccurrence)
    ∧ ¬osync
    ∧ oExistsO3 = TRUE
    ∧ ocurrentEventOccurrence = "O1.actionWritey"
    ∧ oExistsO3′ = FALSE
    ∧ oy′ = oValueO3
    ∧ ostate′ = "valuesComputed"
    ∧ UNCHANGED ⟨ox, oy, oqu, osync⟩
    ∧ UNCHANGED ⟨oExistsO1, oExistsO2⟩
    ∧ UNCHANGED ⟨oValueO1, oValueO2, oValueO3⟩
    ∧ UNCHANGED ⟨oCtrlStartActReadx, oCtrlActReadxActReady⟩

actionSetValueCall(valueO1_Pin1, currentEventOccurence) ≜
    ∧ ¬osync
    ∧ ostate = "valuesComputed"
    ∧ oValueO1_Pin1 = x
    ∧ currentEventOccurence = "O1.actionSetValueCall"
    ∧ oExistsO1′ = FALSE
    ∧ osync′ = TRUE
    ∧ ostate′ = "waitActuator"
    ∧ UNCHANGED ⟨ox, oy, oqu, osync ⟩
    ∧ UNCHANGED ⟨oCtrlStartActReadx, oCtrlActReadxActReady⟩
    ∧ UNCHANGED ⟨oExistsO1, oExistsO2⟩
    ∧ UNCHANGED ⟨oValueO1, oValueO2, oValueO3⟩

computeNewy ≜ [p1 ∈ Real, p2 ∈ Real ↦ p1]

actionComputeNewy(currentEventOccurence) ≜   !!actionComputeNewy ≜
    ∧ ¬osync
    ∧ oExistsO1 = TRUE
    ∧ oExistsO2 = TRUE
    ∧ currentEventOccurence = "O1.actionComputeNewy"
    ∧ oValueO3′ = computeNewy[oValueO1, oValueO2]
    ∧ oExistsO1′ = FALSE
    ∧ oExistsO2′ = FALSE
    ∧ oExistsO3′ = TRUE
    ∧ UNCHANGED ⟨oValueO1, oValueO2, osync⟩
    ∧ UNCHANGED ⟨oCtrlActReadxActReady, oCtrlStartActReadx⟩

setValueCall(value) ≜
      ∧ ostate = "valuesComputed"
      ∧ ostate′ = "waitActuator"
      ∧ osync′ = TRUE
      ∧ UNCHANGED ⟨ox, oy, oqu ⟩
      ∧ UNCHANGED ⟨oCtrlActReadxActReady, oCtrlStartActReadx⟩
      ∧ UNCHANGED ⟨oExistsO1, oExistsO2, oExistsO3⟩
      ∧ UNCHANGED ⟨oValueO1, oValueO2, oValueO3⟩

enqueue(value) ≜
       ∧ ostate = "waitSensor" ∨ ostate = "waitActuator"
       ∧ ostate′ = IF ostate = "waitSensor"
                   THEN "getEnqueued"
                   ELSE IF ostate = "waitActuator"
                   THEN "returnEnqueued"
                   ELSE ostate
       ∧ osync′ = FALSE
       ∧ oqu′ = IF ostate = "waitSensor"
                THEN Append(oqu, value)
                ELSE oqu
       ∧ UNCHANGED ⟨ox, oy ⟩
       ∧ UNCHANGED ⟨oCtrlActReadxActReady, oCtrlStartActReadx⟩
```

```
       ∧ UNCHANGED ⟨oExists01, oExists02, oExists03⟩
       ∧ UNCHANGED ⟨oValue01, oValue02, oValue03⟩

dequeue ≜
       ∧ ostate = "getEnqueued" ∨ ostate = "returnEnqueued"
       ∧ ostate′ = IF ostate = "getEnqueued"
                   THEN "valueDequeued"
                   ELSE "init"
       ∧ oqu′ = Tail(oqu)
       ∧ ox′ = Head(oqu).val
       ∧ UNCHANGED ⟨oy, osync ⟩
       ∧ UNCHANGED ⟨oCtrlActReadxActReady, oCtrlStartActReadx⟩
       ∧ UNCHANGED ⟨oExists01, oExists02, oExists03⟩
       ∧ UNCHANGED ⟨oValue01, oValue02, oValue03⟩


AFair ≜ ∃ value ∈ Data : ∃ proc ∈ Operation :
                              ∧ WF_vars(loop)
                              ∧ WF_vars(dequeue)
                              ∧ WF_vars(stop)
                              ∧ WF_vars(compute)
                              ∧ WF_vars(restart)
                              ∧ WF_vars(getValueCall)
                              ∧ WF_vars(setValueCall([op ↦ proc, val ↦ value]))
                              ∧ WF_vars(enqueue([op ↦ proc, val ↦ value]))

Next ≜ ∃ value ∈ Data : ∃ proc ∈ Operation :
                              ∨ loop
                              ∨ dequeue
                              ∨ stop
                              ∨ compute
                              ∨ restart
                              ∨ getValueCall
                              ∨ setValueCall([op ↦ proc, val ↦ value])
                              ∨ enqueue([op ↦ proc, val ↦ value])

vars ≜ ≪ostate, oqu, ox, oy, osync, oCtrlActReadxActReady,
         oValue01, oValue02, oExists01, oExists02, oExists02,
         oExists03, oValue03, oCtrlStartActReadx≫

Spec ≜ Init ∧ □[Next]_vars
```

B.2 TLC-Spezifikation des Moduls *abstractSensor*

```
MODULE abstractSensor
EXTENDS Naturals, TLC, Sequences

VARIABLES
    x,
    state,
    qu

CONSTANTS Data

Init ≜ ∧ state = "init"
       ∧ x = 0
       ∧ qu  = ≪ ≫

CallEnqueue(d) ≜ ∧ state = "init"
```

```
                ∧ state′ = "enqueued"
                ∧ qu′ = Append( qu, d)
                ∧ UNCHANGED x

Dequeue ≜  ∧ state = "enqueued"
           ∧ state′ = "dequeued"
           ∧ x′ = Head(qu)
           ∧ qu′ = Tail(qu)

Process ≜  ∧ state = "dequeued"
           ∧ state′ = "processed"
           ∧ UNCHANGED ⟨x, qu⟩

EnqueueError ≜  ∧ state = "enqueued"
                ∧ state′ = "Error"
                ∧ UNCHANGED ⟨x, qu⟩

DequeueError ≜  ∧ state = "dequeued"
                ∧ state′ = "Error"
                ∧ UNCHANGED ⟨x, qu⟩

ProcessError ≜  ∧ state = "processed"
                ∧ state′ = "Error"
                ∧ UNCHANGED ⟨x, qu⟩

EnqueueTimeout ≜  ∧ state = "enqueued"
                  ∧ state′ = "Timeout"
                  ∧ UNCHANGED ⟨x, qu⟩

ProcessTimeout ≜  ∧ state = "processed"
                  ∧ state′ = "Timeout"
                  ∧ UNCHANGED ⟨x, qu⟩

DequeueTimeout ≜  ∧ state = "dequeued"
                  ∧ state′ = "Timeout"
                  ∧ UNCHANGED ⟨x, qu⟩

Return(d)  ≜ ∧ (state = "processed" ∨ state = "Error" ∨ state = "Timeout")
             ∧ d = IF state = "returned"
                   THEN x
                   ELSE IF state = "Error"
                   THEN "Error"
                   ELSE "Timeout"
             ∧ x′ = 0
             ∧ state′ = "init"
             ∧ UNCHANGED qu
```

```
Next ≜  ∨ ∃ d ∈ Data : Return(d)
        ∨ CallEnqueue(d)
        ∨ Process
        ∨ Dequeue
        ∨ Return(d)
        ∨ ProcessError
        ∨ DequeueError
        ∨ EnqueueError
        ∨ EnqueueTimeout
        ∨ DequeueTimeout
        ∨ ProcessTimeout

vars1 ≜ ≪state, qu, x≫

Fair ≜  ∀ d ∈ Data : WF_vars1(Return(d))
```

$\land\ \mathrm{WF}_{\mathtt{vars1}}(\mathtt{CallEnqueue(d)})$
$\land\ \mathrm{WF}_{\mathtt{vars1}}(\mathtt{Process})$
$\land\ \mathrm{WF}_{\mathtt{vars1}}(\mathtt{Dequeue})$
$\land\ \mathrm{WF}_{\mathtt{vars1}}(\mathtt{ProcessReturn})$
$\land\ \mathrm{WF}_{\mathtt{vars1}}(\mathtt{ProcessError})$
$\land\ \mathrm{WF}_{\mathtt{vars1}}(\mathtt{DequeueError})$
$\land\ \mathrm{WF}_{\mathtt{vars1}}(\mathtt{EnqueueError})$
$\land\ \mathrm{WF}_{\mathtt{vars1}}(\mathtt{ProcessTimeout})$
$\land\ \mathrm{WF}_{\mathtt{vars1}}(\mathtt{callTimeout})$
$\land\ \mathrm{WF}_{\mathtt{vars1}}(\mathtt{callError})$

$\mathtt{Spec} \triangleq \mathtt{Init} \land \Box[\mathtt{Next}]_{\mathtt{vars1}} \land \mathtt{Fair}$

Anhang C

UML Spezifikationen der Entwurfsmuster-Kombination

In diesem Anhang werden die Statechart-Diagramme der Klassen SensorAdapter (s. Abbildung C.1) und Sensor (s. Abbildung C.2) angegeben.

C.1 Statechart-Diagramm der Klasse *SensorAdapter*

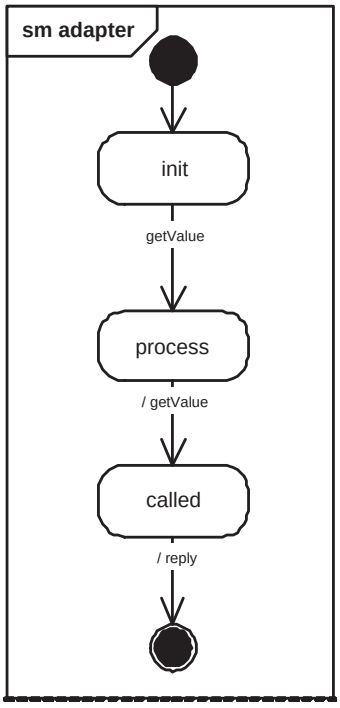


Abbildung C.1: Statechart-Diagramm eines SensorAdapter

C.2 Statechart-Diagramm der Klasse *Sensor*

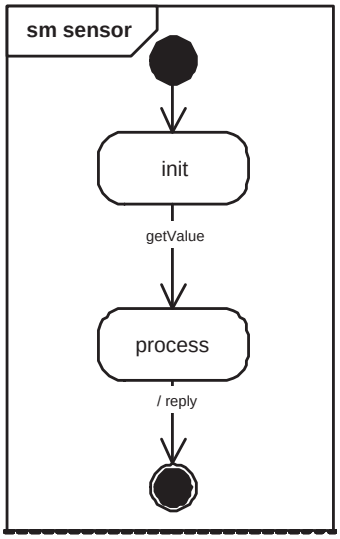


Abbildung C.2: Statechart-Diagramm eines Sensor

Anhang D

cTLA-Spezifikationen der Entwurfsmuster-Kombination

D.1 Prozesstyp *SensorTimes*

```
PROCESS SensorTimes

ACTIONS

enqueue;

dequeue;

process;

getValueCallReply;

P MAX TIME enqueue : tSensorEnqueue;

P MAX TIME dequeue : tSensorDequeue;

P MAX TIME process : tSensorProcess;

P MAX TIME getValueCallReply : tSensorGetValueCallReply;

END SensorTimes
```

D.2 Prozesstyp *SensorAdapterTimes*

```
PROCESS SensorAdapterTimes

ACTIONS
    enqueue;

    dequeue;

    process;

    getValueCall;

    getValueCallReply;

P MAX TIME enqueue : tAdapterEnqueue;

P MAX TIME dequeue : tAdapterDequeue;

P MAX TIME process : tAdapterProcess;

P MAX TIME getValueCall : tAdapterGetValueCall;

P MAX TIME getValueCallReply : tAdapterGetValueCallReply;
```

```
END SensorAdapterTimes
```

D.3 Prozesstyp *SensorProxyTimes*

```
PROCESS SensorProxyTimes

ACTIONS

enqueue;

dequeue;

process;

getValueCall;

getValueCallReply;

error1;

error2;

error3;

error4;

timeout1;

timeout2;

timeout3;

timeout4;

P MAX TIME enqueue : tProxyEnqueue;

P MAX TIME dequeue : tProxyDequeue;

P MAX TIME process : tProxyProcess;

P MAX TIME getValueCall : tProxyGetValueCall;

P MAX TIME getValueCallReply : tProxyGetValueCallReply;

P MAX TIME error1 : tProxyError1;

P MAX TIME error2 : tProxyError2;

P MAX TIME error3 : tProxyError3;

P MAX TIME error4 : tProxyError4;

P MAX TIME timeout1 : tProxyTimeout1;

P MAX TIME timeout2 : tProxyTimeout2;

P MAX TIME timeout3 : tProxyTimeout3;

P MAX TIME timeout4 : tProxyTimeout4;

END SensorProxyTimes
```

D.4 Prozesstyp *concreteCompositionOfSub1*

```
PROCESS concreteCompositionOfSub1

PROCESSES
 ! O1 : concreteControler;
 O2 : SensorProxy;
 TO2 : SensorProxyTimes;
 O3 : SensorAdapter;
 TO3 : AdapterTimes;
```

```
 O4 : Sensor;
 TO4 : SensorTimes;
 sd : SequenceDiagram(≪''O2.dequeue'', ''O2.actionProcess'', ''O2.actionGetValueCall'',
  ''O3.enqueue'', ''O3.actionProcess'', ''O3.actionGetValueCall'', ''O4.dequeue'',
  ''O4.actionProcess'', ''O4.actionGetValueCallReply'', ''O3.dequeue'',
  ''O3.actionGetValueCallReply'', ''O2.dequeue'', ''O2.actionGetValueCallReply''≫);

ACTIONS

ProxyDequeue(currentEventOccurence : String) ≙ ! dequeue
Systemaktion d. Proxys
    ∧ O2.dequeue(currentEventOccurence)
    ∧ TO2.dequeue
    ∧ O3.stutter
    ∧ TO3.stutter
    ∧ O4.stutter
    ∧ TO4.stutter
    ∧ sd.permittedActionOfSD(currentEventOccurence);

ProxyProcess(currentEventOccurence : String) ≙ ! process
Systemaktion d. Proxys
    ∧ O2.process(currentEventOccurence)
    ∧ TO2.process
    ∧ O3.stutter
    ∧ TO3.stutter
    ∧ O4.stutter
    ∧ TO4.stutter
    ∧ sd.permittedActionOfSD(currentEventOccurence);

setCall1(currentEventOccurence : String) ≙ ! setCall1
Systemaktion des Proxys
    ∧ O2.setCall1(currentEventOccurence)
    ∧ TO2.setCall1
    ∧ O3.stutter
    ∧ TO3.stutter
    ∧ O4.stutter
    ∧ TO4.stutter
    ∧ sd.permittedActionOfSD(currentEventOccurence);

setCall2(currentEventOccurence : String) ≙ ! setCall1
Systemaktion des Proxys
    ∧ O2.setCall2(currentEventOccurence)
    ∧ TO2.setCall2
    ∧ O3.stutter
    ∧ TO3.stutter
    ∧ O4.stutter
    ∧ TO4.stutter
    ∧ sd.permittedActionOfSD(currentEventOccurence);

setCall3(currentEventOccurence : String) ≙ ! setCall1
Systemaktion des Proxys
    ∧ O2.setCall2(currentEventOccurence)
    ∧ TO2.setCall2
    ∧ O3.stutter
    ∧ TO3.stutter
    ∧ O4.stutter
    ∧ TO4.stutter
    ∧ sd.permittedActionOfSD(currentEventOccurence);

ProxyError1 ≙ ! ProxyError1 Systemaktion d. Proxys
    ∧ O2.proxyError1(currentEventOccurence)
    ∧ TO2.proxyError1
```

```
    ∧ O3.stutter
    ∧ TO3.stutter
    ∧ O4.stutter
    ∧ TO4.stutter
    ∧ sd.permittedActionOfSD(currentEventOccurence);

ProxyError2 ≜ ! ProxyError2 Systemaktion d. Proxys
    ∧ O2.proxyError2(currentEventOccurence)
    ∧ TO2.proxyError2
    ∧ O3.stutter
    ∧ TO3.stutter
    ∧ O4.stutter
    ∧ TO4.stutter
    ∧ sd.permittedActionOfSD(currentEventOccurence);

ProxyError3 ≜ ! ProxyError3 Systemaktion d. Proxys
    ∧ O2.proxyError3(currentEventOccurence)
    ∧ TO2.proxyError3
    ∧ O3.stutter
    ∧ TO3.stutter
    ∧ O4.stutter
    ∧ TO4.stutter
    ∧ sd.permittedActionOfSD(currentEventOccurence);

ProxyError4 ≜ ! ProxyError4 Systemaktion d. Proxys
    ∧ O2.proxyError4(currentEventOccurence)
    ∧ TO2.proxyError4
    ∧ O3.stutter
    ∧ TO3.stutter
    ∧ O4.stutter
    ∧ TO4.stutter
    ∧ sd.permittedActionOfSD(currentEventOccurence);

ProxyTimeout1 ≜ ! ProxyTimeout1 Systemaktion d. Proxys
    ∧ O2.proxyTimeout1(currentEventOccurence)
    ∧ TO2.proxyTimeout1
    ∧ O3.stutter
    ∧ TO3.stutter
    ∧ O4.stutter
    ∧ TO4.stutter
    ∧ sd.permittedActionOfSD(currentEventOccurence);

ProxyTimeout2 ≜ ! ProxyTimeout2 Systemaktion d. Proxys
    ∧ O2.proxyTimeout2(currentEventOccurence)
    ∧ TO2.proxyTimeout2
    ∧ O3.stutter
    ∧ TO3.stutter
    ∧ O4.stutter
    ∧ TO4.stutter
    ∧ sd.permittedActionOfSD(currentEventOccurence);

ProxyTimeout3 ≜ ! ProxyTimeout3 Systemaktion d. Proxys
    ∧ O2.proxyTimeout3(currentEventOccurence)
    ∧ TO2.proxyTimeout3
    ∧ O3.stutter
    ∧ TO3.stutter
    ∧ O4.stutter
    ∧ TO4.stutter
    ∧ sd.permittedActionOfSD(currentEventOccurence);

ProxyTimeout4 ≜ ! ProxyTimeout4 Systemaktion d. Proxys
    ∧ O2.proxyTimeout3(currentEventOccurence)
```

```
    ∧ TO2.proxyTimeout4
    ∧ O3.stutter
    ∧ TO3.stutter
    ∧ O4.stutter
    ∧ TO4.stutter
    ∧ sd.permittedActionOfSD(currentEventOccurence);

CallToAdapter(val : Real; currentEventOccurence : String) ≜ !
getValue Aufruf des Adapters
    ∧ O2.getcallValueCall(value, currentEventOccurence)
    ∧ TO2.getValueCall
    ∧ O3.enqueue(value)
    ∧ TO3.getValueCall
    ∧ O4.stutter
    ∧ TO4.stutter
    ∧ sd.permittedActionOfSD(currentEventOccurence);

AdapterProcess(currentEventOccurence : String) ≜ ! Verarbeitung
des Adapters
    ∧ O2.stutter
    ∧ TO2.stutter
    ∧ O3.process(currentEventOccurence)
    ∧ TO3.process
    ∧ O4.stutter
    ∧ TO4.stutter
    ∧ sd.permittedActionOfSD(currentEventOccurence);

CallToSensor(val : Real; currentEventOccurence : String) ≜ !
Aufruf des Sensors
    ∧ O2.stutter
    ∧ TO2.stutter
    ∧ O3.getValueCall(value, currentEventOccurence)
    ∧ TO3.getValueCall
    ∧ O4.enqueue(value)
    ∧ TO4.enqueue
    ∧ sd.permittedActionOfSD(currentEventOccurence);

ReturnFromAdapter(val : Real; currentEventOccurence : String) ≜ !
Adapterantwort
    ∧ O1.stutter
    ∧ TO1.stutter
    ∧ O2.enqueue(value)
    ∧ TO2.enqueue
    ∧ O3.getValueCallReply(value, currentEventOccurence)
    ∧ TO3.return
    ∧ O4.stutter
    ∧ sd.permittedActionOfSD(currentEventOccurence);

SensorDequeue(currentEventOccurence : String) ≜ ... !
Entfernung aus Sensorqueue s. Anhang
    ∧ O2.stutter
    ∧ TO2.stutter
    ∧ O3.stutter
    ∧ TO3.stutter
    ∧ O4.dequeue(currentEventOccurence)
    ∧ TO4.dequeue
    ∧ sd.permittedActionOfSD(currentEventOccurence);

SensorProcess(currentEventOccurence : String) ≜ ! Verarbeitung
des Sensors
    ∧ O2.stutter
    ∧ TO2.stutter
```

```
    ∧ O3.stutter
    ∧ TO3.stutter
    ∧ O4.process(currentEventOccurence)
    ∧ TO4.process
    ∧ sd.permittedActionOfSD(currentEventOccurence);

ReturnFromSensor(val : Real; currentEventOccurence : String) ≜ !
Antwort des Sensors
    ∧ O2.stutter
    ∧ TO2.stutter
    ∧ O3.enqueue(value)
    ∧ TO3.enqueue
    ∧ O4.getValueCallReply(value, currentEventOccurence)
    ∧ TO4.getValueCallReply
    ∧ sd.permittedActionOfSD(currentEventOccurence);

END concreteCompositionOfSub1
```

D.5 TLC-Spezifikation des Prozesstyps *concreteController*

```
MODULE concreteController
EXTENDS Naturals, Sequences, TLC

CONSTANTS Data, Operation

VARIABLES
      x,
      y,
      sync,
      qu,
      state,
      ctrlActReadxActReady,
      existsO1,
      existsO2,
      existsO3,
      valueO1,
      valueO2,
      valueO3

ox ≜ x

oCtrlActReadxActReady ≜ ctrlActReadxActReady

oExistsO1 ≜ existsO1

oValueO1 ≜ valueO1

oExistsO2 ≜ existsO2

oValueO2 ≜ valueO2

oExistsO3 ≜ existsO3

oValueO3 ≜ valueO3

ostate ≜
      CASE state = "init" ∧ state = "init" → "init"
      □ state = "stopped" → "stopped"
      □ state = "s1" → "waitSensor"
      □ state = "s2" → "waitSensor"
      □ state = "s3" → "waitSensor"
      □ state = "s4" → "waitSensor"
      □ state = "s5" → "valuesComputed"
      □ state = "s6" → "valuesComputed"
```

```
      □ state = "s7" → "valuesComputed"
      □ state = "s8" → "valuesComputed"
      □ state = "s9" → "waitActuator"
      □ state = "s10" → "waitActuator"
      □ state = "s11" → "waitActuator"
      □ state = "s12" → "waitActuator"
      □ state = "wait" → "wait"
      □ state = "waitSensor" → "waitSensor"
      □ state = "getEnqueued" → "getEnqueued"
      □ state = "valueDequeued" → "valueDequeued"
      □ state = "valuesComputed" → "valuesComputed"
      □ state = "waitActuator" → "waitActuator"
      □ state = "returnEnqueued" → "returnEnqueued"

oy ≜ y

osync ≜ sync

oqu ≜ IF state = "s1" ∨ state = "s2" ∨ state = "s3" ∨ state = "s4" ∨
         state = "s5" ∨ state = "s6" ∨ state = "s7" ∨ state ="s8" ∨
         state = "s9" ∨ state = "s10" ∨ state = "s11" ∨ state = "s12"
      THEN ≪ ≫
      ELSE qu

Init ≜ ∧ x = 0
       ∧ y = 0
       ∧ sync = FALSE
       ∧ state = "init"
       ∧ qu = ≪ ≫
       ∧ ctrlActReadxActReady = FALSE
       ∧ value01 = 0
       ∧ exists01 = FALSE
       ∧ value02 = 0
       ∧ exists02 = FALSE
       ∧ value03 = 0
       ∧ exists03  = FALSE

stop ≜
      ∧ state = "init"
      ∧ state′ = "stopped"
      ∧ UNCHANGED ⟨x, y, qu, sync⟩
      ∧ UNCHANGED ctrlActReadxActReady
      ∧ UNCHANGED ⟨exists01, exists02, exists03⟩
      ∧ UNCHANGED ⟨value01, value02, value03⟩

restart ≜
      ∧ state = "stopped"
      ∧ state′ = "init"
      ∧ UNCHANGED ⟨x, y, qu, sync⟩
      ∧ UNCHANGED ctrlActReadxActReady
      ∧ UNCHANGED ⟨exists01, exists02, exists03⟩
      ∧ UNCHANGED ⟨value01, value02, value03⟩

loop ≜
      ∧ state = "init"
      ∧ state′ = "wait"
      ∧ UNCHANGED ⟨x, y, qu, sync⟩
      ∧ UNCHANGED ctrlActReadxActReady
      ∧ UNCHANGED ⟨exists01, exists02, exists03⟩
      ∧ UNCHANGED ⟨value01, value02, value03⟩

enqInterrupt(message)  ≜
```

```
        ∧ state = "waitSensor" ∨ state = "valuesComputed" ∨ state = "waitActuator"
        ∧ state′ = IF state = "waitSensor"
                   THEN "s1"
                   ELSE IF state = "valuesComputed"
                   THEN "s5"
                   ELSE "s9"
        ∧ qu′ = Append(qu, message)
        ∧ UNCHANGED ⟨x, y, sync⟩
        ∧ UNCHANGED ctrlActReadxActReady
        ∧ UNCHANGED ⟨exists01, exists02, exists03⟩
        ∧ UNCHANGED ⟨value01, value02, value03⟩
deqInterrupt ≜
        ∧ state = "s1" ∨ state = "s5" ∨ state = "s9"
        ∧ state′ = IF state = "s1"
                   THEN "s2"
                   ELSE IF state = "s5"
                   THEN "s6"
                   ELSE "s10"
        ∧ qu′ = Tail(qu)
        ∧ UNCHANGED x
        ∧ UNCHANGED y
        ∧ UNCHANGED sync
        ∧ UNCHANGED ctrlActReadxActReady
        ∧ UNCHANGED ⟨exists01, exists02, exists03⟩
        ∧ UNCHANGED ⟨value01, value02, value03⟩
enqResume(message) ≜
        ∧ state = "s2" ∨ state = "s6" ∨ state = "s10"
        ∧ state′ = IF state = "s2"
                   THEN "s3"
                   ELSE IF state = "s6"
                   THEN "s7"
                   ELSE "s11"
        ∧ qu′ = Append(qu, message)
        ∧ UNCHANGED x
        ∧ UNCHANGED y
        ∧ UNCHANGED sync
        ∧ UNCHANGED ctrlActReadxActReady
        ∧ UNCHANGED ⟨exists01, exists02, exists03⟩
        ∧ UNCHANGED ⟨value01, value02, value03⟩
deqResume ≜
        ∧ state = "s3" ∨ state = "s7" ∨ state = "s11"
        ∧ state′ = IF state = "s3"
                   THEN "waitSensor"
                   ELSE IF state = "s7"
                   THEN "valuesComputed"
                   ELSE "waitActuator"
        ∧ qu′ = Tail(qu)
        ∧ UNCHANGED ⟨x, y, sync⟩
        ∧ UNCHANGED ctrlActReadxActReady
        ∧ UNCHANGED ⟨exists01, exists02, exists03⟩
        ∧ UNCHANGED ⟨value01, value02, value03⟩
sleep ≜
        ∧ state = "waitSensor" ∨ state = "waitActuator" ∨ state = "valuesComputed"
        ∧ state′ = IF state = "waitSensor"
                   THEN "s4"
                   ELSE IF state = "valuesComputed"
                   THEN "s8"
```

```
                ELSE "s12"
        ∧ UNCHANGED ⟨x, y, qu, sync⟩
        ∧ UNCHANGED ctrlActReadxActReady
        ∧ UNCHANGED ⟨exists01, exists02, exists03⟩
        ∧ UNCHANGED ⟨value01, value02, value03⟩

awake ≜
        ∧ state = "s4" ∨ state = "s8" ∨ state = "s12"
        ∧ state′ = IF state = "s4"
                THEN "waitSensor"
                ELSE IF state = "s8"
                THEN "valuesComputed"
                ELSE "waitActuator"
        ∧ UNCHANGED ⟨x, y, qu, sync⟩
        ∧ UNCHANGED ctrlActReadxActReady
        ∧ UNCHANGED ⟨exists01, exists02, exists03⟩
        ∧ UNCHANGED ⟨value01, value02, value03⟩


getValueCall ≜
        ∧ state = "wait"
        ∧ state′ = "waitSensor"
        ∧ sync′ = TRUE
        ! ∧ value = [op ↦ "getValue", val ↦ oy]
        ∧ UNCHANGED ⟨x, y, qu rangle:
        ∧ UNCHANGED ⟨octrlActReadxActReady⟩
        ∧ UNCHANGED ⟨exists01, exists02, exists03⟩
        ∧ UNCHANGED ⟨value01, value02, value03⟩

setValueCall(value) ≜
        ∧ state = "valuesComputed"
        ∧ state′ = "waitActuator"
        ∧ sync′ = TRUE
        ! ∧ value = [op ↦ "setValue", val ↦ oy]
        ∧ UNCHANGED ⟨x, y, qu, sync⟩
        ∧ UNCHANGED ctrlActReadxActReady
        ∧ UNCHANGED ⟨exists01, exists02, exists03⟩
        ∧ UNCHANGED ⟨value01, value02, value03⟩

getEnqueue(value) ≜
        ∧ state = "waitSensor"
        ∧ state′ = "getEnqueued"
        ∧ sync′ = FALSE
        ∧ qu′ = Append(qu, value)
        ∧ UNCHANGED ⟨x, y⟩
        ∧ UNCHANGED ctrlActReadxActReady
        ∧ UNCHANGED ⟨exists01, exists02, exists03⟩
        ∧ UNCHANGED ⟨value01, value02, value03⟩

setEnqueue(value) ≜
        ∧ state = "waitActuator"
        ∧ state′ = "returnEnqueued"
        ∧ sync′ = FALSE
        ! ∧ qu′ = Append(qu, [op ↦ "none", val ↦ value])
        ∧ qu′ = Append(qu, value)
        ∧ UNCHANGED ⟨x, y⟩
        ∧ UNCHANGED ctrlActReadxActReady
        ∧ UNCHANGED ⟨exists01, exists02, exists03⟩
        ∧ UNCHANGED ⟨value01, value02, value03⟩


getDequeue ≜
```

```
        ∧ state = "getEnqueued"
        ∧ state′ = "valueDequeued"
        ∧ qu′ = Tail(qu)
        ∧ x′ = Head(qu)
        ! ∧ x′ = Head(qu).val
        ∧ UNCHANGED ⟨y, sync⟩
        ∧ UNCHANGED ctrlActReadxActReady
        ∧ UNCHANGED ⟨exists01, exists02, exists03⟩
        ∧ UNCHANGED ⟨value01, value02, value03⟩

setDequeue ≜
        ∧ state = "returnEnqueued"
        ∧ state′ = "init"
        ! ∧ qu′ = Append(qu, [op ↦ "none", val ↦ value])
        ∧ qu′ = Tail(qu)
        ∧ x′ = Head(qu)
        ! ∧ x′ = Head(qu).val
        ∧ UNCHANGED ⟨y, sync⟩
        ∧ UNCHANGED ctrlActReadxActReady
        ∧ UNCHANGED ⟨exists01, exists02, exists03⟩
        ∧ UNCHANGED ⟨value01, value02, value03⟩

compute ≜
       ∧ state = "valueDequeued"
       ∧ state′ = "valuesComputed"
       ∧ y′ = x
       ∧ UNCHANGED ⟨x, qu, sync⟩
       ∧ UNCHANGED ctrlActReadxActReady
       ∧ UNCHANGED ⟨exists01, exists02, exists03⟩
       ∧ UNCHANGED ⟨value01, value02, value03⟩

Next ≜ ∨ ∃ message ∈ Operation :
                            ∨ stop
                            ∨ restart
                            ∨ loop
                            ∨ deqInterrupt
                            ∨ deqResume
                            ∨ getValueCall
                            ∨ getDequeue
                            ∨ setDequeue
                            ∨ compute
                            ∨ enqResume(message)
                            ∨ enqInterrupt(message)
                            ∨ ∃ value ∈ Data : getEnqueue([op ↦ message, val ↦ value])
                            ∨ setEnqueue([op ↦ message, val ↦ value])
                            ∨ setValueCall(value)

vars ≜ ≪x, y, sync, qu, state≫

CFair ≜ ∀ message ∈ Operation : ∀ value ∈ Data :
                                    ∧ WF_vars(stop)
                                    ∧ WF_vars(restart)
                                    ∧ WF_vars(loop)
                                    ∧ WF_vars(deqInterrupt)
                                    ∧ WF_vars(deqResume)
                                    ∧ WF_vars(getValueCall)
                                    ∧ WF_vars(setValueCall(value))
                                    ∧ WF_vars(getDequeue)
                                    ∧ WF_vars(setDequeue)
                                    ∧ WF_vars(compute)
                                    ∧ WF_vars(enqResume(message))
```

```
                                    ∧ WF_vars(enqInterrupt(message))
                                    ∧ WF_vars(getEnqueue([op ↦ message, val ↦ value]))
                                    ∧ WF_vars(setEnqueue([op ↦ message, val ↦ value]))

Spec ≜ Init ∧ □[Next]_vars ∧ CFair
```

```
AT ≜ INSTANCE abstractContr2 x ← ox, y ←oy, state ← ostate, qu
← oqu, sync ← osync
```

D.6 TLC-Spezifikation für das *SubsystemSensor*

```
MODULE SubsystemSensor

EXTENDS Naturals, Sequences, TLC, PrintValues

CONSTANTS Data, EO, EOA, States

VARIABLES
      currentTrace,
      aSxx,
      aSxsSensor,
      aSxqu,
      adapterxx,
      adapterxqu,
      adapterxsAdapter,
      proxyxx,
      proxyxqu,
      proxyxss,
      hqu

ox ≜ proxyxx

ostate ≜  CASE proxyxss = "init" ∧ aSxsSensor = "init" ∧ aSxsSensor = "init" →
"init"
□ proxyxss = "callEnqueued" ∧ adapterxsAdapter = "init" ∧ aSxsSensor = "init" →
"enqueued"
□ proxyxss = "callDequeued" ∧ adapterxsAdapter = "init" ∧ aSxsSensor = "init" →
"enqueued"
□ proxyxss = "processed" ∧ adapterxsAdapter = "init" ∧ aSxsSensor = "init" →
"enqueued"
□ proxyxss = "readyForCall" ∧ adapterxsAdapter = "init" ∧ aSxsSensor = "init" →
"enqueued"
□ proxyxss = "called" ∧ adapterxsAdapter = "enqueued" ∧ aSxsSensor = "init" →
"enqueued"
□ proxyxss = "called" ∧ adapterxsAdapter = "dequeued" ∧ aSxsSensor = "init" →
"enqueued"
□ proxyxss = "called" ∧ adapterxsAdapter = "called" ∧ aSxsSensor = "enqueued" →
"enqueued"
□ proxyxss = "called" ∧ adapterxsAdapter = "called" ∧ aSxsSensor = "dequeued" →
"dequeued"
□ proxyxss = "called" ∧ adapterxsAdapter = "called" ∧ aSxsSensor = "processed" →
"processed"
□ proxyxss = "called" ∧ adapterxsAdapter = "returnEnqueued" ∧ aSxsSensor = "init" →
"processed"
□ proxyxss = "returnEnqueued" ∧ adapterxsAdapter = "init" ∧ aSxsSensor = "init" →
"processed"
□ proxyxss = "returnDequeued" ∧ adapterxsAdapter = "init" ∧ aSxsSensor = "init" →
"processed"
□ proxyxss = "readyForError" ∧ adapterxsAdapter = "init" ∧ aSxsSensor = "init" →
"processed"
□ proxyxss = "calledError" ∧ adapterxsAdapter = "enqueued" ∧ aSxsSensor = "init" →
```

```
"processed"
□ proxyxss = "calledError" ∧ adapterxsAdapter = "dequeued" ∧ aSxsSensor = "init" →
"processed"
□ proxyxss = "calledError" ∧ adapterxsAdapter = "called" ∧ aSxsSensor = "enqueued" →
"processed"
□ proxyxss = "calledError" ∧ adapterxsAdapter = "called" ∧ aSxsSensor = "dequeued" →
"processed"
□ proxyxss = "calledError" ∧ adapterxsAdapter = "called" ∧ aSxsSensor = "processed" →
"processed"
□ proxyxss = "calledError" ∧ adapterxsAdapter = "returnEnqueued" ∧ aSxsSensor = "init" →
"processed"
□ proxyxss = "returnDeqError" ∧ adapterxsAdapter = "init" ∧ aSxsSensor = "init" →
"processed"
□ proxyxss = "returnEnqError" ∧ adapterxsAdapter = "init" ∧ aSxsSensor = "init" →
"processed"
□ proxyxss = "Error" ∧ adapterxsAdapter = "init" ∧ aSxsSensor = "init" →
"Error"
□ proxyxss = "readyForTimeout" ∧ adapterxsAdapter = "init" ∧ aSxsSensor = "init" →
"dequeued"
□ proxyxss = "calledTimeout" ∧ adapterxsAdapter = "enqueued" ∧ aSxsSensor = "init" →
"processed"
□ proxyxss = "calledTimeout" ∧ adapterxsAdapter = "dequeued" ∧ aSxsSensor = "init" →
"processed"
□ proxyxss = "calledTimeout" ∧ adapterxsAdapter = "called" ∧ aSxsSensor = "enqueued" →
"processed"
□ proxyxss = "calledTimeout" ∧ adapterxsAdapter = "called" ∧ aSxsSensor = "dequeued" →
"processed"
□ proxyxss = "calledTimeout" ∧ adapterxsAdapter = "called" ∧ aSxsSensor = "processed" →
"processed"
□ proxyxss = "calledTimeout" ∧ adapterxsAdapter = "returnEnqueued" ∧ aSxsSensor = "init" →
"processed"
□ proxyxss = "returnDeqTimeout" ∧ adapterxsAdapter = "init" ∧ aSxsSensor = "init" →
"processed"
□ proxyxss = "returnEnqTimeout" ∧ adapterxsAdapter = "init" ∧ aSxsSensor = "init" →
"processed"
□ proxyxss = "Timeout" ∧ adapterxsAdapter = "init" ∧ aSxsSensor = "init" →
"Timeout"

oqu ≜ IF(aSxsSensor = "dequeued" ∨ aSxsSensor = "processed")
      THEN ≪ ≫
      ELSE IF(proxyxss = "callEnqueued" ∨ proxyxss = "callDequeued" ∨
              proxyxss = "processed" ∨ proxyxss = "readyForCall" ∨
              proxyxss = "called" ∨ adapterxsAdapter = "dequeued" ∨
              adapterxsAdapter = "called" ∨ proxyxss = "calledError")
      THEN hqu
      ELSE ≪ ≫


ocurrentTrace ≜ CASE currentTrace = ≪≫ → ≪≫
□ currentTrace = ≪ "O2.actionDequeue"≫ → ≪≫
□ currentTrace = ≪ "O2.actionDequeue", "O2.actionProcess"≫ → ≪≫
□ currentTrace = ≪ "O2.actionDequeue", "O2.actionProcess", "O2.actionSetCall1"≫ → ≪≫
□ currentTrace = ≪ "O2.actionDequeue", "O2.actionProcess", "O2.actionSetCall1",
"O2.actionCall"≫ → ≪≫
□ currentTrace = ≪ "O2.actionDequeue", "O2.actionProcess", "O2.actionSetCall1",
"O2.actionCall"≫ → ≪≫
□ currentTrace = ≪ "O2.actionDequeue", "O2.actionProcess", "O2.actionSetCall1",
"O2.actionError1"≫ → ≪≫
□ currentTrace = ≪ "O2.actionDequeue", "O2.actionProcess", "O2.actionSetCall1",
"O2.actionCall",  "O3.actionDequeue"≫ → ≪≫
□ currentTrace = ≪ "O2.actionDequeue", "O2.actionProcess", "O2.actionSetCall1",
"O2.actionCall", "O3.actionDequeue", "O3.actionCall"≫ → ≪≫
□ currentTrace = ≪ "O2.actionDequeue", "O2.actionProcess", "O2.actionSetCall1",
"O2.actionCall", "O3.actionDequeue", "O3.actionCall", "O4.actionDequeue"≫ →
≪"O2.dequeue"≫
□ currentTrace = ≪ "O2.actionDequeue", "O2.actionProcess", "O2.actionSetCall1",
```

```
"O3.actionDequeue", "O3.actionCall", "O4.actionDequeue"≫ → ≪"O2.dequeue"≫
□ currentTrace = ≪ "O2.actionDequeue", "O2.actionProcess", "O2.actionSetCall1",
"O2.actionCall", "O3.actionDequeue", "O3.actionCall", "O4.actionDequeue",
"O4.actionProcess" ≫ → ≪"O2.dequeue", "O2.process"≫
□ currentTrace = ≪ "O2.actionDequeue", "O2.actionProcess", "O2.actionSetCall1",
"O2.actionCall", "O3.actionDequeue", "O3.actionCall", "O4.actionDequeue",
"O4.actionProcess", "O4.actionReturn" ≫ → ≪"O2.dequeue", "O2.process"≫
□ currentTrace = ≪ "O2.actionDequeue", "O2.actionProcess", "O2.actionSetCall1",
"O2.actionCall", "O3.actionDequeue", "O3.actionCall", "O4.actionDequeue",
"O4.actionProcess", "O4.actionReturn" ≫ → ≪"O2.dequeue", "O2.process"≫
□ currentTrace = ≪ "O2.actionDequeue", "O2.actionProcess", "O2.actionSetCall1",
"O2.actionCall", "O3.actionDequeue", "O3.actionCall", "O4.actionDequeue",
"O4.actionProcess", "O4.actionReturn", "O3.actionReturn" ≫ →
≪"O2.dequeue", "O2.process"≫
□ currentTrace = ≪ "O2.actionDequeue", "O2.actionProcess", "O2.actionSetCall2"≫
→ ≪≫
□ currentTrace = ≪ "O2.actionDequeue", "O2.actionProcess", "O2.actionSetCall2",
"O2.actionCall"≫ → ≪≫
□ currentTrace = ≪ "O2.actionDequeue", "O2.actionProcess", "O2.actionSetCall2",
"O2.actionError1"≫ → ≪"O2.enqueueError"≫
□ currentTrace = ≪ "O2.actionDequeue", "O2.actionProcess", "O2.actionSetCall2",
"O2.actionCall", "O3.actionDequeue" ≫ → ≪≫
□ currentTrace = ≪ "O2.actionDequeue", "O2.actionProcess", "O2.actionSetCall2",
"O2.actionCall", "O2.actionDequeue" ≫ → ≪≫
□ currentTrace = ≪ "O2.actionDequeue", "O2.actionProcess", "O2.actionSetCall2",
"O2.actionCall", "O2.actionError2" ≫ → ≪"O2.enqueueError"≫
□ currentTrace = ≪ "O2.actionDequeue", "O2.actionProcess", "O2.actionSetCall2",
"O2.actionCall", "O3.actionDequeue", "O2.actionError2" ≫ → ≪"O2.enqueueError"≫
□ currentTrace = ≪ "O2.actionDequeue", "O2.actionProcess", "O2.actionSetCall2",
"O2.actionCall", "O3.actionDequeue", "O3.actionCall" ≫ → ≪≫
□ currentTrace = ≪ "O2.actionDequeue", "O2.actionProcess", "O2.actionSetCall2",
"O2.actionCall", "O3.actionDequeue", "O3.actionCall", "O2.actionError2" ≫ →
≪"O2.enqueueError"≫
□ currentTrace = ≪ "O2.actionDequeue", "O2.actionProcess", "O2.actionSetCall2",
"O2.actionCall", "O3.actionDequeue", "O3.actionCall", "O4.actionDequeue" ≫
→ ≪"O2.dequeue"≫
□ currentTrace = ≪ "O2.actionDequeue", "O2.actionProcess", "O2.actionSetCall2",
"O2.actionCall", "O3.actionDequeue", "O3.actionCall", "O4.actionDequeue",
"O2.actionError2" ≫  → ≪"O2.dequeue", "O2.dequeueError"≫
□ currentTrace = ≪ "O2.actionDequeue", "O2.actionProcess", "O2.actionSetCall2",
 "O2.actionCall", "O3.actionDequeue", "O3.actionCall", "O4.actionDequeue",
 "O4.actionProcess" ≫ → ≪"O2.dequeue", "O2.process"≫
□ currentTrace = ≪ "O2.actionDequeue", "O2.actionProcess", "O2.actionSetCall2",
"O2.actionCall", "O3.actionDequeue", "O3.actionCall", "O4.actionDequeue",
"O4.actionProcess", "O2.actionError2" ≫ → ≪"O2.dequeue", "O2.process",
"O2.processError"≫

ProxyDequeue(cE0) ≜
               ∧ cE0 = "O2.actionDequeue"
               ∧ Append(currentTrace, cE0) ∈ setOfPossibleTraces
               ∧ currentTrace′ = Append(currentTrace, cE0)
               ∧ (proxyxss = "callEnqueued" ∨ proxyxss = "returnEnqueued"
                   ∨ proxyxss = "returnEnqError" ∨ proxyxss = "returnEnqTimeout")
               ∧  proxyxss′ = IF(proxyxss = "callEnqueued")
                              THEN "callDequeued"
                              ELSE IF proxyxss = "returnEnqueued"
                              THEN "returnDequeued"
                              ELSE IF proxyxss = "returnEnqError"
                              THEN "returnDeqError"
                              ELSE "returnDeqTimeout"
```

```
                ∧ proxyxqu′ = Tail(proxyxqu)
                ∧ UNCHANGED ⟨adapterxsAdapter, adapterxqu, adapterxx⟩
                ∧ UNCHANGED ⟨aSxqu, aSxsSensor, aSxx⟩
                ∧ UNCHANGED aSxqu
                ∧ UNCHANGED aSxsSensor
                ∧ UNCHANGED aSxx
                ∧ UNCHANGED adapterxsAdapter
                ∧ UNCHANGED adapterxqu
                ∧ UNCHANGED adapterxx
                ∧ UNCHANGED proxyxx
                ∧ UNCHANGED hqu

ProxyProcess(cE0) ≜
          ∧ cE0 = "02.actionProcess"
          ∧ Append(currentTrace, cE0) ∈ setOfPossibleTraces
          ∧ proxyxss = "callDequeued"
          ∧ currentTrace′ = Append(currentTrace, cE0)
          ∧ proxyxss′ = "processed"
          ∧ UNCHANGED ⟨proxyxqu, proxyxx⟩
          ∧ UNCHANGED ⟨adapterxsAdapter, adapterxqu, adapterxx⟩
          ∧ UNCHANGED ⟨aSxqu, aSxsSensor, aSxx⟩
          ∧ UNCHANGED hqu

setCall1(cE0) ≜
      ∧ cE0 = "02.actionSetCall1"
      ∧ Append(currentTrace, cE0) ∈ setOfPossibleTraces
      ∧ currentTrace′ = Append(currentTrace, cE0)
      ∧ proxyxss = "processed"
      ∧ proxyxss′ = "readyForCall"
      ∧ UNCHANGED ⟨proxyxqu, proxyxx⟩
      ∧ UNCHANGED ⟨adapterxsAdapter, adapterxqu, adapterxx⟩
      ∧ UNCHANGED ⟨aSxqu, aSxsSensor, aSxx⟩
      ∧ UNCHANGED hqu

setCall2(cE0) ≜
      ∧ cE0 = "02.actionSetCall2"
      ∧ Append(currentTrace, cE0) ∈ setOfPossibleTraces
      ∧ proxyxss = "processed"
      ∧ currentTrace′ = Append(currentTrace, cE0)
      ∧ proxyxss′ = "readyForError"
      ∧ UNCHANGED ⟨proxyxqu, proxyxx⟩
      ∧ UNCHANGED ⟨adapterxsAdapter, adapterxqu, adapterxx⟩
      ∧ UNCHANGED ⟨aSxqu, aSxsSensor, aSxx⟩
      ∧ UNCHANGED hqu

setCall3(cE0) ≜
      ∧ cE0 = "02.actionSetCall2"
      ∧ Append(currentTrace, cE0) ∈ setOfPossibleTraces
      ∧ proxyxss = "processed"
      ∧ currentTrace′ = Append(currentTrace, cE0)
      ∧ proxyxss′ = "readyForTimeout"
      ∧ UNCHANGED ⟨proxyxqu, proxyxx⟩
      ∧ UNCHANGED ⟨adapterxsAdapter, adapterxqu, adapterxx⟩
      ∧ UNCHANGED ⟨aSxqu, aSxsSensor, aSxx⟩
      ∧ UNCHANGED hqu

ProxyError1(cE0) ≜
          ∧ cE0 = "02.actionError1"
          ∧ Append(currentTrace, cE0) ∈ setOfPossibleTraces
          ∧ proxyxss = "readyForError"
          ∧ currentTrace′ = Append(currentTrace, cE0)
```

```
        ∧ aSxqu′ = ≪ ≫
        ∧ aSxsSensor′ = "init"
        ∧ aSxx′ = 0
        ∧ adapterxsAdapter′ = "init"
        ∧ adapterxqu′ = ≪ ≫
        ∧ adapterxx′ = 0
        ∧ proxyxss′ = "Error"
        ∧ UNCHANGED proxyxqu
        ∧ UNCHANGED proxyxx
        ∧ UNCHANGED hqu

ProxyTimeout1(cE0) ≜
        ∧ cE0 = "O2.actionTimeout1"
        ∧ Append(currentTrace, cE0) ∈ setOfPossibleTraces
        ∧ proxyxss = "readyForTimeout"
        ∧ currentTrace′ = Append(currentTrace, cE0)
        ∧ aSxqu′ = ≪ ≫
        ∧ aSxsSensor′ = "init"
        ∧ aSxx′ = 0
        ∧ adapterxsAdapter′ = "init"
        ∧ adapterxqu′ = ≪ ≫
        ∧ adapterxx′ = 0
        ∧ proxyxss′ = "Timeout"
        ∧ UNCHANGED ⟨proxyxqu, proxyxx⟩
        ∧ UNCHANGED hqu

ProxyError2(cE0) ≜
        ∧ cE0 = "O2.actionError2"
        ∧ Append(currentTrace, cE0) ∈ setOfPossibleTraces
        ∧ currentTrace′ = Append(currentTrace, cE0)
        ∧ proxyxss = "calledError"
        ∧ proxyxss′ = "Error"
        ∧ currentTrace′ = Append(currentTrace, cE0)
        ∧ aSxqu′ = ≪ ≫
        ∧ aSxsSensor′ = "init"
        ∧ aSxx′ = 0
        ∧ adapterxsAdapter′ = "init"
        ∧ adapterxqu′ = ≪ ≫
        ∧ adapterxx′ = 0
        ∧ UNCHANGED ⟨proxyxqu, proxyxx⟩
        ∧ UNCHANGED hqu

ProxyTimeout2(cE0) ≜
        ∧ cE0 = "O2.actionTimeout2"
        ∧ Append(currentTrace, cE0) ∈ setOfPossibleTraces
        ∧ proxyxss = "calledTimeout"
        ∧ aSxqu′ = ≪ ≫
        ∧ aSxsSensor′ = "init"
        ∧ aSxx′ = 0
        ∧ adapterxsAdapter′ = "init"
        ∧ adapterxqu′ = ≪ ≫
        ∧ adapterxx′ = 0
        ∧ currentTrace′ = Append(currentTrace, cE0)
        ∧ proxyxss′ = "Timeout"
        ∧ UNCHANGED ⟨proxyxqu, proxyxx⟩
        ∧ UNCHANGED hqu

ProxyError3(cE0) ≜
        ∧ cE0 = "O2.actionError3"
```

```
        ∧ Append(currentTrace, cE0) ∈ setOfPossibleTraces
        ∧ currentTrace′ = Append(currentTrace, cE0)
        ∧ proxyxss = "returnEnqError"
        ∧ proxyxss′ = "Error"
        ∧ UNCHANGED ⟨proxyxqu, proxyxx⟩
        ∧ UNCHANGED ⟨adapterxsAdapter, adapterxqu, adapterxx⟩
        ∧ UNCHANGED ⟨aSxqu, aSxsSensor, aSxx⟩
        ∧ UNCHANGED hqu

ProxyTimeout3(cE0) ≜
        ∧ cE0 = "02.actionTimeout3"
        ∧ Append(currentTrace, cE0) ∈ setOfPossibleTraces
        ∧ proxyxss = "returnEnqTimeout"
        ∧ currentTrace′ = Append(currentTrace, cE0)
        ∧ proxyxss′ = "Timeout"
        ∧ UNCHANGED ⟨adapterxsAdapter, adapterxqu, adapterxx⟩
        ∧ UNCHANGED ⟨aSxqu, aSxsSensor, aSxx⟩
        ∧ UNCHANGED ⟨proxyxqu, proxyxx⟩
        ∧ UNCHANGED hqu

ProxyError4(cE0) ≜
        ∧ cE0 = "02.actionError4"
        ∧ Append(currentTrace, cE0) ∈ setOfPossibleTraces
        ∧ currentTrace′ = Append(currentTrace, cE0)
        ∧ proxyxss = "returnDeqError"
        ∧ proxyxss′ = "Error"
        ∧ proxyxx = 0
        ∧ UNCHANGED proxyxqu
        ∧ UNCHANGED ⟨adapterxsAdapter, adapterxqu, adapterxx⟩
        ∧ UNCHANGED ⟨aSxqu, aSxsSensor, aSxx⟩
        ∧ UNCHANGED hqu

ProxyTimeout4(cE0) ≜
        ∧ cE0 = "02.actionTimeout4"
        ∧ Append(currentTrace, cE0) ∈ setOfPossibleTraces
        ∧ currentTrace′ = Append(currentTrace, cE0)
        ∧ proxyxss = "returnDeqTimeout"
        ∧ proxyxss′ = "Timeout"
        ∧ proxyxx = "Timeout"
        ∧ UNCHANGED proxyxqu
        ∧ UNCHANGED ⟨adapterxsAdapter, adapterxqu, adapterxx⟩
        ∧ UNCHANGED ⟨aSxqu, aSxsSensor, aSxx⟩
        ∧ UNCHANGED hqu

ValueToAdapter(d, cE0) ≜
        ∧ cE0 = "02.actionCall"
        ∧ Append(currentTrace, cE0) ∈ setOfPossibleTraces
        ∧ currentTrace′ = Append(currentTrace, cE0)
        ∧ adapterxsAdapter = "init"
        ∧ adapterxsAdapter′ = "enqueued"
        ∧ adapterxqu′ = Append(adapterxqu, d)
        ∧ (proxyxss = "readyForTimeout" ∨ proxyxss = "readyForError" ∨
                      proxyxss = "readyForCall") ! readyForCall
        ∧ proxyxss′ = IF proxyxss = "readyForError"
                      THEN "calledError"
                      ELSE "called"
        ∧ UNCHANGED aSxqu
        ∧ UNCHANGED aSxsSensor
        ∧ UNCHANGED aSxx
        ∧ UNCHANGED proxyxqu
        ∧ UNCHANGED proxyxx
```

```
           ∧ UNCHANGED adapterxx
           ∧ UNCHANGED hqu

AdapterDequeue(cE0) ≜
       ∧ cE0 = "03.actionDequeue"
       ∧ Append(currentTrace, cE0) ∈ setOfPossibleTraces
       ∧ adapterxsAdapter = "enqueued"
       ∧ currentTrace′ = Append(currentTrace, cE0)
       ∧ adapterxsAdapter′ = "dequeued"
       ∧ adapterxx′ = Head( adapterxqu )
       ∧ adapterxqu′ = Tail( adapterxqu )
       ∧ UNCHANGED ⟨proxyxss, proxyxqu, proxyxx⟩
       ∧ UNCHANGED ⟨aSxqu, aSxsSensor, aSxx⟩
       ∧ UNCHANGED hqu

ValueToSensor(d, cE0) ≜
       ∧ aSxsSensor = "init"
       ∧ aSxsSensor′ = "enqueued"
       ∧ aSxqu′ = Append(aSxqu,d)
       ∧ adapterxsAdapter = "dequeued"
       ∧ adapterxsAdapter′ = "called"
       ∧ cE0 = "03.actionCall"
       ∧ Append(currentTrace, cE0) ∈ setOfPossibleTraces
       ∧ currentTrace′ = Append(currentTrace, cE0)
       ∧ UNCHANGED ⟨proxyxss, proxyxqu, proxyxx⟩
       ∧ UNCHANGED ⟨adapterxqu, adapterxx⟩
       ∧ UNCHANGED aSxx
       ∧ UNCHANGED hqu

SensorDequeue(cE0) ≜
       ∧ cE0 = "04.actionDequeue"
       ∧ Append(currentTrace, cE0) ∈ setOfPossibleTraces
       ∧ aSxsSensor = "enqueued"
       ∧ aSxsSensor′ = "dequeued"
       ∧ Append(currentTrace, cE0) ∈ setOfPossibleTraces
       ∧ currentTrace′ = Append(currentTrace, cE0)
       ∧ UNCHANGED ⟨proxyxx, proxyxqu, proxyxss⟩
       ∧ UNCHANGED ⟨adapterxx, adapterxsAdapter, adapterxqu⟩
       ∧ UNCHANGED ⟨aSxx, aSxqu⟩
       ∧ UNCHANGED hqu


ReturnFromAdapter(d, cE0) ≜
       ∧ adapterxsAdapter = "returnEnqueued"
       ∧ cE0 = "03.actionReturn"
       ∧ Append(currentTrace, cE0) ∈ setOfPossibleTraces
       ∧ d = Head( adapterxqu)
       ∧ currentTrace′ = Append(currentTrace, cE0)
       ∧ adapterxsAdapter′ = "init"
       ∧ adapterxx′ = 0
       ∧ adapterxqu′ = Tail(adapterxqu)
       ∧ proxyxqu′ = Append(proxyxqu, d)
       ∧ proxyxss′ = IF proxyxss = "called"
                     THEN "returnEnqueued"
                     ELSE IF proxyxss = "calledError"
                     THEN "returnDeqError"
                     ELSE "returnDeqTimeout"
       ∧ UNCHANGED proxyxx
       ∧ UNCHANGED aSxqu
       ∧ UNCHANGED aSxsSensor
       ∧ UNCHANGED aSxx
```

```
       ∧ UNCHANGED hqu

vars ≜ ≪ aSxsSensor, adapterxx, adapterxqu, aSxx, aSxqu,
adapterxsAdapter, proxyxx, proxyxqu, proxyxss, currentTrace ≫

Next ≜ ∨  ∃ cE0 ∈ E0 : ∨ ProxyDequeue(cE0)
                        ∨ ProxyProcess(cE0)
                        ∨ AdapterDequeue(cE0)
                        ∨ SensorDequeue(cE0)
                        ∨ SensorProcess(cE0)
                        ∨ setCall1(cE0)
                        ∨ setCall2(cE0)
                        ∨ ProxyError1(cE0)
                        ∨ ProxyError2(cE0)
                        ∨ ProxyError3(cE0)
                        ∨ ProxyError4(cE0)
                        ∨  ∃ d ∈ Data :
                                    ∨ ValueToProxy(d, cE0)
                                    ∨ ValueToAdapter(d, cE0)
                                    ∨ ValueToSensor(d, cE0)
                                    ∨ ReturnFromSensor(d, cE0)
                                    ∨ ReturnFromProxy(d, cE0)
                                    ∨ ReturnFromAdapter(d, cE0)

Liveness1 ≜ ∧ ∀ cE0 ∈ E0 :
                          ∧ WF_vars(ProxyDequeue(cE0))
                          ∧ WF_vars(ProxyProcess(cE0))
                          ∧ WF_vars(setCall1(cE0))
                          ∧ WF_vars(setCall2(cE0))
                          ∧ WF_vars(AdapterDequeue(cE0))
                          ∧ WF_vars(SensorDequeue(cE0))
                          ∧ WF_vars(SensorProcess(cE0))
                          ∧ WF_vars(ProxyError1(cE0))
                          ∧ WF_vars(ProxyError2(cE0))
                          ∧ WF_vars(ProxyError3(cE0))
                          ∧ WF_vars(ProxyError4(cE0))
                          ∧ ∀ d ∈ Data : ∧ WF_vars(ValueToAdapter(d, cE0))
                                         ∧ WF_vars(ValueToProxy(d, cE0))
                                         ∧ WF_vars(ReturnFromSensor(d, cE0))
                                         ∧ WF_vars(ReturnFromProxy(d, cE0))
                                         ∧ WF_vars(ReturnFromAdapter(d, cE0))
                                         ∧ WF_vars(ValueToSensor(d, cE0))

AP ≜ INSTANCE abstractProxy x ← ox, qu ←oqu, state ← ostate
```

Anhang E

Zeitbeweise

E.1 Hilfsvariablen der Aktion *Tick*

```
...
        h3′ = IF(enabled(SensorProcess) ∨
                 enabled(ReturnFromSensor) ∨
                 enabled(ReturnFromAdapter))
              THEN h3 + (now′ - now)
∧             ELSE h3
        h4′ = IF(enabled(ProxyError1) ∧ timerProxyError1 = 0)
              THEN h4 + h1 + (now′ - now)
              ELSE IF(enabled(ProxyError1) ∧ timerProxyError1 > 0)
              THEN h4 + (now′ - now)
∧             ELSE h4
        h5′ = IF(enabled(ProxyError2) ∧ timerProxyError2 = 0)
              THEN h1 + h2 + h3 + h5 + (now′ - now)
              ELSE IF(enabled(ProxyError2) ∧ timerProxyError2 > 0)
              THEN h5 + (now′ - now)
∧             ELSE h5
        h6′ = IF(enabled(ProxyError3) ∧ timerProxyError3 = 0)
              THEN h1 + h2 + h3 +  h6 + (now′ - now)
              ELSE IF(enabled(ProxyError3) ∧ timerProxyError3 > 0)
              THEN h6 + (now′ - now)
∧             ELSE h6
        h7′ = IF(enabled(ProxyDequeue))
              THEN h7 + (now′ - now)
∧             ELSE h7
        h8′ = IF(enabled(ProxyError4) ∧ timerProxyError4 = 0)
              THEN h1 + h2 + h3 + h7 + h8 (now′ - now)
              ELSE IF(enabled(ProxyError4) ∧ timerProxyError4 > 0)
              THEN h8 + (now′ - now)
∧             ELSE h8
```

Literaturverzeichnis

[ABB+00] Ahrendt, Wolfgang, Thomas Baar, Bernhard Beckert, Martin Giese, Elmar Habermalz, Reiner Hähnle, Wolfram Menzel und Peter H. Schmitt: *The KeY Approach: Integrating Object Oriented Design and Formal Verification.* In: Ojeda-Aciego, Manuel, Inma P. de Guzmán, Gerhard Brewka und Luís Moniz Pereira (Herausgeber): *Proc. 8th European Workshop on Logics in AI (JELIA), Malaga, Spain*, Band 1919 der Reihe *LNCS*, Seiten 21–36. Springer-Verlag, Oktober 2000. URL: `ftp://ftp.cs.chalmers.se/pub/users/reiner/jelia.ps.gz`.

[Abb02] Abbott, Doug: *Linux for Embedded and Real-Time Applications.* Newnes, 2002.

[ABI+07] Auerbach, J., D.F. Bacon, D.T. Iercan, C.M. Kirsch, H. Röck V.T. Rajan und R. Trummer: *Java Takes Flight: Time-Portable Real-Time Programming with Exotasks.* In: *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES).* ACM Press, 2007, 2007.

[AIS+77] Alexander, Christopher, Sara Ishikawa, Murray Silverstein, Max Jacobson und Ingrid Fiksdahl Kingand Schlomo Angel: *A pattern language: towns, buildings, construction.* Oxford University Press, 1977.

[AL88] Abadi, Martín und Leslie Lamport: *The Existence of Refinement Mappings.* Technischer Bericht 29, DEC Digital Systems Research Center, Palo Alto, August 1988. Research Report.

[AL90] Abadi, Martín und Leslie Lamport: *The Existence of Refinement Mappings.* Technischer Bericht 66, DEC Digital Systems Research Center, Palo Alto, Mai 1990. Research Report.

[AL91a] Abadi, Martín und Leslie Lamport: *The Existence of Refinement Mappings.* Theoretical Computer Science, 82(2):253–284, Mai 1991. http://www.research.digital.com/SRC/staff/lamport/bib.html.

[AL91b] Abadi, Martín und Leslie Lamport: *An Old-Fashioned Recipe for Real Time.* Technischer Bericht 91, DEC Digital Systems Research Center, Palo Alto, November 1991. Research Report, http://www.research.digital.com/ SRC/staff/lamport/bib.html.

[Arj07] Arjona, Jorge Ortega: *Design Patterns for Communication Components of Parallel Programs.* In: *EuroPLoP 2007.* Universitätsverlag Konstanz, 2007.

[AS85] Alpern, Bowen und Fred B. Schneider: *Defining Liveness.* Information Processing Letters, 21:181–185, 1985.

[BBAF+01] Berard, B., M. Bidoit, F. Laroussinie A. Finkel, A. Petit, L. Petrucci, Ph. Schnoebelen und P. McKenzie: *Systems and Software Verification.* Springer, 2001.

[BC87] Beck, Kent und Ward Cunningham: *Using Pattern Languages for Object-Oriented Programs.* Technischer Bericht, Apple Computer, Inc, 1987. Verfügbar via WWW: http://c2.com/doc/oopsla87.html.

[BCR03] Bacon, David F., Perry Cheng und V.T. Rajan: *A Real-time Garbage Collector with Low Overhead and Consistent Utilization.* In: *Conference Record of the Thirtieth ACM Symposium on Principles of Programming Languages*, 2003.

[BCS00] Bordeleau, F., J.P. Corriveau und B. Selic: *A Scenario-Based Approach for Hierarchical State Machine Design.* In: *3rd IEEE International Symposium on Object-Oriented Real-time Distributed Computing (ISORC'2000).* IEEE, 200.

[BE94] Balzer, Dieter und Ullrich Epple: *Standortsicherung für die Leit- und Automatisierungstechnik: Vorträge der GMA-Fachtagung anlässlich des VDE-Kongresses '94 am 19. und 20.´Oktober 1994 in München*, Kapitel Technologieinvariante Prozeßführungsmodelle, Seiten 83 – 90. VDE-Verlag, 1994.

[BHK04] Born, Marc, Echhardt Holz und Olaf Kath: *Softwareentwicklung mit UML 2.* Addison-Wesley, 2004.

[BHS07a] Beckert, Bernhard, Reiner Hähnle und Peter H. Schmitt (Herausgeber): *Verification of Object-Oriented Software: The KeY Approach.* LNCS 4334. Springer-Verlag, 2007.

[BHS07b] Buschmann, F., K. Henney und D. C. Schmidt: *Pattern-Oriented Software Architecture: On Patterns and Pattern Languages.* Wiley Sons, 2007.

[BMR+96] Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad und Michael Stal: *A System of Patterns.* Wiley, 1996.

[BMR+98] Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad und Michael Stal: *Pattern-orientierte Software-Architektur.* Addison-Wesley, 1998.

[Boc03] Bock, Conrad: *UML 2 Activity and Action Models Part 2: Actions.* Journal of Object Technology, 2(5):41 – 56, 2003. Verfügbar via WWW: http://www.jot.fm/issues/issue_2003_09/column4.

[BR07] Backschat, Martin und Bernd Rücker: *Enterprise Java Beans 3.0.* Elsevier, 2007.

[Bra93] Brack, Georg: *Automatisierungstechnik für Anwender.* Deutscher Verlag für Grundstoffindustrie, 1993.

[CDK05] Coulouris, George, Jean Dollimore und Tim Kindberg: *Verteilte Systeme. Konzepte und Design.* Pearson Studium, 2005.

[CGP00] Clarke, E. M., Orna Grumberg und Doron Peled: *Model Checking.* MIT Press, 2000.

[Chr00] Christensen, J.H.: *Basic Concepts of IEC 61499.* In: Döschner, Ch. (Herausgeber): *Fachtagung verteilte Automatisierung*, Seiten 55 – 62, 2000.

[CS02] Corsaro, Angelo und Douglas C. Schmidt: *Evaluating Real-Time Java Features and Performance for Real-time Embedded Systems.* In: *Proceedings of the 8th IEEE Real-Time Technology and Applications Symposium*, 2002.

[DAV95] DAVID, R.: *GRAFCET: A POWERFUL TOOL FOR SPECIFICATION OF LOGIC CONTROLLERS.* IEEE transaction on control systems technology, 3(3):253–368, 1995. Verfügbar via WWW: http://www.cs.utexas.edu/users/feixie/pubs/tbcr.pdf.

[DG08] DiPippo, Lisa und Christopher Gill (Herausgeber): *Design Patterns for Distributed Real-Time Embedded Systems.* Springer, 2008.

[DH04a] D. Harel, B. Rumpe: *Meaningful Modeling: Whats the Semantics of Semantics?* IEEE Computer, 37(10):64–72, 2004. availabale via WWW: www.sse.cs.tu-bs.de/publications/.

[DH04b] D. Harel, H. Kugler: *The Rhapsody Semantics of Statecharts (or On the Executable Core of the UML).* In: al., H. Ehrig et (Herausgeber): *3rd Intl Workshop Integration of Software Specification Techniques for Applications in Engineering*, Nummer 3147 in *LNCS*, Seiten 325–354. Springer, 2004.

[Dib02] Dibble, Peter C.: *Real-Time Java Platform Programming.* Prentice Hall, 2002.

[DL99] Diego Latella, Istvan Majzik, Mieke Massink: *Towards a formal operational semantics of UML statechart diagrams.* In: *FMOODS99, IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems.* Kluwer, 1999.

[DME09] Dig, Danny, John Marrero und Michael D. Ernst: *Refactoring sequential Java code for concurrency via concurrent libraries.* In: *ICSE*, Seiten 397–407, 2009.

[Dou99] Douglass, Bruce Powell: *Doing Hard Time.* Addison Wesley, 1999.

[Dou04] Douglass, Bruce Powel: *Real Time UML Third Edition.* Addison-Wesley, 2004.

[Dra00] Draack, Volker: *Ein Entwurfsmuster-System zur Steuerung einer chemietechnischen Anlage.* Diplomarbeit, Universität Dortmund, 2000.

[dRE98] Roever, Willem Paul de und Kai Engelhardt: *Data Refinement: Model-Oriented Proof Methods and their Comparison.* CAMBRIDGE TRACTS IN THEORETICAL COMPUTER SCIENCE 47. Cambrige University Press, 1998.

[Ech90] Echtle, Klaus: *Fehlertoleranzverfahren.* Springer, 1990.

[EH99] Eisenach, B. und H. Hennecke: *Erfahrungen mit einem Prozeßleitsystem Toolkit in einer Batchanwendung.* Automatisierungstechnische Praxis, Seiten 56 – 59, 1999.

[EKS92] Epple, Ulrich, Hellmut Kopec und Rüdiger Schmidt: *Strukturierung von Prozeßführungsaufgaben und Leitsystemsoftware.* Automatisierunstechnische Praxis, 34, 1992.

[Ens00] Enste, Udo: *Generische Entwurfsmuster in der Funktionsbausteintechnik und deren Anwendung in der operativen Prozeßführung.* Fortschritt Berichte VDI. VDI Verlag, 2000.

[Epp93] Epple, U.: *Leit- und automatisierungstechnische Einrichtungen: Zuverlässigkeit, Sicherheit und Qualität; Vorträge der GMA-Fachtagung vom 21. bis 22. Januar 1993 anläßlich des VDE-Jubiläumskongresses '93 in Berlin*, Kapitel Die Bausteintechnik - Grundlage einer objektorientierten Netzstruktur für die Prozeßleittechnik, Seiten 91 – 100. vde-Verlag, 1993.

[Esk99] Eskelin, Philip: *Component Interaction Patterns.* In: *PLoP'99*, 1999. Verfügbar via WWW: http://jerry.cs.uiuc.edu/ plop/plop99/proceedings/.

[FB04] Frank Buschmann, Kevlin Henney, Douglas C. Schmidt: *Paatern-Oriented Software Architecure–A Pattern Language for Distributed Computing*, Band 4. JOHN WILEY, 2004.

[Fow99] Fowler, Martin: *Analysemuster.* Addison-Wesley, 1999.

[FW04] Frank de Boeur und Wilhelm Paul de Roever: *A compositional operational Semantics of Java-MT.* In: Jean Bézivin, Pierre Alain Muller (Herausgeber): *Verification: Theory and Practice*, Band 2772 der Reihe *LNCS*, Seite 107119, 2004.

[GB98] Grady Booch, Ivar Jacobson, James Rumbaugh: *The Unified Modeling Language User Guide.* Addison-Wesley Pub Co, October 1998.

[GH02] Graw, Günter und Peter Herrmann: *Verification of xUML Specifications in the Context of MDA.* In: Bézivin, Jean und Robert France (Herausgeber): *Workshop in Software Model Engineering (WISMEUML'2002)*, Dresden, 2002.

[GH04a] Graf, Susanne und Jozef Hooman: *Correct Development of Embedded Systems.* In: Flavio Oquendeo, Brian WarBoys, Ron Morrison (Herausgeber): *European Workshop on Software Architecture: Languages, Styles, Models, Tools, and Applications (EWSA 2004)*, Nummer 3047 in *LNCS*, Seiten 241–249. Springer, 2004.

[GH04b] Graw, Günter und Peter Herrmann: *Transformation and Verification of Executable UML Models.* Electr. Notes Theor. Comput. Sci., 101:3–24, 2004.

[GHJV93] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Design Patterns: Abstraction and Reuse of Object-Oriented Design.* In: Nierstrasz, O. M. (Herausgeber): *7th European conference on Object oriented programming*, Kaiserslautern, 1993. Springer Verlag.

[GHK99a] Graw, Günter, Peter Herrmann und Heiko Krumm: *Composing object oriented specifications and verfications with cTLA.* In: Hüttel, H., J. Kleist, U. Nestmann und A. Ravara (Herausgeber): *2nd international Workshop on Semantics of Objects as Processes (SOAP99)*, 1999.

[GHK99b] Graw, Günter, Peter Herrmann und Heiko Krumm: *Constraint-Oriented Formal Modelling of OO-Systems.* In: *Second IFIP WG 6.1 International Working Conference on Distributed Applications and Interoperable Systems (DAIS 99)*, Seiten 345–358, Helsinki, Juni 1999. Kluwer Academic Publisher.

[GHK00] Graw, Günter, Peter Herrmann und Heiko Krumm: *Verification of UML-based real-time system designs by means of cTLA.* In: *Proceedings of the 3rd IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC2K)*, Seiten 86–95, Newport Beach, 2000. IEEE Computer Society Press.

[GM93] Gardiner, Paul H. B. und Carroll Morgan: *A Single Complete Rule for Data Refinement.* Formal Asp. Comput., 5(4):367–382, 1993.

[GPP98] Gogolla, Martin und Francesca Parisi-Presicce: *State Diagrams in UML – A Formal Semantics using Graph Transformation.* In: Broy, Manfred, Derek Coleman, Thomas Maibaum und Berhard Rumpe (Herausgeber): *Proceedings of the ICSE'98 Workshop on Precise Semantics of Modeling Techniques (PSMT'98)*, Nummer TUM-I9803, 1998.

[Gra93] Graw, Günter: *Interpretationen von Spezifikationen verteilter Systeme in TLA+.* Diplomarbeit, Universität Dortmund, Informatik IV, D-44221 Dortmund, September 1993.

[Gra99a] Grand, Mark: *Transaction Patterns.* In: *Pattern Languages of Programm Design PLOP 99*, 1999. Verfügbar via WWW: http://jerry.cs.uiuc.edu/ plop/plop99/proceedings/.

[Gra99b] Graw, Günter: *Refining Analysis Patterns into correct control software.* In: Hofmann, P. und A. Schürr (Herausgeber): *Proceedings of GI Workshop Object-Oriented Modeling of Embedded Realtime Systems (OMER)*, Seiten 113 – 118. Department of Computer Science, University Bw Munich, 1999.

[GTB+03] Giese, H., M. Tichy, S. Burmester, W. Schäfer und S. Flake: *Towards the Compositional Verification of Real-Time UML Designs.* In: *European Software Engineering Conference (ESEC)*, 2003.

[Gut97] Guttag, John V.: *Abstract data types and the development of data structures.* Communications of the ACM, 20(6):396–404, 1997.

[GV08] Grumberg, Orna und Helmut Veith (Herausgeber): *25 Years of Model Checking.* Springer, 2008.

[GVZ05] Gruschko, Boris, Friedrich H. Vogt und Simon Zambrovski: *Enabling the usage of formal methods by creation of convenient tools.* In: *2nd International Workshop on Web Services and Formal Methods*, Versailles, France, 2005.

[Hal08] Halang: *Distributed Embedded Control Systems.* Springer, 2008.

[Har87] Harel, David: *Statecharts: A Visual Formalism for Complex Systems.* Science of Computer Programming, 8(3):231 – 274, June 1987.

[Hau01] Haugen, Øystein: *MSC-2000 Interaction Diagrams for the New Millennium.* Computer Networks, 35(6):721–732, 2001.

[Her98] Herrmann, Peter: *Problemnaher korrektheitssichernder Entwurf von Hochleistungsprotokollen.* Deutscher Universitätsverlag, 1998.

[Hey95] Heyl, Carsten: *Werkzeugunterstützung des kompositionalen Entwurfs verteilter Systeme in TLA.* Diplomarbeit, Universität Dortmund, Informatik IV, D-44221 Dortmund, 1995.

[HF06] Harel, David und Yishai Feldmann: *Algorithmik.* Springer, 2006.

[HGK98] Herrmann, Peter, Günter Graw und Heiko Krumm: *Compositional Specification and Structured Verification of Hybrid Systems in cTLA.* In: *Proceedings of the 1st IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC98)*, Seiten 335–340, Kyoto, April 1998. IEEE Computer Society Press.

[HK94a] Herrmann, Peter und Heiko Krumm: *Compositional Specification and Verification of High-Speed Transfer Protocols.* In: Vuong, Son T. und Samuel T. Chanson (Herausgeber): *Protocol Specification, Testing, and Verification XIV*, Seiten 339–346, Vancouver, B.C., Canada, 1994. IFIP, Chapman & Hall.

[HK94b] Herrmann, Peter und Heiko Krumm: *Compositional Specification and Verification of High-Speed Transfer Protocols.* Research report 540, Universität Dortmund, D-44221 Dortmund, Germany, Februar 1994. Available via WWW: http://ls4-www.informatik.uni-dortmund.de/RVS/puliste.html.

[HK95] Herrmann, Peter und Heiko Krumm: *Re-Usable Verification Elements for High-Speed Transfer Protocol Configurations.* In: Dembiński, Piotr und Marek Średniawa (Herausgeber): *Protocol Specification, Testing, and Verification XV*, Seiten 171–186, Warsaw, Poland, 1995. IFIP, Chapman & Hall.

[HK97a] Herrmann, Peter und Heiko Krumm: *Kompositionale Constraints hybrider Systeme.* In: Schnieder, E. und D. Abel (Herausgeber): *Entwurf komplexer Automatisierungssysteme*, Seiten 243–264, Braunschweig, 1997.

[HK97b] Herrmann, Peter und Heiko Krumm: *Specification of Hybrid Systems in cTLA+.* In: *Proceedings of the 5th International Workshop on Parallel & Distributed Real-Time Systems (WPDRTS'97)*, Seiten 212–216, Geneva, Switzerland, 1997. IEEE Computer Society Press.

[HK99] Halang, Wolfgang A. und Rudolf Konakovsky: *Sicherheitsgerichtete Echtzeitsysteme.* Oldenbourg Verlag, 1999.

[HK00] Herrmann, Peter und Heiko Krumm: *A Framework for Modeling Transfer Protocols.* Computer Networks, 34(2):317–337, 2000.

[HK05] Hitz, Martin und Gerti Kappel: *UML@Work: Von der Analyse zur Realisierung.* dpunkt-Verlag, Heidelberg, 2005.

[HKMM02] Henzinger, Thomas A., Christoph M. Kirsch, Rupak Majumdar und Slobodan Matic: *Time-Safety Checking for Embedded Programs.* In: *Proceedings of the Second International Workshop on Embedded Software.* Lecture Notes in Computer Science, Springer-Verlag, 2002.

[HKSP02] Henzinger, Thomas A., Christoph M. Kirsch, Marco A.A. Sanvido und Wolfgang Pree: *From Control Models to Real-Time Code Using Giotto.* In: *Proceedings of the Second International Workshop on Embedded Software.* Lecture Notes in Computer Science, Springer-Verlag, 2002.

[HM03] Hoare, Tony und Misra: *The grand challenge Project.* 2003. Verfügbar via WWW: http://www.jot.fm/issues/issue_2003_11/column1.

[Hog89] Hogreve, Dieter: *Estelle, LOTOS und SDL: Standard-Spezifikationssprachen für verteilte Systeme.* Springer, 1989.

[Hol08] Holzmann, Gerard J.: *A Stack-Slicing Algorithm for Multi-Core Model Checking.* Electron. Notes Theor. Comput. Sci., 198(1):3–16, 2008, ISSN 1571-0661.

[HS03] Haugen, Øystein und Ketil Stølen: *STAIRS Steps to analyze interactions with refinement semantics.* In: *Proc. Sixth International Conference on UML (UML'2003)*, Nummer 2863 in *LNCS*, Seiten 388 – 402, 2003.

[HS08] Herlihy, Maurice und Mir Shavit: *The Art of Multiprocessor Programming.* Morgan Kaufmann, 2008.

[HvdZ05] Hooman, Jozef und Mark v. d. Zwaag: *A Semantics of Communicating Reactive Objects with Timing.* Journal on Software Tools for Technology Transfer, 2005.

[HW90] Herlihy, Maurice und Jeannette M. Wing: *Linearizability: A Correctness Condition for Concurrent Objects.* ACM Trans. Program. Lang. Syst., 12(3):463–492, 1990.

[ID96] Islam, Nayeem und Murthy V. Devarakonda: *An Essential Design Pattern for Fault-Tolerant Distributed State Sharing.* Communications of the ACM (CACM), 39(10):65–74, 1996.

[J C00] J Consortiums Real-Time Java Working Group: *Real-Time Core Extensions.* Technischer Bericht, INTERNATIONAL J CONSORTIUM, 2000. Verfügbar via WWW: www.j-consortium.org.

[Jac82] Jackson, Michael: *Software Development as an Engineering Problem.* Angewandte Informatik, 24(2):96–103, 1982.

[JKS91] Järvinen, H. M. und R. Kurki-Suonio: *DisCo specification language: marriage of actions and objects.* In: *Proceedings of 11th International Conference of Distributed Computing Systems*, Seiten 142–151. IEEE Computer Society Press, 1991.

[JLHM91] Jaffe, Matthew S., Nancy G. Leveson, Mats Per Erik Heimdahl und Bonnie E. Melhart: *Software Requirements Analysis for Real-Time Process-Control Systems.* IEEE Trans. Software Eng., 17(3):241–258, 1991.

[JRH+03] Jeckle, M., C. Rupp, J. Hahn, B. Zengler und S. Queins: *UML2 glasklar.* Hanser, 2003.

[Kai99] Kaindl, Herrmann: *Difficulties in the Transition from OO Analysis to Design.* IEEE Software, Seiten 94–101, 1999.

[KBH07] Kraemer, Frank Alexander, Rolv Bræk und Peter Herrmann: *Synthesizing Components with Sessions from Collaboration-Oriented Service Specifications.* In: Emmanuel Gaudin, Elie Najm, Rick Reed (Herausgeber): *SDL 2007*, Band 4745 der Reihe *Lecture Notes in Computer Science*, Seiten 166–185. Springer–Verlag Berlin Heidelberg, 2007.

[KGC07] Konrad, Sascha, Heather Goldsby und Betty H.C. Cheng: *i2MAP An Incremental and Iterative Modeling and Analysis Process.* In: *Proceedings of the ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS 2007)*, Nashville, TN, October 2007.

[KH07a] Kraemer, Frank Alexander und Peter Herrmann: *Formalizing Collaboration-Oriented Service Specifications using Temporal Logic.* In: *Networking and Electronic Commerce Research Conference 2007 (NAEC 2007)*, 2007.

[KH07b] Kraemer, Frank Alexander und Peter Herrmann: *Transforming Collaborative Service Specifications into Efficiently Executable State Machines.* In: Ehring, Karsten und Holger Giese (Herausgeber): *Proceedings of the 6th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007)*, Band 7, 2007.

[KHB06] Kraemer, Frank Alexander, Peter Herrmann und Rolv Bræk: *Aligning UML 2.0 State Machines and Temporal Logic for the Efficient Execution of Services.* In: R. Meersmann, Z. Tari (Herausgeber): *Proceedings of the 8th International Symposium on Distributed Objects and Applications (DOA), 2006, Montpellier, France*, Band 4276 der Reihe *Lecture Notes in Computer Science*, Seiten 1613–1632. Springer–Verlag Heidelberg, 2006.

[KL99] Kevin Lano, Juan Bicarregui: *Semantics and transformations for UML models.* In: Bézivin, Jean und Pierre Alain Muller (Herausgeber): *The Unified Modeling Language, UML98 - Beyond the Notation. First International Workshop*, Band 1618 der Reihe *LNCS*, Seite 107119, 1999.

[KSH07] Kraemer, Frank Alexander, Vidar Slåtten und Peter Herrmann: *Engineering Support for UML Activities by Automated Model-Checking.* In: *4th International Workshop on Rapid Integration of Software Engineering Techniques (RISE 2007) 2007 (Rise 2007)*, 2007.

[KSJ88] Kurki-Suonio, R. und H. M. Järvinen: *Action system approach to the specification and design of distributed systems.* ACM Transactions on Programming Languages and Systems, 4(10):510–554, Oktober 1988.

[KSK03] Krishna, Arvind, Douglas C. Schmidt und Raymond Klefstad: *Enhancing Real-Time CORBA via Real-Time Java.* In: *submitted to the 24th IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2003.

[KSKC03] Krishna, Arvind, Douglas C. Schmidt, Raymond Klefstad und Angelo Corsaro: *Towards Predictable Real-time Java Object Request Brokers.* In: *Proceedings of the 9th IEEE Real-time/Embedded Technology and Applications Symposium (RTAS)*, 2003.

[KWB03] Kleppe, A., J. Warmer und W. Bast: *MDA Explained.* Addison-Wesley, 2003.

[Lam91a] Lamport, Leslie: *Introducing TLA+.* to appear, preliminary version, DEC Digital Systems Research Center, Palo Alto, September 1991. Research Report.

[Lam91b] Lamport, Leslie: *The Temporal Logic of Actions.* Technischer Bericht Research Report 79, DEC Digital Systems Research Center, Palo Alto, Mai 1991. To appear in *ACM Transactions on Programming Languages and Systems.*

[Lam92] Lamport, Leslie: *TLA+: Syntax and Semantics.* to appear, preliminary version, DEC Digital Systems Research Center, Palo Alto, Februar 1992. Research Report.

[Lam94] Lamport, Leslie: *The Temporal Logic of Actions.* ACM Transactions on Programming Languages and Systems, 16(3):872–923, Mai 1994.

[Lam95a] Lamport, Leslie: *On visual TLA.* Technischer Bericht 91, DEC Digital Systems Research Center, Palo Alto, November 1995. Research Report, http://www.research.digital.com/ SRC/staff/lamport/bib.html.

[Lam95b] Lamport, Leslie: *TLA$^+$.* Research Note, DEC Digital Systems Research Center, Palo Alto, Juli 1995. http://www.research.digital.com/ SRC/personal/Leslie_Lamport/tla/papers.html.

[Lam03] Lamport, Leslie: *Specifying Systems.* Addison-Wesley, 2003.

[Lam05] Lamport, Leslie: *Real Time is Really Simple.* In: *Proceedings of Charme 2005*, 2005.

[Lau89] Lauber, Rudolf: *Prozeßautomatisierung.* Springer, 1989.

[LB03] Lamport, Leslie und Brannon Batson: *High-Level Specifications: Lessons from Industry.* In: Boer, F. S. de, Marcello M. Bonsangue, Susanne Graf und Willem Paul de Roever (Herausgeber): *Formal Methods for Components and Objects*, Nummer 2863 in *LNCS*, Seiten 242–262. Springer, 2003.

[Lea00] Lea, Doug: *Patterns-Discussion FAQ.* http://gee.cs.oswego.edu/dl/pd-FAQ/pd-FAQ.html, 2000.

[Lev95] Leveson, Nancy: *Safeware.* Addison-Wesley, 1995.

[LKK93] Lockemann, P., G. Krüger und H. Krumm: *Telekommunikation und Datenhaltung.* Hanser, 1993.

[LP99] Lilius, Johan und Ivan Porres Paltor: *Formalising UML State Machines for Model Checking.* In: France, Robert und Bernhard Rumpe (Herausgeber): *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, Band 1723 der Reihe *LNCS*, Seiten 430–445. Springer, 1999.

[LT90] Lee, P.A. und T.Anderson: *Fault Tolerance.* Springer, 1990.

[Mar07] Marwedel, Peter: *Eingebettete Systeme.* Springer, 2007.

[Mar09] Marek, Viktor W.: *Introduction to the mathematics of satisfyability.* CRC Studies, 2009.

[MB02] Mellor, Stephen J. und Marc J. Balcer: *Executable UML: A Foundation for Model Driven Architecture.* Addison-Wesley, 2002.

[ME99] Morr, Wolfgang und Bernd Eisenach: *Anwendungsorientierte Standardisierung der Funktionsbausteine und der Bedienoberflächen von Leitsystemen in einem Toolkit.* Automatisierungstechnische Praxis, 41(97):56–60, 1999.

[Mes02] Mester, Arnulf: *Rechnergestützte Konstruktion verteilter Anwendungen mit Spezifikationsmustern.* Dissertation, Universität Dortmund, 2002.

[Mey00] Meyer, D.: *Innovation leittechnischer Softwarestrukturen.* In: Döschner, Ch. (Herausgeber): *Fachtagung verteilte Automatisierung*, Seiten 237 – 244, 2000.

[Mis99] Misch, Oliver: *Musterbasierter Entwurf einer verteilten Fertigungszelle.* Diplomarbeit, Universität Dortmund, 1999.

[MK04] Michael Kirchner, Prashant Jain: *Pattern-Oriented Software Architecure–Patterns for Resource Management*, Band 3. JOHN WILEY, 2004.

[MLD09] Mahnke, Wolfgang, Stefan Helmut Leitner und Matthias Damm: *OPC Unified Architecture*. Springer, 2009.

[MP05] Mukul Prasad, Armin Biere, Aarti Gupta: *A Survey of Recent Advances in SAT-based Formal Verification*. Intl. Journal on Software Tools for Technology Transfer (STTT), 7(2), 2005.

[MTAL98] Mellor, Stephen J., Steve Tockey, Rodolphe Arthaud und Philippe LeBlanc: *Software-platform-independent, Precise Action Specifications for UML*. In: Bézivin, Jean und Pierre Alain Muller (Herausgeber): *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998*, Seiten 281–286, 1998.

[MTAL99] Mellor, Stephen J., Stephen R. Tockey, Rodolphe Arthaud und Philippe LeBlanc: *An Action Language for UML: Proposal for a Precise Execution Semantics*. In: Bézivin, Jean und Pierre Alain Muller (Herausgeber): *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998, Selected Papers*, Band 1618 der Reihe *LNCS*, Seiten 307–318. Springer, 1999.

[Neu00] Neumann, P.: *Triebkräfte für die Entwicklung von Automatisierungssystemen*. In: Döschner, Ch. (Herausgeber): *Fachtagung verteilte Automatisierung*, Seiten 193 – 205, 2000.

[NGLS98] Neumann, Peter, Eberhard Grötsch, Christoph Lubkoll und René Simon: *SPS-Standard: IEC 1131*. Oldenbourg Verlag, 1998.

[Nil98] Nilsen, Kelvin: *Adding Real-Time Capabilities to Java*. CACM, 31(6):49–56, 1998.

[Nip86] Nipkov, Tobias: *Non-deterministic data types: Models and implementations*. Acta Informatica, 22(16):629–661, 1986.

[OMG03] OMG: *UML: Superstructure v. 2.0 – Third revised UML 2.0 Superstructure Proposal*, omg doc# ad/03-04-01 Auflage, 2003. Verfügbar via WWW: www.u2-partners.org/uml2-proposals.htm.

[OPR96] Otte, Randy, Paul Patrick und Mark Roy: *CORBA*. Prentice Hall, 1996.

[OW04] Oaks, S. und H. Wong: *Java Threads*. O'Reilly, 2004.

[Pet00] Petig, M.; Riedl, M.: *Funktionsblocktechnologie und IEC 61499 im modularen Maschinenbau*. In: Döschner, Ch. (Herausgeber): *Fachtagung verteilte Automatisierung*, Seiten 96 – 102, 2000.

[Pol94] Polke, M.: *Prozeßleittechnik*. Oldenbourg Verlag, 1994.

[Pon06] Pons, Claudia: *Heuristics on the Definition of UML Refinement Patterns*. In: *SOFSEM*, Seiten 461–470, 2006.

[PSW96] Popien, Claudia, Gerd Schürmann und Karl Heinz Weiß: *Verteilte Verarbeitung in offenen Systemen*. Teubner Verlag, 1996.

[QC99] Quibeldey-Cirkel, K.: *Entwurfsmuster*. Springer, 1999.

[RACH00] Reggio, G., E. Astesiano, C. Choppy und H. Husmann: *Analysing UML Active Classes and Associated Statecharts - A Lightweight Formal Approach*. In: *Proceedings FASE 2000 - Fundamental Approaches to Software Engineering*, LNCS 1783, 2000.

[Rie00] Rieger, P.: *Funktionsblocktechniken im Wandel - eine progressive Betrachtung.* In: Döschner, Ch. (Herausgeber): *Fachtagung verteilte Automatisierung*, Seiten 32 – 39, 2000.

[RK03] Rothmaier, Gerrit und Heiko Krumm: *cTLA 2003 Description.* Technischer Bericht, Universität Dortmund, 2003. Verfügbar via WWW: http://www4.cs.uni-dortmund.de/RVS/MA/hk/cTLA2003description.pdf.

[RKK05] Rothmaier, Gerrit, Tobias Kneiphoff und Heiko Krumm: *Using SPIN and Eclipse for Optimized High-Level Modeling and Analysis of Computer Network Attack Models.* In: *Proceedings of 12th International SPIN Workshop on Model Checking of Software (SPIN 2005)*, Seiten 221–235. Springer-Verlag, 2005.

[RZ96] Riehle, Dirk und Heinz Züllighoven: *Understanding and Using Patterns in Software Development.* Theory and Practice of Object Systems, 2(1):3–13, 1996.

[Sch02] Schmidt, Douglas C.: *Overview Corba Real-Time Research.* Technischer Bericht, Verfügbar via WWW: http://www.cs.wustl.edu/ schmidt/corba-research-realtime.html, 2002.

[Sei03] Seitz, Matthias: *Speicherprogrammierbare Steuerungen.* Hanser, 2003.

[Sha03] Shankar, Nataniel: *The verified Software Roadmap.* 2003. Verfügbar via WWW: http://www.jot.fm/issues/issue_2003_11/column1.

[Sie08] Siebert, Fridtjof: *JEOPARD: Java environment for parallel real-time development.* In: *JTRES*, Seiten 87–93, 2008.

[SKM01] Schäfer, Timm, Alexander Knapp und Stephan Merz: *Model Checking UML State Machines and Collaborations.* In: *CAV 2001 Workshop on Software Model Checking. Paris, France*, Band ENTCS 55(3), 2001.

[SSRB00] Schmidt, Douglas, Michael Stal, Hans Rohnert und Frank Buschmann: *PATTERN-ORIENTED SOFTWARE ARCHITECTURE.* Wiley, 2000.

[SSS09] Szczepanski, Thorsten, René Simon und Sören Scharf: *Transaktionssichere Feldgeräte.* ATP Automatisierungstechnische Praxis, 51(4):46–54, 2009.

[Stö05] Störrle, Harald: *UML 2 für Studenten.* Pearson Studium, 2005.

[Szy00] Szyperski, Clemens: *Components and the Way Ahead*, Kapitel 1, Seiten 1–20. Cambridge University Press, 2000.

[TABP07] Titzer, Ben L., Joshua Auerbach, David F. Bacon und Jens Palsberg: *The ExoVM System for Automatic VM and Application Reduction.* In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007.

[Tai07] Taibi, Toufik (Herausgeber): *Design Pattern formalization techniques.* ICI Global, 2007.

[Tec09] Technical Committee TC4 Communications: *PLCopen and OPC Foundation: OPC UA Information Model for IEC 61131-3.* Technischer Bericht, PLCopen, 2009. Verfügbar via WWW: www.plcopen.org.

[THNMN09] Taibi, Toufik, Ángel Herranz-Nieva und Juan José Moreno-Navarro: *Stepwise Refinement Validation of Design Patterns Formalized in TLA+ using the TLC Model Checker.* Journal of Object Technology, 8(2):137–161, 2009.

[TN03] Taibi, T. und D. C. L. Ngo: *Formal Specification of Design Patterns A Balanced Approach.* Journal of Object Technology, 2(4):pp. 127–140, 2003.

[UG07] Uslar, Mathias und Fabian Grüning: *Zur semantischen Interoperabilität in der Energiebranche: CIM IEC 61970.* Wirtschaftsinformatik, 49(4):295–303, 2007.

[Unb92] Unbehauen, Heinz: *Regelungstechik I.* Vieweg Verlag, 1992.

[VBBB09] Verstoep, K., H. Bal, J. Barnat und L. Brim: *Efficient Large-Scale Model Checking.* In: *23rd IEEE International Parallel Distributed Processing Symposium (IPDPS 2009).* IEEE, 2009.

[WA97] W. Ahrens, H.-J. Scheurlen, G. U. Spohr: *Informationsorientierte Leittechnik.* Oldenbourg, 1997.

[Web97] Weber, Matthias: *Systematic Design of Embedded Control.* Oldenbourg Verlag, 1997.

[Wel04] Wellings, Andy: *Concurrent and Real-Time Programming in Java.* Wiley, 2004.

[Wir71] Wirth, Niklaus: *Program Development by Stepwise Refinement.* Communications of the ACM, 14(4):221–227, 1971.

[WSA07] Winkels, Ludger, Tanja Schmedes und Hans Jürgen Appelrath: *Dezentrale Energiemanagementsysteme.* Wirtschaftsinformatik, 49(5):386–390, 2007.

[YML99] Yu, Y., P. Manolios und L. Lamport: *Model Checking TLA+ Specifications.* In: Pierre, L. und T. Kropf (Herausgeber): *Correct Hardware Design and Verification Methods (CHARME '99)*, Lecture Notes in Computer Science 1703, Seiten 54–66. Springer-Verlag, 1999.

[Zdu07] Zdun, Uwe: *Systematic pattern selection using pattern language grammars and design space analysis.* Softw. Pract. Exper., 37(9):983–1016, 2007, ISSN 0038-0644.