Formale, semantische Fundierung und eine darauf abgestützte Verifikationsmethode für SDL

Ursula Hinkel

Fakultät für Informatik
der Technischen Universität München

Formale, semantische Fundierung und eine darauf abgestützte Verifikationsmethode für SDL

Ursula Hinkel

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. B. Radig

Prüfer der Dissertation:

1. Univ.-Prof. Dr. M. Broy
2. Univ.-Prof. Dr.-Ing. J. Eberspächer

Die Dissertation wurde am 21.04.1998 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 16.07.1998 angenommen.

Zusammenfassung

SDL (Specification and Description Language) ist eine in der Industrie weit verbreitete und von der ITU-T standardisierte Spezifikationssprache. Sie ist auf die Modellierung von ereignisgesteuerten und interaktiven verteilten Systemen ausgerichtet, die nebenläufige Aktivitäten und asynchrone Kommunikation über Nachrichtenaustausch aufweisen.

Bei SDL handelt es sich um eine informelle Spezifikationssprache. Den überwiegend graphischen Sprachmitteln steht eine nur ungenügend definierte informelle Semantik gegenüber. Dies hat zur Folge, daß die Bedeutung von SDL-Spezifikationen oft unklar und widersprüchlich ist, was vor allem für das Zeitkonzept von SDL gilt. Darüber hinaus ist mangels einer formalen Semantik, die auf mathematisch-logischen Konzepten basiert, die Durchführung von Verifikationsaufgaben nicht möglich.

In dieser Arbeit wird eine formale Semantik für SDL und darauf aufbauend eine Methode für die Verifikation von SDL-Spezifikationen entwickelt, so daß SDL zukünftig als formale Spezifikationssprache im Systementwicklungsprozeß eingesetzt werden kann. Der Semantikdefinition geht zunächst eine eingehende Analyse von SDL voraus, wobei Schwachpunkte von SDL aufgedeckt und Lösungsvorschläge entwickelt werden. Die formale Semantik für SDL wird basierend auf Strömen und stromverarbeitenden Funktionen im Rahmen der formalen Entwicklungsmethodik FOCUS definiert. FOCUS gibt einen eindeutigen, mathematisch-logischen Rahmen für die Systementwicklung verteilter, reaktiver Systeme sowie eine Reihe wohldefinierter Beschreibungstechniken und integrierter Verifikationskonzepte vor. Für statische und dynamische SDL-Systeme wird die jeweils adäquate mathematisch-logische Variante von FOCUS für die Semantikdefinition gewählt. Am Beispiel des Alternating Bit Protokolls wird basierend auf der formalen SDL-Semantik eine Methode für die Verifikation von SDL-Spezifikationen entwickelt, die auf den in FOCUS zur Verfügung stehenden Beweistechniken aufbaut.

Danksagung

Mein besonderer Dank gilt Manfred Broy für die Betreuung meiner Arbeit, vor allem für seine fachliche Anleitung und konstruktive Anmerkungen zu Vorversionen dieser Arbeit. Jörg Eberspächer danke ich für die Übernahme des zweiten Gutachtens und die Durchsicht einer Vorversion meiner Arbeit.

Bernhard Schätz, Katharina Spies und Ketil Stølen bin ich zu großem Dank verpflichtet. Sie haben mir durch ihren fachlichen Rat und viele wertvolle Diskussionen entscheidend bei der Erstellung dieser Arbeit geholfen. Für das kritische Lesen einer vollständigen Vorversion dieser Arbeit danke ich Katharina Spies und Michael Haubner. Bei Birgit Schieder bedanke ich mich für die sorgfältige Durchsicht der Beweise. Zahlreiche weitere Kollegen haben mich bei meiner Arbeit unterstützt, sei es durch interessante Diskussionen oder durch Kommentare zu Teilen von Vorversionen. Dank hierfür gilt Oscar Slotosch, Radu Grosu, Jan Philipps, David von Oheimb, Olaf Müller, Georg Zeis und Øystein Haugen.

Inhaltsverzeichnis

Kapitel 1

Einleitung und Überblick

1.1 Motivation

Die nachweisbare Korrektheit von Softwaresystemen ist ein wesentliches Ziel im Softwareentwicklungsprozeß. Fehler, die erst in späten Entwicklungsphasen erkannt werden, führen zu einem kostspieligen Reengineering und können, sofern sich das entwickelte System bereits im Einsatz befindet, eine Gefahr für menschliches Leben und Umwelt darstellen ([Lev95, BS93, Rei97]). Formale Methoden, die im Forschungsbereich der theoretischen Informatik entwickelt werden, umfassen formale Beschreibungstechniken für die Modellierung verschiedener Systemsichten und methodische Richtlinien für den Systementwicklungsprozeß. Mit ihrer präzisen Semantik bieten sie die Möglichkeit, Eigenschaften eines Systems zu verifizieren und somit auf mathematisch-logischer Basis nachzuweisen. Allerdings ergeben sich gerade wegen dieser Basis komplexe, nur schwer zugängliche Darstellungstechniken, so daß Anwender aus der Industrie den formalen Methoden sehr skeptisch gegenüberstehen – der Einsatz formaler Methoden stößt auf Akzeptanzprobleme.

Bei den Methoden, die in der Industrie bei der Systementwicklung eingesetzt werden, handelt es sich überwiegend um sogenannte pragmatische Methoden. Bei diesen steht den anschaulichen, meist graphischen Beschreibungstechniken eine nur ungenügend definierte formale Semantik gegenüber, so daß die Bedeutung von Spezifikationen oft unklar und widersprüchlich ist. Außerdem fehlt in der Regel die mathematisch-logische Grundlage für die Verifikation der erstellten Spezifikationen.

Ziel der aktuellen Forschung ist es deshalb, die Stärken von formalen und pragmatischen Softwaremethoden zu vereinen. Auf diese Weise entstehen Methoden, die einerseits in der Industrie akzeptiert und tatsächlich eingesetzt werden und andererseits einen durchgängig verifizierbaren Systementwicklungsprozeß unterstützen ([Huß97, Rum96, SHB96, BHH+97, BGH+97]).

Eine in der Industrie weit verbreitete pragmatische Spezifikationssprache ist SDL (Specification and Description Language, [IT93b]), eine von der ITU-T[1] standardisierte und

[1] Internationales Standardisierungsgremium für den Bereich Telekommunikation, ehemals CCITT

empfohlene Beschreibungssprache. SDL ist auf die Modellierung von ereignisgesteuerten und interaktiven verteilten Systemen ausgerichtet, die durch nebenläufige Aktivitäten und asynchrone Kommunikation mittels Nachrichtenaustausch charakterisiert sind. Mit SDL können die Struktur, das Verhalten und der Datenanteil eines Systems spezifiziert werden, wobei intuitiv erfaßbare, graphische Darstellungstechniken zur Verfügung stehen.

SDL wurde in den 70er Jahren in der Telekommunikationsindustrie für die Spezifikation von Kommunikationssystemen entwickelt und wird zunehmend auch in anderen Anwendungsbereichen eingesetzt. Ziel war es, eine Sprache zu entwerfen, die bei den Anwendern Akzeptanz findet. Bei der Standardisierung von SDL sind praktische Erfahrungen und Vorstellungen von Anwendern aus der Telekommunikationsindustrie eingeflossen – der akademische Einfluß war dabei sehr gering. Die Syntax und eine informelle Beschreibung von SDL sind in dem sehr technisch gehaltenen Standardisierungsdokument Z.100 der ITU-T ([IT93b]) enthalten. Erst Ende der 80er Jahre wurde eine Semantikdefinition in das Dokument Z.100 integriert – seitdem gilt SDL als formale Spezifikationssprache und wird auch von führenden Werkzeugherstellern und Anwendern als formal bezeichnet.

Dieser Bezeichnung hält SDL jedoch nur auf den ersten Blick stand. Bei genauerer Betrachtung wird deutlich, daß das ausschlaggebende Kriterium für eine formale Sprache – die auf mathematisch-logischen Konzepten basierende formale Semantik – mit der Semantikdefinition der ITU-T nicht gegeben ist. Dies ist der entscheidende Schwachpunkt von SDL, da damit die Bedeutung von SDL-Spezifkationen nur informell festgelegt und darüber hinaus die Durchführung von formalen Verifikationsaufgaben nicht möglich ist. Dies führt dazu, daß SDL-Anwender das Verhalten von SDL-Spezifikationen unterschiedlich interpretieren. So existieren beispielsweise unter den SDL-Anwendern, die sich zum Teil auch mit der Standardisierung und Weiterentwicklung von SDL beschäftigen, mehrere, zum Teil widersprüchliche Auffassungen, wie das zeitliche Verhalten in einer SDL-Spezifikation zum Ausdruck kommt. SDL wird häufig als Sprache für Echtzeitsysteme bezeichnet. Für die Spezifikation von sogenannten harten Echtzeitanforderungen ([TBYS96]), deren Verletzung zu schwerwiegenden Auswirkungen führt, ist SDL sicher nicht geeignet, wie bereits in anderen Arbeiten ([Leu95, BB91]) gezeigt wurde. Aber selbst für die Spezifikation sogenannter weicher Zeiteigenschaften, wie sie überwiegend im Bereich der Telekommunikationssysteme vorliegen, ist das Zeitkonzept mangelhaft und wird von SDL-Anwendern je nach Anwendungsfall unterschiedlich ausgelegt.

Eine formale Methodik, die im Gegensatz zu SDL die Bezeichnung „formal“ zu Recht trägt, ist FOCUS ([BS98, BDD+93]). FOCUS gibt einen eindeutigen, mathematisch-logischen Rahmen für die Systementwicklung verteilter, reaktiver Systeme sowie eine Reihe wohldefinierter Beschreibungstechniken und Verifikationskonzepte vor. FOCUS ist im universitären Bereich entstanden und stößt, wie andere akademische Ansätze auch, auf Akzeptanzprobleme in der Industrie. Aktuelle Forschungsarbeiten beschäftigen sich damit, graphische und in der Industrie vertraute Darstellungen in FOCUS zu integrieren und Richtlinien für den Einsatz von FOCUS zu entwickeln.

In Hinblick auf die Modellierungskonzepte weisen SDL und FOCUS große Ähnlichkeiten auf. In beiden wird ein verteiltes System als ein Netzwerk von Komponenten spezifiziert, die asynchron über unbeschränkte Kanäle kommunizieren. Während in SDL das Verhalten

eines Prozesses graphisch durch die Angabe eines erweiterten Zustandsautomaten erfolgt, werden in FOCUS überwiegend prädikatenlogische Formeln verwendet, um das Verhalten von Komponenten zu spezifizieren.

Das Ziel der vorliegenden Arbeit ist es, die Stärken von SDL und FOCUS zu kombinieren und somit eine Beschreibungssprache mit intuitiver graphischer Darstellung und formaler Semantik zu erhalten. Darauf aufbauend wird eine Verifikationsmethode für SDL-Spezifikationen entwickelt. Mit diesem Vorgehen werden die Defizite von SDL behoben, und FOCUS wird um eine in der Industrie akzeptierte Beschreibungstechnik erweitert. Die Verbindung von SDL und FOCUS wird erreicht, indem die Sprachkonstrukte von SDL eine formale Semantik auf Basis der mathematisch-logischen Konzepte von FOCUS erhalten und ihnen somit eine eindeutige Bedeutung zugewiesen wird. Damit können die Beweistechniken von FOCUS für die Verifikation von SDL-Spezifikationen verwendet werden – der formale Nachweis von Eigenschaften einer SDL-Spezifikation ist möglich. SDL trägt die Bezeichnung „formale“ Spezifikationssprache dann zu Recht.

1.2 Ziele und Methoden der Arbeit

Die Zielsetzung dieser Arbeit liegt in der Entwicklung einer formalen, semantischen Fundierung von SDL im Rahmen der formalen Entwicklungsmethodik FOCUS und darauf aufbauend einer Methode für die Verifikation von SDL-Spezifikationen. Damit kann SDL zukünftig als formale Spezifikationssprache im Systementwicklungsprozeß eingesetzt werden. Im einzelnen wird die Arbeit

- den Nachweis erbringen, daß SDL bisher keine formale, sondern eine semiformale Spezifikationssprache darstellt,
- den Zeitbegriff in SDL diskutieren und eine Lösung für ein eindeutiges und praktikables SDL-Zeitkonzept angeben,
- die formale Semantik von SDL in FOCUS definieren, wobei für statische sowie für dynamische SDL-Systeme die jeweils adäquate mathematisch-logische Variante von FOCUS gewählt wird,
- eine Methode für die Verifikation von SDL-Spezifikationen entwickeln und als Fallstudie das Alternating Bit Protokoll heranziehen.

Wie diese Zielsetzungen zeigen, beschäftigen wir uns in dieser Arbeit ausschließlich mit der Spezifikationssprache SDL, nicht mit methodischen Aspekten für den Einsatz von SDL bei der Systementwicklung.

Hinsichtlich der Semantikdefinition für SDL wird der Anwenderfreundlichkeit besondere Bedeutung zukommen. Ziel ist es, die Semantikdefinition für SDL intuitiv und somit einsehbar adäquat und nachvollziehbar zu gestalten.

Abbildung 1.1 veranschaulicht unser Vorgehen und stellt den Zusammenhang der in dieser Arbeit verwendeten Methoden dar. Eine ausführliche Vorstellung der einzelnen Methoden erfolgt in den späteren Kapiteln.

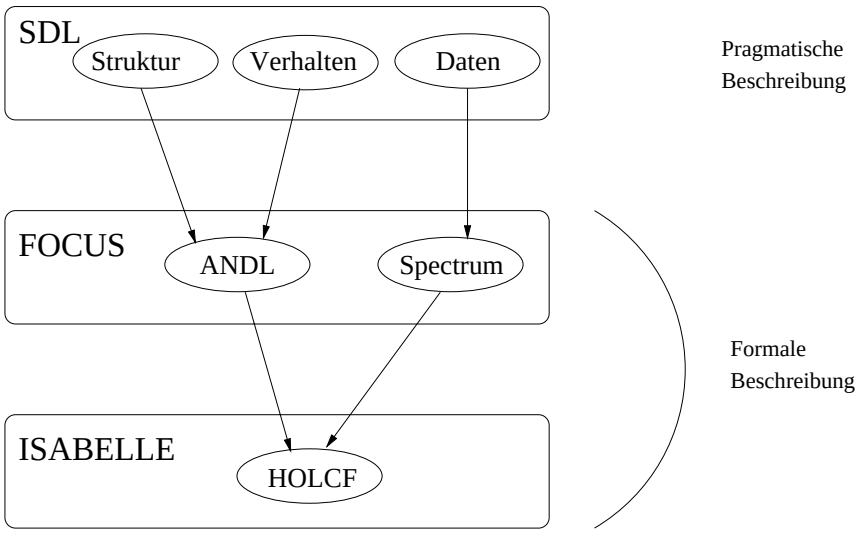


Abbildung 1.1: Die formale, semantische Fundierung von SDL

Den Ausgangspunkt unseres Vorgehens bildet eine SDL-Spezifikation, die eine pragmatische Beschreibung eines Systems wiedergibt. Die SDL-Spezifikation beschreibt die Struktur, den Datenanteil und das Verhalten des Systems. In der vorliegenden Arbeit beschränken wir uns auf eine Teilmenge des SDL-Sprachumfangs, die im wesentlichen mit Basic SDL übereinstimmt und die meist verwendeten Sprachkonstrukte von SDL enthält (siehe [IT93b]).

Um die formale Semantik für die SDL-Spezifikation zu erhalten, wird diese in eine formale Beschreibung nach FOCUS übersetzt, wobei das in FOCUS beschriebene System das gleiche Verhalten wie das mit SDL beschriebene System aufweist. Für das FOCUS-System läßt sich die formale Semantik basierend auf mathematisch-logischen Konzepten angegeben. Dadurch erhalten wir eine wohldefinierte formale Semantik für die SDL-Beschreibung.

Um die Semantikdefinition verständlich zu gestalten, verwenden wir für die FOCUS-Systembeschreibung ANDL ([SS95]). ANDL ist die Kernsprache von FOCUS und unterstützt die schematische Erstellung von Spezifikationen mit programmiersprachenähnlichen Konzepten, ohne daß dabei direkt auf semantische Konzepte zuzugreifen ist. Wie in Abbildung 1.1 dargestellt, werden die Struktur und das Verhalten aus der SDL-Spezifikation nach ANDL umgesetzt. Für die Formalisierung des Datenanteils sind in FOCUS, das konzeptionell auf die Aspekte Kommunikation und Interaktion verteilter, reaktiver Systeme ausgerichtet ist, bisher keine eigenständigen Beschreibungstechniken vorhanden. Deshalb formalisieren wir den Datenanteil einer SDL-Spezifikation mit der algebraischen Spezifikationsmethode SPECTRUM ([BFG+93a]), die wir in FOCUS integrieren.

Auf der FOCUS-Ebene können wir nun Eigenschaften der SDL-Spezifikation formal verifizieren, d.h. wir können die Frage, ob die SDL-Spezifikation eine bestimmte Eigenschaft erfüllt, durch die Anwendung mathematisch-logischer Beweistechniken beantworten. Dabei

ist es möglich, die Verifikation maschinenunterstützt mit dem interaktiven Theorembeweiser Isabelle ([Pau94]) durchzuführen. Die geeignete logische Basis in Isabelle stellt HOLCF dar – eine Logik höherer Stufe, die um die Konzepte der Bereichstheorie erweitert ist ([Reg94]). In dieser Logik formalisieren wir die SDL-Semantikdefinition. In Abbildung 1.1 ist der Übergang von der FOCUS-Spezifikation des SDL-Systems nach HOLCF dargestellt. Sowohl die ANDL- als auch die SPECTRUM-Anteile der Spezifikation werden in die Logik HOLCF umgesetzt.

Die hier vorgestellte Vorgehensweise für die formale Fundierung von SDL umfaßt SDL-Spezifikationen mit statischem Verhalten. Für SDL-Systeme mit dynamischer Prozeßgenerierung während des Systemablaufs werden wir eine Variante des semantischen Modells von FOCUS verwenden, die einen geeigneten Rahmen für die Spezifikation mobiler, dynamischer Systeme bietet.

Für die in Abbildung 1.1 dargestellte Vorgehensweise läßt sich eine durchgehende Werkzeugunterstützung entwickeln. Für den ersten Umsetzungsschritt von SDL nach FOCUS wurde bereits im Rahmen eines Projekts ein Prototyp entwickelt, der für SDL-Spezifikationen die zugehörigen FOCUS-Beschreibungen in ANDL generiert (siehe [Hin96b]). Zukünftige Arbeiten des Projekts SFB 342/A6[2] werden sich mit der Entwicklung einer Beweismethodik unter Einbeziehung des Theorembeweisers Isabelle beschäftigen ([BBSS97]). In diesem Zusammenhang wird ein Parser erstellt, der für eine ANDL-Beschreibung die dazugehörige Logikbeschreibung in HOLCF generiert.

1.3 Vergleichbare Arbeiten

In den letzten Jahren hat SDL als Vertreterin einer pragmatischen, industriell eingesetzten Spezifikationssprache für verteilte Systeme vermehrt das Interesse in akademischen Kreisen geweckt. Im folgenden geben wir einen Überblick über einige Arbeiten, die sich mit der Erstellung einer formalen Semantik für SDL befassen. Der Überblick erhebt keinen Anspruch auf Vollständigkeit.

Semantikdefinition mit Object Z – BaseSDL [FLP95, ITU94]:
In [FLP95, ITU94] wird die Entwicklung eines neuen Semantikkonzepts für SDL beschrieben. Ziel ist es, eine Semantik für SDL zu definieren, welche die in SDL 92 enthaltenen objektorientierten Konstrukte berücksichtigt, und dabei SDL neu zu überdenken und an einigen Stellen zu erweitern oder zu reduzieren. Die Semantik wird demnach für ein neues SDL (SDL 2000) entwickelt; die Arbeiten werden im Rahmen der Arbeitsgruppe SG10 der ITU-T durchgeführt. Die derzeitige SDL-Semantikdefinition der ITU-T ([IT93a]) wird zwar hinsichtlich ihrer Komplexität und der fehlenden Zeitbehandlung kritisiert; die Formalität der Semantik wird jedoch nicht in Frage gestellt.
Als Grundlage für die neue Semantik dient die abstrakte, objektorientierte Sprache BSDL (BaseSDL), die in ihrer intuitiven Semantik SDL ähnelt. Die komplizierten SDL-Konstrukte

[2]Sonderforschungsbereich SFB 342 „Werkzeuge und Methoden zur Nutzung paralleler Rechnerarchitekturen“

sollen auf einfache BSDL-Konstrukte abgebildet werden. Die Semantik von BSDL selbst wird unter Verwendung von Object-Z ([DKRS91]) definiert, das eine objektorientierte Erweiterung von Z ([Spi92]) darstellt. Fallstudien, die die Eignung von BSDL für die Beweisführung untersuchen, sind noch nicht abgeschlossen. Die Semantik und Teile der Syntax von BSDL sind noch im Entwicklungsstadium.

Semantikdefinition mit Prozeßalgebra [God91a, God91b]:
In [God91a, God91b] wird eine operationelle Semantik für Basic SDL definiert, indem der Ablauf eines SDL-Systems mittels einer Prozeßalgebra nach [Mil80] beschrieben wird. Ziel dieser Arbeit ist es, eine semantische Basis für SDL zu schaffen, um darauf aufbauend eine Interpretationsstruktur zu definieren, in der Eigenschaften einer SDL-Spezifikation mit temporaler Logik beschrieben werden können. In der Arbeit wird allerdings nur die Definition der operationellen Semantik behandelt.
Die offizielle Semantik der ITU-T ([IT93a]) wird als zu wenig abstrakt kritisiert. Auf die Zeitbehandlung wird nur am Rande eingegangen, insbesondere wird festgestellt, daß SDL nicht als Echtzeitsprache bezeichnet werden kann. Die Autoren weisen darauf hin, daß ihre operationelle Semantik von SDL in einigen Punkten von der offiziellen Semantik in den Empfehlungen der ITU-T abweicht.

Semantikdefinition mit Prozeß- und Netzwerkalgebra – φSDL [BM95, BMU97, BMŞ97a]:
Ein weiterer Ansatz für eine formale SDL-Semantik wird in den Arbeiten [BM95, BMU97, BMŞ97a] vorgestellt. Für eine Teilmenge des Sprachumfangs von SDL – genannt φSDL – wird eine Umsetzung in eine Prozeßalgebra mit diskreter Zeit (ACP) definiert. Dabei wird die Struktur von SDL-Systemen als Datenflußnetz behandelt und mit Basic Network Algebra ([BMŞ97b]) modelliert. In φSDL werden einige Sprachkonstrukte vereinfacht. So entfällt zum Beispiel die Verwendung von Blöcken für die Systemstrukturierung; auf die lokalen Variablen der SDL-Prozesse kann systemweit zugegriffen werden. In der Arbeit wird insbesondere die Behandlung des Zeitbegriffs in der offiziellen SDL-Semantik als zu komplex und inadäquat kritisiert. Für φSDL wird ein diskretes Zeitmodell festgelegt, in dem ein SDL-Prozeß das nächste Zeitintervall erreicht, wenn keine Signale mehr zur Verarbeitung vorliegen. Eine nachvollziehbare Motivation für diese Zeitwahl wird nicht gegeben. Mit der Semantikdefinition wird die Ausführung eines mit SDL spezifizierten Systems beschrieben. Es ist nicht möglich, Eigenschaften des Systems auf einer hohen Abstraktionsebene zu formulieren. Auf die Anwendung der formalen Semantik (Verifikationsmethodik und Werkzeugunterstützung) für die Analyse von SDL-Systemen wird nicht näher eingegangen. Ebensowenig erfolgt ein Vergleich mit existierenden Ansätzen, wie etwa mit [HS94], zu dessen Modellierungsidee von kompositionaler SDL-Semantik als Datenflußnetz [BMŞ97a] offensichtlich große Ähnlichkeiten aufweist.

Semantikdefinition mit Timed Rewriting Logic [SK98]:
In [SK98] wird für einen stark eingeschränkten Sprachumfang von SDL 88 eine operationelle Semantik mit einem algebraischen Formalismus definiert, genannt Timed Rewriting Logic (TRL). TRL erweitert die klassischen algebraischen Spezifikationstechniken um Termersetzungsregeln und ermöglicht damit die Spezifikation von dynamischem Systemverhalten;

Termersetzungsregeln modellieren die Zustandsübergänge des Systems. Die Regeln können mit quantitativen Angaben zum Zeitverbrauch eines Zustandsübergangs versehen werden. Die Frage, wieviel Zeit eine SDL-Transition verbraucht, die durch eine solche Termersetzungsregel beschrieben ist, wird nicht behandelt.

Semantikdefinition mittels stromverarbeitender Funktionen [Bro91, HS94]:
In [Bro91] wird ein erster Ansatz für eine funktionale Semantikdefinition für SDL vorgestellt. SDL-Spezifikationen werden funktionalen Systembeschreibungen gleichgesetzt und erhalten eine denotationelle Semantik mittels stromverarbeitender Funktionen. Das Zeitkonzept von SDL wird dabei kritisiert, jedoch nicht näher untersucht. [Bro91] liefert die Grundlage für die Durchführung dieser Arbeit. In [Bro92] werden der Ansatz nochmals aufgegriffen und anhand eines einfachen Pufferbeispiels die Anwendung von Beweiskonzepten diskutiert.

In [HS94] wird aufbauend auf [Bro91] eine eingeschränkte Version von SDL als Zielsprache für FOCUS entwickelt. SDL wird eine denotationelle, auf stromverarbeitenden Funktionen basierende Semantik zugeordnet. Zeitliche Aspekte von SDL werden dabei nicht behandelt. Ziel der Arbeit ist es, ein System schrittweise mit FOCUS zu entwickeln und erst in späten Entwicklungsphasen in SDL-Beschreibungen zu transformieren.

Semantikdefinition mit Petrinetzen [Gra90, FT97]:
In der Diplomarbeit [Gra90] werden Analysemöglichkeiten von SDL-Spezifikationen auf der Basis von Petrinetz-Darstellungen untersucht. Zunächst werden SDL-Spezifikationen genormt, um ihre Umsetzung in Netzsysteme zu vereinfachen. So werden zum Beispiel Signalwege ohne Übertragungsverzögerung durch Kanäle mit Verzögerung ersetzt. Um den Datenanteil in SDL-Spezifikationen adäquat behandeln zu können, werden SDL-Prozesse mit lokalen Variablen in Prädikat-/Transitionssysteme umgesetzt. Lokale Daten werden dabei durch attributierte Marken modelliert. Um die Analyse großer Netze zu bewältigen, werden Morphismen (Vergröberungen und Verfeinerungen) verwendet. Es ist nicht bekannt, ob das in [Gra90] beschriebene Verfahren praktisch angewandt wird.

In [FT97] erfolgt für eine Echtzeiterweiterung von SDL, genannt SDL/R, eine Semantikdefinition mit High Level Petrinetzen. SDL-Spezifikationen werden mit Zusicherungen in Form von Zeitintervallen angereichert: in den SDL-Prozessen werden den Aktionen eines Zustandsübergangs minimale bzw. maximale Ausführungszeiten zugeordnet, die Kanalverbindungen sowie die Eingabepuffer der SDL-Prozesse werden mit minimalen bzw. maximalen Verzögerungszeiten versehen. Die so erweiterten SDL-Spezifikationen werden in TM-Netze übersetzt, wodurch eine kompositionale Semantik für SDL/R gewonnen wird. TM-Netze sind High Level Petrinetze, für deren Transitionen Zeitbeschränkungen definiert sind. SDL/R-Spezifikationen können somit auf der semantischen Ebene mit den Methoden gezeiteter Petrinetze werkzeugunterstützt analysiert werden.

Arbeiten zum Zeitkonzept:
Einige Arbeiten haben sich mit dem Zeitkonzept von SDL befaßt, zum Beispiel [BB91], [HL92], [Leu95] und [MGHS96]. In diesen Arbeiten geht es jedoch um die mangelnde Aus-

druckskraft dieses Konzepts hinsichtlich der Spezifikation von Echtzeitanforderungen. So wird zum Beispiel in [Leu95] der Timermechanismus von SDL analysiert und festgestellt, daß sich dieser nicht für die Modellierung von Echtzeitbedingungen eignet. Es wird vorgeschlagen, SDL mit temporaler Logik zu kombinieren, um Systeme mit Echtzeitaspekten adäquat modellieren zu können. In [MGHS96] wird eine formale Semantik für SDL mittels des Duration Calculus ([Cha93]) definiert. Durch diese Semantikdefinition können untere und obere Schranken für die Ausführungszeiten von SDL-Systemen angegeben werden. Damit lassen sich Aspekte wie Schedulingstrategien und Partitionierung von SDL-Prozessen auf Betriebssystemprozesse und -prozessoren behandeln. Es werden Aussagen auf sehr implementierungsnaher Ebene getroffen, nicht auf der abstrakten Ebene der Spezifikation.

Wie der Überblick gezeigt hat, existiert eine Reihe unterschiedlicher Ansätze für eine formale Semantikdefinition für SDL. Eine Diskussion der Frage, ob es sich bei SDL um eine formale Spezifikationssprache handelt, ist in keiner der Arbeiten enthalten. In den uns bekannten Arbeiten wird die Semantikdefinition der ITU-T in der Z.100 nur oberflächlich untersucht, ihre Formalität nicht in Frage gestellt. Das Zeitkonzept wird nur hinsichtlich der mangelnden Ausdrucksfähigkeit für Echtzeitanforderungen kritisiert – der grundlegende Zusammenhang zwischen dem Ablauf eines SDL-Systems und dem Fortschreiten der Zeit wird nicht untersucht. Bis auf [Bro92] wird in keiner dieser Arbeiten auf Basis der Semantikdefinition eine formale Verifikationsaufgabe für SDL-Spezifikationen durchgeführt, so daß sich über die Eignung und Praktikabilität der Semantikvorschläge in Bezug auf Verifikation keine Aussage treffen läßt.

1.4 Gliederung der Arbeit

Im folgenden erläutern wir den Aufbau unserer Arbeit.

Kapitel 2 setzt sich kritisch mit der Sprache SDL auseinander und diskutiert, inwieweit SDL den Anforderungen einer formalen Spezifikationssprache gerecht wird. Zunächst wird der in dieser Arbeit betrachtete Sprachumfang von SDL beschrieben. Danach untersuchen wir, ob die von der ITU-T vorgegebene und somit offizielle SDL-Semantik eine formale Semantik von SDL darstellt. Anschließend folgt das eigentliche Kernstück des Kapitels: eine Untersuchung des Zeitkonzepts von SDL. Wir stellen die Ergebnisse einer Umfrage zum SDL-Zeitkonzept unter SDL-Anwendern vor und präsentieren ein alternatives Zeitkonzept, das die Schwächen des bestehenden behebt. Nach einer Analyse des Datenkonzepts in SDL sowie einer Aufzählung weiterer Schwachpunkte von SDL beantworten wir die Frage, ob es sich bei SDL um eine formale oder lediglich um eine semiformale Spezifikationssprache handelt.

In Kapitel 3 stellen wir die formale Entwicklungsmethodik FOCUS vor. Dabei beschränken wir uns auf die Aspekte von FOCUS, die für die anschließende Semantikdefinition von SDL von Bedeutung sind. Dazu zählt vor allem ANDL, die logische Kernsprache von FOCUS. Wir definieren eine Erweiterung von ANDL, mit der sich auch zeitliche Aspekte eines Sy-

stems spezifizieren lassen. Den Abschluß bildet eine kurze Einführung in die algebraische Spezifikationssprache SPECTRUM.

In Kapitel 4 definieren wir eine formale denotationelle Semantik für SDL im Rahmen der Entwicklungsmethodik FOCUS. Die Semantikdefinition erfolgt zunächst für statische SDL-Systeme; die dynamische Prozeßgenerierung wird in Kapitel 5 behandelt. Für die Darstellung der Semantik verwenden wir die logische Kernsprache ANDL. Unser Vorgehen ist dadurch charakterisiert, daß einer SDL-Spezifikation eine formale Spezifikation in FOCUS zugeordnet wird, wobei das mit SDL und das in FOCUS beschriebene System das gleiche Verhalten aufweisen. Die Semantikdefinition für SDL gliedert sich gemäß der Aspekte Struktur, Daten und Verhalten in drei Abschnitte. Zuerst erfolgt die formale Fundierung der Systemstruktur von SDL-Spezifikationen. Anschließend definieren wir die formale Semantik für die Datentypen von SDL. Den dritten und umfangreichsten Teil der Semantikdefinition für SDL bildet die formale Fundierung der SDL-Prozesse. Hier erfolgt die Definition einer Reihe von Semantikbausteinen sowie die Ableitung von Funktionsgleichungen aus einem SDL-Prozeß. Eine Bewertung unserer Semantikdefinition für SDL beschließt das Kapitel.

Kapitel 5 befaßt sich mit der formalen Fundierung der Prozeßgenerierung in SDL, mit der dynamische Systemerweiterungen um SDL-Prozesse modelliert werden. Zunächst stellen wir eine Erweiterung des klassischen FOCUS vor, die die Modellierung mobiler, dynamischer Systeme ermöglicht und damit die semantische Basis für die dynamische Prozeßgenerierung in SDL bildet. Anschließend definieren wir die formale Semantik für die SDL-Prozeßerzeugung, wobei wir zwischen der Erzeugung einer einzelnen Prozeßinstanz und einer Menge von Prozeßinstanzen unterscheiden. Um unsere Semantikdefinition an einem Beispiel vorzustellen, geben wir für die SDL-Spezifikation des Daemon Game – einem SDL-Standardbeispiel für die dynamische Prozeßerzeugung – in Anhang A die formale Semantikdefinition an. Dabei gehen wir ausführlich auf die dynamische Veränderung der Struktur des SDL-Systems ein, die durch die Prozeßerzeugung hervorgerufen wird.

In Kapitel 6 entwickeln wir eine Verifikationsmethode für SDL Spezifikationen. Zu Beginn führen wir eine Verifikationsaufgabe für die SDL-Spezifikation des Alternating Bit Protokolls durch und zeigen, wie sich die Beweistechniken von FOCUS für unsere Verifikationsaufgabe verwenden lassen. Aufbauend auf den Erfahrungen, die wir in der Fallstudie gewonnen haben, geben wir eine Verifikationsmethode für SDL-Spezifikationen an. Diese Methode beschreibt einen durchgängigen Ansatz, der von der informellen Systembeschreibung über die SDL-Spezifikation bis zur Beweisführung reicht. Um die Voraussetzungen für maschinengestütztes Beweisen zu schaffen, formalisieren wir die SDL-Semantik in der Logik HOLCF des interaktiven Theorembeweisers Isabelle. Daran anschließend diskutieren wir die formale Systementwicklung von SDL-Spezifikationen in FOCUS und stellen unserer Verifikationsmethode zwei Verfahren gegenüber, die in der Industrie für die Überprüfung von SDL-Spezifikationen eingesetzt werden: Model Checking und Simulation.
Kapitel 6 wird durch Anhang B ergänzt. In diesem finden sich Prädikate sowie Lemmata und Beweise zur Fallstudie des Alternating Bit Protokolls.

Kapitel 7 faßt die Ergebnisse der Arbeit zusammen und schließt mit einem Ausblick auf mögliche zukünftige Arbeiten im Bereich formale, semantische Fundierung und Verifikation von SDL-Spezifikationen.

Kapitel 2

SDL – eine formale Sprache?

SDL ist eine Spezifikationssprache, die von Anwendern in der Industrie und führenden Werkzeugherstellern sowie in der Literatur gern als formale Beschreibungssprache bezeichnet wird. SDL-Anwender schätzen an SDL besonders, daß SDL eine formale und standardisierte Spezifikationssprache ist. Dabei verstehen sie unter „formal“, daß der Sprachumfang von SDL genau festgelegt und die Bedeutung der einzelnen Sprachkonstrukte mittels der Empfehlung Z.100 der ITU-T definiert sind. Die Eigenschaft „formal“ wird jedoch im Bereich der theoretischen Informatik strenger definiert als im industriellen Umfeld. Wir bezeichnen eine Spezifikationssprache als formal, wenn sie über folgende Eigenschaften verfügt:

- eine klar festgelegte Syntax,
- eine wohldefinierte, auf mathematischen und logischen Konzepten basierende, formale Semantik,
- eine Menge von Beweis- und Transformationsregeln.

Eine Spezifikationssprache, die nur das erste Kriterium erfüllt, wird als semiformal bezeichnet.

Im vorliegenden Kapitel klären wir, inwieweit SDL den eben genannten Kriterien einer formalen Beschreibungssprache tatsächlich gerecht wird. Zunächst stellen wir in Abschnitt 2.1 den Sprachumfang von SDL vor, den wir in dieser Arbeit behandeln. Danach analysieren wir in Abschnitt 2.2, ob die von der ITU-T vorgegebene und somit „offizielle“ SDL-Semantik, die als Anhang in den Empfehlungen der Z.100 enthalten ist, eine formale Semantik von SDL darstellt. Anschließend folgt mit Abschnitt 2.3 der Schwerpunkt des Kapitels: eine eingehende Untersuchung des Zeitkonzepts von SDL. Wir stellen die unterschiedlichen Interpretationen des Zeitbegriffs in SDL vor und präsentieren ein alternatives Zeitkonzept, das die Defizite von SDL hinsichtlich der Zeitbehandlung behebt. Im Anschluß daran betrachten wir den Datenanteil von SDL (Abschnitt 2.4) und gehen in Abschnitt 2.5 auf weitere Schwachpunkte von SDL ein. In Abschnitt 2.6 beantworten wir die Frage, ob SDL eine formale Spezifikationssprache darstellt.

2.1 Basic SDL

SDL wurde für die Spezifikation von Telekommunikationssystemen entwickelt. Mitte der 70er Jahre wurde erstmals von der ITU-T (damals CCITT) eine Sprachversion empfohlen, die seitdem mehrmals ergänzt und überarbeitet wurde. Unsere Arbeit basiert auf SDL 92 ([IT93b]), wobei wir die objektorientierten Sprachanteile nicht berücksichtigen. In der vorliegenden Arbeit beschränken wir uns auf eine Teilmenge des SDL-Sprachumfangs, welche die meist verwendeten Sprachkonstrukte von SDL enthält. Diese Teilmenge entspricht im wesentlichen dem Sprachumfang von Basic SDL, das laut der Empfehlung Z.100 die grundlegenden Sprachkonstrukte von SDL umfaßt, so daß wir im weiteren von Basic SDL sprechen werden. SDL bietet neben der graphischen Syntax eine textuelle Syntax an, genannt SDL/PR. Da bei der Spezifikation mit SDL jedoch überwiegend die graphische Darstellung verwendet wird, werden wir die textuelle Syntax in dieser Arbeit nicht berücksichtigen. Im folgenden geben wir einen Überblick über Basic SDL, wobei wir uns auf die wesentlichen Konzepte beschränken und auf syntaktische Details verzichten. Eine ausführlichere Einführung in Basic SDL enthalten beispielsweise [EHS97, BH93, OFMP+94].

Ein System wird in SDL auf oberster Strukturierungsebene durch ein Systemdiagramm beschrieben, wobei sich das System aus mehreren Teilen, sogenannten Blöcken, zusammensetzt. Diese sind untereinander und mit der Systemumgebung über uni- oder bidirektionale Kanäle verbunden. Über die Kanäle können Signale empfangen oder verschickt werden, wobei die Signale Datenwerte mit sich führen können. Die Kanäle agieren nach dem FIFO-Prinzip[1]; die Reihenfolge der auf dem Kanal verschickten Signale bleibt erhalten, es werden keine Signale hinzugefügt, gelöscht oder verdoppelt. Hinsichtlich der Übertragungsdauer kann zwischen verzögerungsfreier Übertragung oder einer beliebig langen, aber endlichen Verzögerung gewählt werden. Die Blöcke können hierarchisch weiter in Blöcke zerlegt werden, so daß sich eine baumartige Strukturierung des Systems ergibt. Abbildung 2.1 zeigt eine schematische Darstellung einer Systemstrukturierung mit SDL.

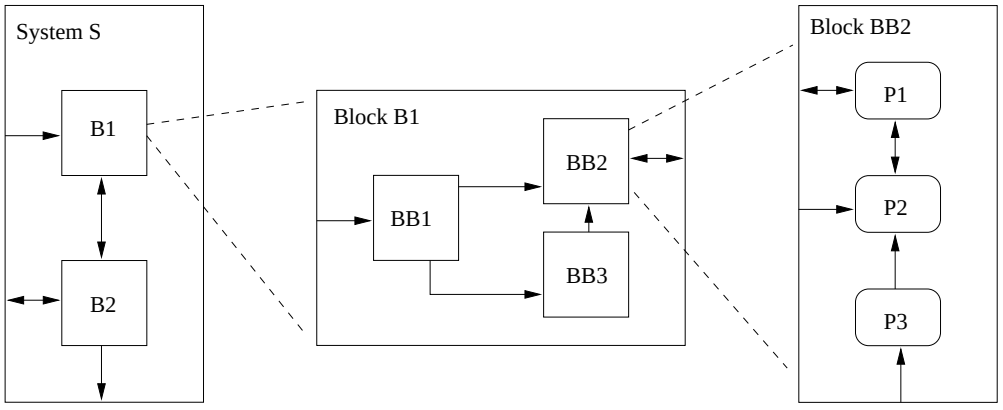


Abbildung 2.1: Strukturierung einer SDL-Spezifikation

[1]FIFO (First In First Out): Die Signale werden genau in der Reihenfolge aus dem Kanal ausgegeben, in der sie in den Kanal eingegeben worden sind.

Ein Block besteht auf unterster Ebene aus einer Menge von Prozessen, die durch erweiterte Zustandsautomaten beschrieben werden. Die Prozesse beschreiben das Verhalten des Systems. Die Kommunikationsverbindungen zwischen den Prozessen innerhalb eines Blocks werden Signalwege genannt. An der Blockgrenze erfolgt der Übergang von den Kanälen in die uni- oder bidirektionalen Signalwege. Die Übertragung über Signalwege erfolgt verzögerungsfrei ohne Zeitverbrauch nach dem FIFO-Prinzip. Die Prozesse agieren nebenläufig und gleichberechtigt und kommunizieren untereinander und mit der Systemumgebung asynchron durch den Austausch von Signalen.

Das Verhalten eines Prozesses wird durch einen Zustandsautomaten mit Ein- und Ausgabe beschrieben, der zusätzlich über einen möglicherweise unendlichen Datenzustand verfügt. Ein Prozeß besteht demzufolge aus einer Menge von (Kontroll-)Zuständen und Zustandsübergängen, auch Transitionen genannt. Ein Zustandsübergang wird durch das Verarbeiten eines Signals ausgelöst.

In Abbildung 2.2 werden die einzelnen Bestandteile einer SDL-Prozeßspezifikation genauer dargestellt. In der graphischen Darstellung werden der Prozeßname, der Prozeßdeklarationsbereich und der Bereich des Prozeßgraphen, mit dem das Verhalten des Prozesses beschrieben wird, angegeben.

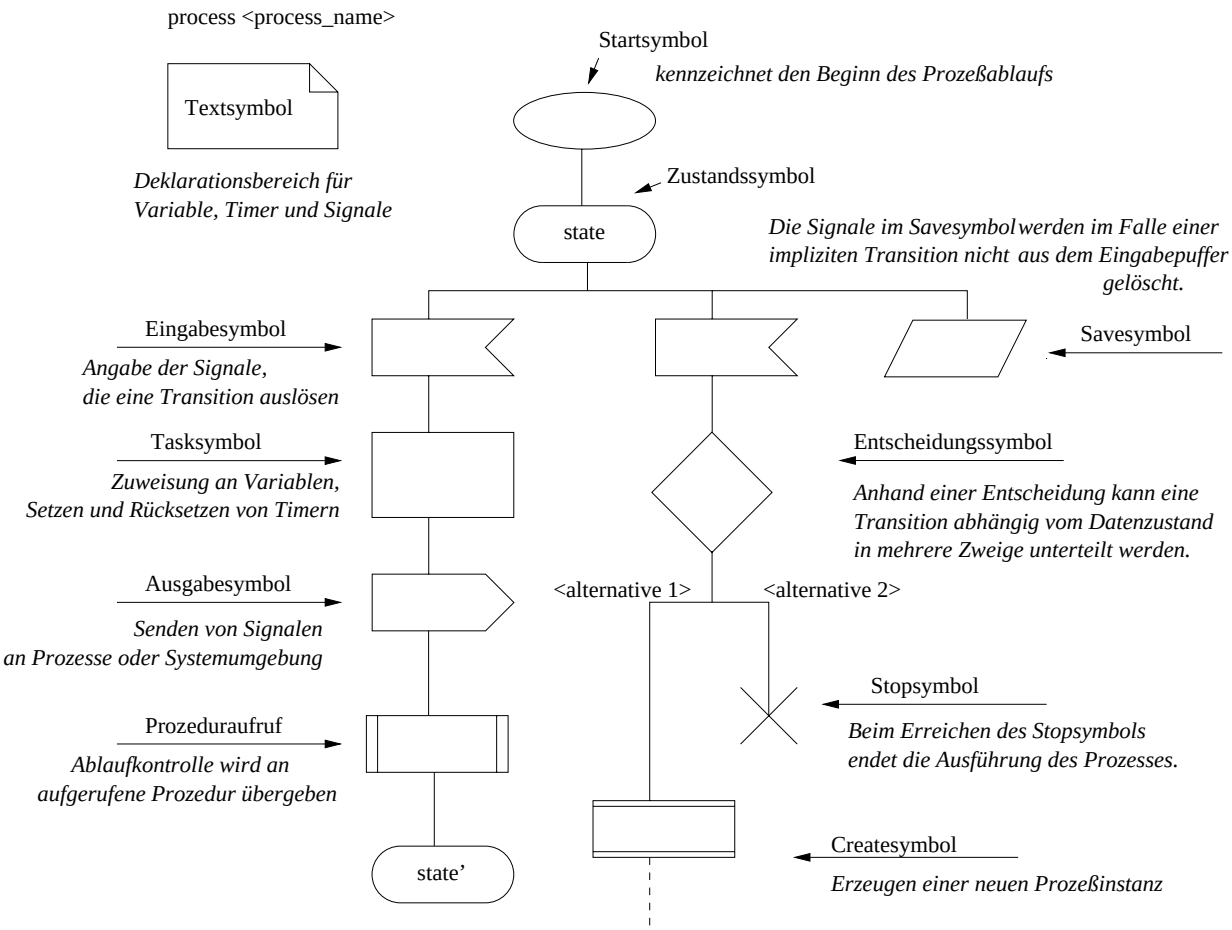


Abbildung 2.2: Schematische Darstellung eines SDL-Prozesses

Jeder Prozeß verfügt über einen unbegrenzten Eingabepuffer, den er nach dem FIFO-Prinzip abarbeitet. Signale, die den Prozeß über Signalwege erreichen, werden gemäß der Reihenfolge ihrer Ankunft in den Puffer eingetragen. Dabei wird das Prinzip des Interleaving angewendet, d.h. gleichzeitig eintreffende Signale werden in beliebiger Reihenfolge in den Eingabepuffer eingetragen („*If two or more signals arrive on different paths „simultaneously" they are arbitrarily ordered.*", [IT93b]). Befindet sich der Prozeß in einem Zustand, so liest er das nächste Signal aus seinem Puffer und führt den zugehörigen Zustandsübergang aus. Ist für das Signal kein Übergang angegeben, so wird es aus dem Eingabepuffer gelöscht, wobei der Prozeßzustand (sowohl Kontroll- als auch Datenzustand) unverändert bleibt. Dieser Vorgang wird als implizite Transition bezeichnet. Während eines Zustandsübergangs können Aktionen wie das Senden von Signalen, die Manipulation des Datenzustands oder das Setzen und Rücksetzen von Timern ausgeführt werden. Zudem kann ein Prozeß während eines Zustandsübergangs neue Prozeßinstanzen anderer Prozesse innerhalb des eigenen Blocks erzeugen.
Der Datenzustand eines Prozesses setzt sich aus der Belegung der prozeßlokalen Variablen zusammen. Das Datentypkonzept in SDL basiert auf dem Konzept der abstrakten Datentypen (ADT). Einige Datentypen (Sorten) sind in SDL vorgegeben, wie z.B. Boolesche Werte, Integers und Arrays. In Abschnitt 2.4 wird auf das Datenkonzept näher eingegangen.

Spezifikationsteile von Prozessen, die wiederholt vorkommen, können als Prozedur definiert werden. Eine Prozedur ist ein parametrisierter Teil einer SDL-Prozeßbeschreibung, der über einen eigenen Gültigkeitsbereich für lokale Variablen und Zustände verfügt. Beim Aufruf der Prozedur stoppt die Ausführung des SDL-Prozesses, bis die Prozedur beendet ist. Daneben ist die Definition von Prozeduren möglich, die Ergebniswerte zurückliefern und innerhalb von Anweisungen in Tasksymbolen aufgerufen werden.

Zu Beginn eines Systemablaufs werden die Prozeßinstanzen erzeugt und initialisiert. Von einer SDL-Prozeßbeschreibung können mehrere Instanzen unabhängig voneinander existieren. Zudem können während des Systemablaufs dynamisch weitere Instanzen erzeugt werden. Jede Prozeßinstanz erhält einen eindeutigen Bezeichner.

Das Verhalten eines SDL-Systems ist in hohem Maße nichtdeterministisch:

- Das Anwenden von Interleaving bei der Einordnung der Signale in die Eingabepuffer der Prozesse führt zu nichtdeterministischem Verhalten der Prozesse und damit des gesamten Systems.
- Befinden sich zwischen Blöcken Kanäle mit Verzögerung, so resultiert daraus ebenfalls nichtdeterministisches Systemverhalten, da die Signalübertragung beliebige, aber endliche Zeit in Anspruch nehmen kann.
- Innerhalb eines Prozesses kann nichtdeterministisches Verhalten durch spontane Übergänge und nichtdeterministische Entscheidungen beschrieben werden. Spontane Übergänge kann ein Prozeß ohne Vorliegen eines Eingabesignals ausführen; sie sind durch das Schlüsselwort *none* im Eingabesymbol gekennzeichnet. Bei einer nichtdeterministischen Entscheidung ist im Entscheidungssymbol das Schlüsselwort *any* enthalten. Aus der Menge der angegebenen Verzweigungen wird bei jeder Ausführung des Zustandsübergangs eine beliebige ausgewählt.

2.2 Die „offizielle“ SDL-Semantik

Gemäß den Empfehlungen der ITU-T existiert eine formale Semantik zu SDL ([IT93a]), die mittels MetaIV ([BJ78]) und CSP-ähnlichen Kommunikationskonstrukten ([Hoa85]) definiert ist. Obwohl erste Empfehlungen für SDL von der ITU-T bereits in den 70er Jahren herausgegeben worden waren, wurde die Semantikdefinition erst 1988 nachträglich in die ITU-T-Empfehlung Z.100 aufgenommen. Die Semantikdefinition stellt jedoch lediglich einen Anhang der Z.100 dar, wobei darauf hingewiesen wird, daß bezüglich der Definition von SDL die technische SDL-Beschreibung in der Z.100 Vorrang vor der SDL-Semantik hat - Ausnahmen sind Inkonsistenzen in der Z.100. In diesem Abschnitt stellen wir die Semantikdefinition gemäß Annex F der Z.100 kurz vor und erörtern, warum es sich dabei um keine Semantik im formalen Sinn (gemäß der Kriterien auf Seite 11) handelt.

2.2.1 Beschreibung der Semantik gemäß Z.100

Die offizielle SDL-Semantik beschreibt die Implementierung von SDL auf einer abstrakten Maschine und besteht aus den Teilen statische und dynamische Semantik. Die statische Semantik enthält Anforderungen an die formale Syntax von SDL und ist somit für die Frage, welches Verhalten ein mit SDL spezifiziertes System aufweist, nicht ausschlaggebend. In der dynamischen Semantik wird ein formales Modell angegeben, das das Verhalten eines SDL-Systems beschreibt. Dieses Modell besteht aus einer Menge synchron kommunizierender Prozesse. Für die Kommunikation zwischen den Prozessen werden CSP-ähnliche Konstrukte verwendet, die Prozesse selbst werden in MetaIV beschrieben.

Das Modell besteht aus sieben verschiedenen Meta-Prozessen[2] und zielt darauf ab, den Ablauf eines mit SDL spezifizierten Systems zu veranschaulichen:

- *system*,
  zuständig für das Versenden von Signalen zwischen den SDL-Prozessen und für das Erzeugen von eindeutigen Prozeßnummern.
- *path*,
  zuständig für die Modellierung von Kanälen mit Verzögerung.
- *timer*,
  zuständig für die Systemzeit.
- *view*,
  zuständig für die Umsetzung der Sprachkonstrukte *Reveal* und *View*. Über diese Sprachkonstrukte wird es einem SDL-Prozeß ermöglicht, innerhalb eines Blocks auf die Werte der Variablen eines anderen Prozesses zuzugreifen[3].

[2] Die Bezeichnung *Meta-Prozeß* stammt aus [IT93a] und soll den Unterschied zwischen den SDL-Prozessen und den Prozessen des semantischen Modells verdeutlichen. Es handelt sich nicht um die Sprachbezeichnung MetaIV.

[3] Die Verwendung dieser beiden Sprachkonstrukte ist gemäß der Z.100 nicht mehr empfehlenswert, so daß wir sie in dieser Arbeit nicht berücksichtigen.

- *process-set-admin*,
  verwaltet alle Instanzen einer SDL-Prozeßbeschreibung.
- *sdl-process*,
  interpretiert das Verhalten einer SDL-Prozeßinstanz.
- *input-port*,
  zuständig für den Eingabepuffer einer SDL-Prozeßinstanz und für die Überwachung von Timern.

Für die Meta-Prozesse *system*, *timer* und *view* existiert in dem Modell jeweils nur eine Instanz. Für jede SDL-Prozeßbeschreibung wird eine Instanz des Meta-Prozesses *process-set-admin* erzeugt, die die Instanzen der SDL-Prozeßbeschreibung verwaltet. Für jede Prozeßinstanz werden eine Instanz des Meta-Prozesses *sdl-process* und eine Instanz des Meta-Prozesses *input-port* kreiert. Für jeden möglichen Pfad mit Verzögerung, über den ein Signal gesendet werden kann, wird eine Instanz des Meta-Prozesses *path* erzeugt.

Um einen Eindruck vom Verhalten des formalen Modells zu bekommen, beschreiben wir die Modellierung des Signalaustauschs zwischen SDL-Prozessen durch die Meta-Prozesse:

> Der Austausch von Signalen erfolgt zentral über den Meta-Prozeß *system*. Der Prozeß *sdl-process*, der den sendenden SDL-Prozeß interpretiert, sendet seine Ausgabe an den Prozeß *system*. Dieser wählt den Pfad zum Empfänger des Signals aus. Enthält der gewählte Pfad keine Verzögerung, so gibt *system* das Signal an diejenige Instanz des Meta-Prozesses *process-set-admin* weiter, zu der der Empfängerprozeß gehört. Falls der gewählte Pfad eine Verzögerung bei der Übertragung enthält, so wird das Signal an die entsprechende Instanz von *path* übergeben, die es ebenfalls an die Instanz von *process-set-admin* sendet. Der Prozeß *process-set-admin* sendet das Signal an die Instanz *input-port* des Empfängerprozesses.

Auf die Behandlung der SDL-Zeit in der offiziellen Semantik wird in Abschnitt 2.3.2 eingegangen.

2.2.2 Beurteilung der „offiziellen“ Semantik

MetaIV wurde Mitte der 70er Jahre als Metasprache für die Vienna Development Method VDM entwickelt ([BJ78, BJ82]). VDM war ursprünglich für die systematische Softwareentwicklung konzipiert, wurde aber wiederholt eingesetzt, um formale Definitionen von bestehenden Programmiersprachen wie Algol 60 oder Pascal zu erstellen. Die Metasprache von VDM wurde für die Beschreibung sequentieller Systeme ausgerichtet; Nebenläufigkeit wurde dabei nicht berücksichtigt (*„... that no examples will be given of concurrent system architectures: the subject meta-language was not designed to cater for this [sadly neglected] area“*, Seite 33 in [BJ78]). Folglich sind in MetaIV keine Sprachkonstrukte für die Kommunikation zwischen nebenläufig agierenden Komponenten enthalten. MetaIV besitzt eine formale Semantik, die auf einer pragmatischen Ausrichtung von denotationellen Konzepten beruht (siehe Seite VIII in [BJ78]).

Um die Metasprache MetaIV für die Semantikdefinition von SDL verwenden zu können, wurde sie um eine programmiersprachliche Notation für Kommunikation in Anlehnung an CSP erweitert. Damit ist es möglich, Kommunikation zwischen Komponenten zu beschreiben und somit Nebenläufigkeit nachzubilden. Allerdings wurde eine Kommunikationsform nach dem Handshake-Prinzip[4] gewählt, die der asynchronen Kommunikation in SDL-Systemen nicht gerecht wird.

Für die Erweiterung von MetaIV wurde jedoch keine formale Semantik angegeben. Die existierende informelle Beschreibung der Kommunikationskonstrukte (siehe Seite 26 in Annex F.1 [IT93a]) ist ungenau und läßt insbesondere die Frage nach der Fairneß der Kommunikation offen. Da es sich um synchrone Kommunikationsprimitive handelt, kann der Nachrichtenaustausch nur durchgeführt werden, wenn Sender und Empfänger gleichzeitig dazu bereit sind. Es wird nicht darauf eingegangen, wie sich ein Prozeß bei der Wahl zwischen mehreren möglichen Kommunikationsanfragen verhält, ob jede Anfrage in endlicher Zeit behandelt wird oder ob eine Verklemmungssituation entstehen kann, wenn die Anfrage eines Prozesses von seinem Kommunikationspartner nicht beantwortet wird. Somit kann eine faire Übertragung von Nachrichten in endlicher Zeit nicht garantiert werden. Das hat zur Folge, daß zum Beispiel die unbestimmte, aber endliche Verzögerung von Signalen über Kanäle, wie sie in SDL möglich ist, nicht modelliert werden kann.

Zusammenfassend läßt sich feststellen, daß die der SDL-Semantikdefinition zugrundeliegende Erweiterung von MetaIV keine formale Semantik besitzt. Demzufolge handelt es sich bei der SDL-Semantikdefinition gemäß der ITU-T um keine mathematisch präzise formale Semantik, bestenfalls um ein Ausführungsmodell für SDL. Darüber hinaus ist die Wahl des Modells an und für sich inadäquat: die synchrone Kommunikationsform steht im Gegensatz zum asynchronen Nachrichtenaustausch in SDL, Fairneßbedingungen aus SDL lassen sich in dem Modell nicht ausdrücken. Ferner ist die informelle Beschreibung des Modells derart ungenau, daß Fragen, die wesentliche Aspekte von SDL betreffen, wie zum Beispiel das Zeitkonzept (siehe Abschnitt 2.3), anhand des Modells nicht geklärt werden können.

2.3 Das Zeitkonzept in SDL

In diesem Kapitel steht die Beziehung zwischen SDL und der Zeit im Mittelpunkt. Bereits in anderen Arbeiten wurde das Zeitkonzept von SDL kritisiert ([BB91, HL92, Leu95]). Dabei ging es um die mangelnde Ausdruckskraft dieses Konzepts hinsichtlich der Spezifikation von Echtzeitanforderungen. Ein wesentlicher Punkt des Zeitkonzepts wurde jedoch bisher nicht diskutiert: die Beziehung zwischen dem Fortschreiten der Zeit und dem Verhalten eines Systems, das mit SDL spezifiziert wird. Wie wir im folgenden darlegen werden, ist der Zeitbegriff in SDL unzureichend definiert und somit der jeweiligen Interpretation des Anwenders überlassen.

Abschnitt 2.3.1 stellt die Sprachkonstrukte aus SDL vor, mittels derer zeitliches Verhalten spezifiziert werden kann. Daran schließt sich in Abschnitt 2.3.2 ein Überblick an, wie in der

[4] Der Nachrichtenaustausch erfolgt ohne Pufferung; eine Nachricht kann nur dann gesendet werden, wenn der Empfänger für die Annahme bereit ist.

Z.100 und in Büchern zu SDL das Zeitkonzept behandelt wird. In Abschnitt 2.3.3 diskutieren wir die unterschiedlichen Zeitinterpretationen der SDL-Anwender. Ausgangspunkt der Diskussion ist eine kleine Umfrage zum SDL-Zeitbegriff, die wir unter SDL-Anwendern durchgeführt haben. Schließlich erfolgt in Abschnitt 2.3.4 unser Vorschlag, wie sich das Zeitkonzept von SDL eindeutig definieren läßt.

2.3.1 Sprachkonstrukte mit Beziehung zu Zeit

Zunächst geben wir einen Überblick über die SDL-Sprachkonstrukte, mittels derer Bezug auf die Zeit genommen wird.

- In SDL werden die zwei Datentypen *Time* und *Duration* verwendet, um zeitabhängige Werte auszudrücken. Werte des Typs *Time* stehen für Zeitpunkte, Werte des Typs *Duration* für Zeitintervalle. Zeiteinheiten für die Typen, wie z. B. Sekunden, sind in SDL nicht definiert, können aber vom Anwender eingeführt werden.
- Die Signalübertragung zwischen Prozessen innerhalb eines Blocks erfolgt über Signalwege ohne zeitliche Verzögerung. Bei der Signalübertragung zwischen Blöcken, die über Kanäle erfolgt, kann gewählt werden, ob die Übertragung ohne oder mit beliebiger zeitlicher Verzögerung erfolgen soll.
- Ein Prozeß kann über den *NOW*-Operator während eines Zustandsübergangs die aktuelle globale Systemzeit abfragen.
- Mit dem Timerkonzept in SDL ist es möglich, bestimmte Zeitbedingungen in Spezifikationen zu erfassen. Über die Anweisungen *set* und *reset* in den Tasksymbolen kann ein Timer während eines Zustandsübergangs gesteuert werden. Beim Setzen des Timers wird ein Ablaufzeitpunkt angegeben; der Timer ist damit aktiv. Ist der Ablaufzeitpunkt erreicht, so wird das Timersignal in den Eingabepuffer des Prozesses eingereiht. Ein gesetzter Timer kann zurückgesetzt werden und wird dadurch inaktiv.

Abbildung 2.3 zeigt einen Ausschnitt aus einem SDL-Prozeß aus [OFMP+94] mit einer einfachen Anwendung von Timern:
Der Timer t wird gesetzt, um die Wartezeit auf das Signal *late* auf 14 Zeiteinheiten ab dem aktuellen Zeitpunkt *NOW* zu begrenzen. Falls das Signal *late* innerhalb dieser Zeit nicht eintrifft, so läuft der Timer t ab. Falls das Signal *late* eintrifft, bevor t abläuft, so wird der Timer t zurückgesetzt. Ist der Timer zu diesem Zeitpunkt bereits abgelaufen und befindet sich das Timersignal in der Warteschlange des Prozesses, so wird es aus der Warteschlange gelöscht.

Bei der Verwendung von Timern ist es erforderlich, genaue Zeitpunkte und Zeitdauern anzugeben. Das Problem bezüglich des SDL-Zeitbegriffs liegt nun darin, daß in der Definition von SDL nicht verbindlich festgelegt ist, mit welcher Geschwindigkeit die Prozesse ablaufen, d.h. wann und wieviel Zeit während des Ablaufs eines SDL-Systems tatsächlich vergeht.

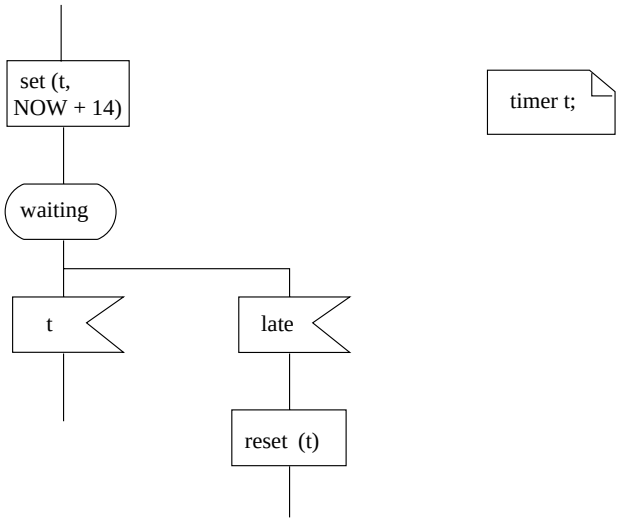


Abbildung 2.3: Beispiel einer Timerspezifikation

2.3.2 Definition der Zeit in der Z.100 und in der Literatur

In diesem Abschnitt untersuchen wir, wie der Zeitbegriff in den Empfehlungen der ITU-T, insbesondere in der offiziellen SDL-Semantik, und in der Literatur zu SDL behandelt wird.

Z.100 [IT93b]:

In der Z.100 wird nichts über den Zeitverbrauch ausgesagt, der bei der Ausführung von Aktionen wie Lesen eines Eingabesignals, Zuweisungen in Tasks und Ausgabe von Signalen in einem SDL-Prozeß auftritt.

In bezug auf den Operator *NOW* findet sich folgende Bemerkung (Seite 160):

The NOW expression represents an expression requesting the current value of the system clock giving the time. The origin and unit of time are system dependent. Whether two occurrences of NOW in the same transition will give the same value is system dependent.

User Guidelines, Annex D in [IT93b]:

In den User Guidelines der Z.100 sind zum Zeitbegriff in SDL folgende Aussagen zu finden:

The abstract machine used in SDL is an extension of the deterministic Finite-State Machine (FSM). ... Transitions from one state to another are usually regarded as taking zero time. (Seite 8, Abschnitt D.2.1.1)

Einige Abschnitte darunter findet sich folgende Aussage:

SDL allows for the possibility of non-zero transition time, and defines a first-in first-out conceptual queueing mechanism for signals which arrive at a machine while it is executing a transition. (Seite 8, Abschnitt D.2.1.1)

Hinsichtlich des Ausdrucks *NOW*, mit dem SDL-Prozesse auf die aktuelle Systemzeit zugreifen, wird ausgesagt, daß sich die Systemzeit während einer Transition erhöhen kann, aber nicht muß.

Somit wird in den Empfehlungen der ITU-T offengelassen, wann während des Ablaufs eines Prozesses Zeit vergeht und ob Zustandsübergänge zeitlos sind.

Dynamische Semantik, Annex F von [IT93b]:

In der offiziellen SDL-Semantik der ITU-T ([IT93a]) ist die Interpretation des Zeitbegriffs ebenfalls nicht eindeutig geklärt. In dem Modell, das der offiziellen Semantik zugrunde liegt (siehe Abschnitt 2.2), existiert ein Meta-Prozeß *timer*, der für die Systemzeit verantwortlich ist. Dieser erhält aus der Systemumgebung von einem Prozeß *tick* in regelmäßigen Abständen Ticks und erhöht daraufhin die Systemzeit. Der Prozeß *timer* ist Teil der formalen Semantik, wohingegen der Prozeß *tick* nur der Veranschaulichung dient und keinen Teil der formalen Semantik darstellt („*It must be noted that the informal model of the tick-process does not form part of the dynamic semantics, it is only included for explanatory reasons.*“, Seite 2, Annex F.3).
Für die Interpretation des Ausdrucks *NOW* sendet *sdl-process* eine Anforderung *Time-Request* an den Meta-Prozeß *timer*. Dieser antwortet mit der Nachricht *Time-Answer*, die die aktuelle Systemzeit enthält.
Der Timermechanismus wird in dem formalen Modell der offiziellen Semantik folgendermaßen umgesetzt: Wird ein Timer gesetzt, so sendet *sdl-process* eine Nachricht mit dem Namen des Timers und dem Ablaufzeitpunkt an seinen Prozeß *input-port*, der den Timer startet. Der Ablauf des Timers wird von *input-port* überwacht, der in regelmäßigen Abständen über den Meta-Prozeß *timer* die aktuelle Systemzeit abfragt. Bei der Interpretation der Reset-Anweisung sendet *sdl-process* eine Nachricht an *input-port*, der den Timer daraufhin stoppt.

Da über die Fairneß der synchronen Kommunikation zwischen den Meta-Prozessen im Modell keine Aussage gemacht wird (vgl. Abschnitt 2.2), können folgende Fragen nicht beantwortet werden:

- In welchen Abständen fragen die Instanzen von *input-port* beim Meta-Prozeß *timer* die aktuelle Systemzeit ab?
- Ist der Meta-Prozeß *timer* jederzeit zu einer Antwort bereit?
- Werden alle Anfragen der Instanzen von *input-port* und *sdl-process* beantwortet?
- In welchen Abständen verarbeitet der Meta-Prozeß *timer* die Ticks aus der Systemumgebung?

Aufgrund dieser offenen Fragen kann es vorkommen, daß die Zeit überhaupt nicht fortschreitet, da keine Kommunikation zwischen den Prozessen *timer* und *tick* erfolgt. Genauso kann es passieren, daß die einzelnen Meta-Prozesse *input-process* nicht über das Fortschreiten der Zeit informiert sind, da sie entweder keine Anfragen an den Meta-Prozeß *timer* stellen oder dieser nicht bereit ist, ihre Anfragen zu beantworten. Angesichts dieser un-

geklärten Fragen hilft die Zeitbehandlung in der offiziellen Semantik nicht, die Beziehung zwischen SDL und der Zeit zu klären.

Bücher zu SDL:

Da die Z.100 eine sehr technische Beschreibung von SDL gibt, beziehen die meisten SDL-Anwender ihr Wissen über SDL aus Büchern. Wir zitieren im folgenden einige Aussagen zum SDL-Zeitkonzept aus Büchern, die sich mit SDL befassen.

R. Saracco et al.: Telecommunication Systems Engineering using SDL [SSR89]:
Seite 12:
The null time in event interchange and manipulation was not clearly stated anywhere in the recommendation but it became a common understanding in the SDL group.
Seite 13:
The most common approach has been to include information on time consumption in comments associated with each SDL construct where this information was felt to be needed.

O. Færgemand: Introduction to SDL, in Using Formal Description Techniques [Tur93]:
Seite 100:
Of course, only local assumptions based on reading a global time can be made by processes in a distributed system. How time proceeds in the interpretation of a process instance is left for the particular implementation.

R. Bræk and O. Haugen: Engineering Real Time Systems [BH93]:
Seite 96:
Each process consists of the input port and an extended finite state machine. ... For each signal it performs one transition which will take a short but undefined time.
Seite 127:
Tasks in SDL are often considered to have no time duration; the only places where time elapses are in the states.

A. Olsen et al: Systems Engineering Using SDL-92 [OFMP+94]:
Seite 102:
How time proceeds during the execution of a process is intentionally left for particular implementations. For verification purposes it may be more convenient to consider zero-time transitions, whereas for simulation a certain time consumption could be associated with execution of each language construct.

Wenn wir diese Aussagen aus der Z.100 und aus der Literatur zu SDL zusammenfassen, so wird deutlich, daß nicht oder nur sehr kurz darauf eingegangen wird, wie sich ein Anwender den zeitlichen Ablauf eines mit SDL modellierten Systems vorzustellen hat. Es wird nicht klar, welcher Zusammenhang zwischen dem Ablauf eines SDL-Systems und dem Fortschreiten der SDL-Systemzeit besteht. Es wird vorgeschlagen, daß sich der Fortschritt der Zeit an der Implementierung des Systems orientieren soll. Das Ziel einer formalen Spezifikationssprache ist es jedoch, ein abstraktes Modell des zu entwickelnden Systems zu

beschreiben und seine Eigenschaften ohne Bezug auf die spätere Implementierung anzugeben. Somit sollte das SDL-Zeitkonzept auf der Spezifikationsebene unabhängig von der späteren Implementierung des Systems sein.

2.3.3 Mögliche Interpretationen der SDL-Zeit – eine Diskussion mit SDL-Anwendern

Um einen Eindruck zu erhalten, wie SDL-Anwender mit dem Begriff Zeit umgehen, und um herauszufinden, ob unter SDL-Anwendern eine einheitliche Interpretation des SDL-Zeitbegriffs existiert, haben wir im Oktober 1995 eine Umfrage durchgeführt[5]. Wir haben das nachfolgende Beispiel mit einer Frage an Mitglieder einer SDL-Mailingliste und an uns bekannte SDL-Anwender gesendet und sie gebeten, uns ihre Interpretation der SDL-Zeit zu erläutern. Unter den Befragten waren auch Mitglieder der Standardisierungsgruppe SG10 der ITU-T für SDL. Das Beispiel macht deutlich, wie die Interpretation der Zeit das Verhalten eines SDL-Systems beeinflußt.

Das Beispiel

Abbildung 2.4 zeigt eine einfache SDL-Spezifikation[6], anhand derer sich die Auswirkung des Zeitbegriffs auf das Verhalten des spezifizierten Systems zeigen läßt.

Die beiden Prozesse $P1$ und $P2$ im Block $B1$ sind - abgesehen von den Signalnamen - gleich beschrieben: $P1$ ($P2$) wartet in Zustand *wait* auf das Eingabesignal r (s). Der Verbrauch des Signals r (s) initiiert einen Zustandsübergang, dessen einzige Aktion in der Ausgabe des Signals a (b) besteht. Der Prozeß gelangt anschließend wieder in den Zustand *wait*.

Ferner existiert ein SDL-Prozeß Q in Block $B2$, an den die Signale a und b gesendet werden:
Wenn Q das Signal a erhält, sendet es als Bestätigung ein Signal r an $P1$.
Wenn Q das Signal b erhält, sendet es als Bestätigung ein Signal s an $P2$.
Zwischen den Blöcken $B1$ und $B2$ existiert ein verzögerungsfreier Kanal c, über den die Signale gesendet werden. Die Prozesse sind jeweils über Signalwege mit diesem Kanal verbunden. Beim Start des SDL-Systems sendet Q die Signale r und s an $P1$ und $P2$.

[5] Dabei handelt es sich um keine repräsentative Umfrage.

[6] Abbildung 2.4 sowie die Abbildungen 6.1, 6.2, 6.3 in Abschnitt 6.1.1 und die Abbildungen A.2, A.3, A.4, A.5 in Anhang A wurden mit dem Werkzeug ObjectGEODE von VERILOG erstellt.

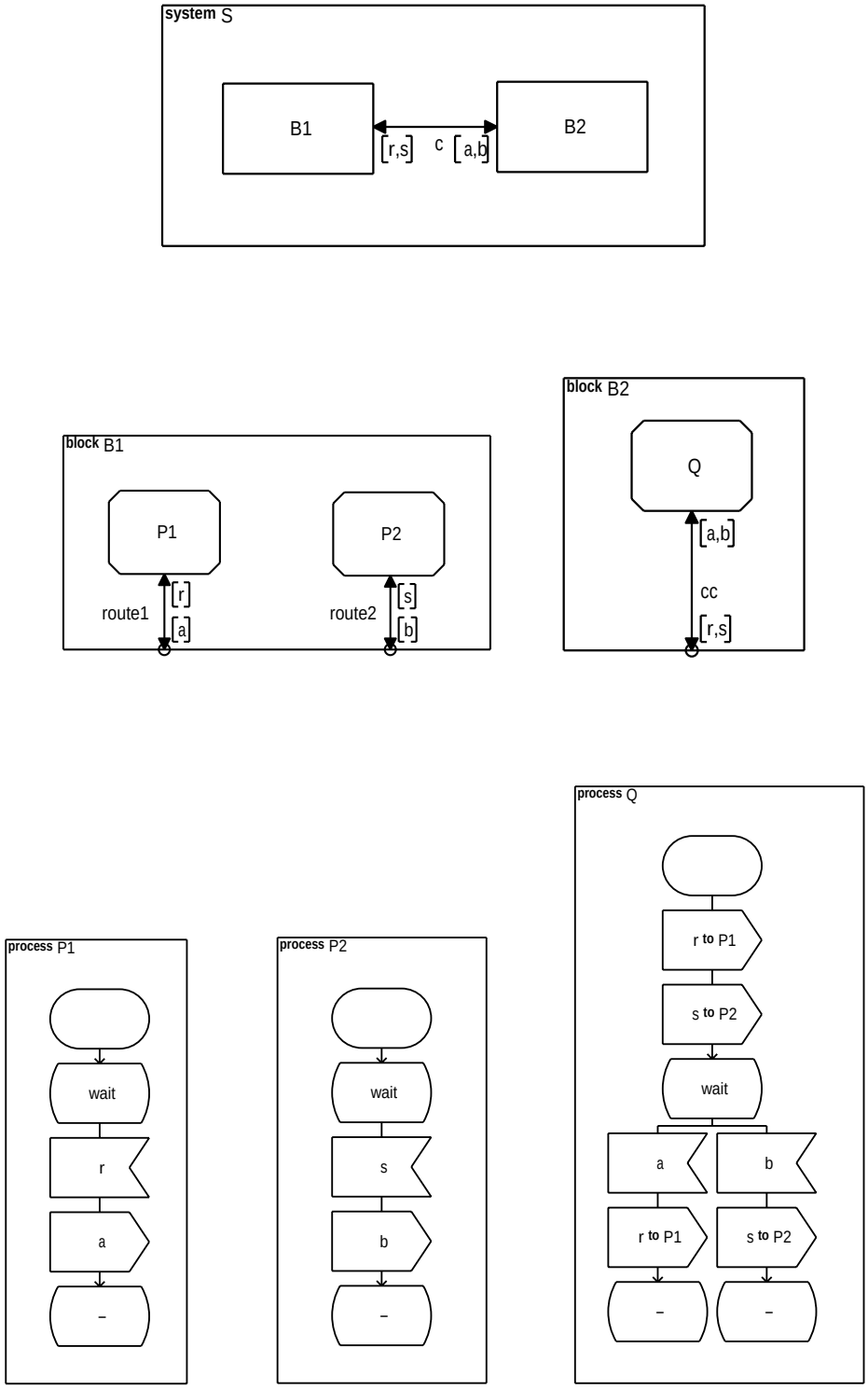


Abbildung 2.4: Das Beispiel – eine einfache SDL-Spezifikation

Die Frage

Die SDL-Prozesse $P1$ und $P2$ schreiben ihre Ausgabesignale auf denselben Kanal. Betrachten wir nun den Inhalt des Kanals c bezüglich der Signale a und b. Es stellt sich die Frage, welche der beiden folgenden Möglichkeiten für den Inhalt des Kanals (in Form von regulären Ausdrücken) zutrifft:

I: $(a \mid b)^*$
II: $(ab \mid ba)^*$

Trifft die erste Möglichkeit zu, so führen die Prozesse ihre Transitionen auf beliebige Weise durch. Es kann die Situation auftreten, daß Prozeß P1 zehnmal die Ausgabe von a durchführt, während Prozeß P2 im gleichen Zeitraum nur einmal das Signal b ausgibt. Wir setzen allerdings voraus, daß die Prozesse ihre Aktionen zwar beliebig lange, aber nur endliche Zeit verzögern. Das bedeutet, daß das Signal b auf dem Kanal enthalten sein muß. Läuft das beschriebene Prozeßsystem unendlich lange, so finden sich im Kanal unendlich viele a und unendlich viele b. Die Möglichkeit „I“ bedeutet, daß zwei Prozesse, die gleich spezifiziert sind, nicht unbedingt das gleiche Verhalten zeigen.

Bei der zweiten Möglichkeit verhalten sich beide Prozesse gleich, d. h. sie führen ihre Transitionen gleichzeitig aus, wenn sie beide zur gleichen Zeit ein Eingabesignal erhalten. Das Verhalten der Prozesse läuft synchronisiert ab. Die Ausgabesignale a und b werden gleichzeitig an den Kanal gereicht. In diesem Fall gilt, daß die Signale in beliebiger Reihenfolge auf den Kanal geschrieben werden.

Die Antworten

Insgesamt sind 19 Antworten auf unsere Frage eingegangen. Die meisten Beteiligten haben sich für Antwort I entschieden. Die Argumente für ihre Entscheidung reichen von persönlicher Erfahrung bis hin zur formalen SDL-Semantik der Z.100. Unter den Teilnehmern unserer Umfrage waren Mitglieder der SDL-Standardisierungsgruppe der ITU-T sowie SDL-Werkzeugentwickler und Wissenschaftler. Obwohl 19 Antworten nicht als repräsentativ bewertet werden können, zeigt sich an dieser Umfrage doch eindrucksvoll, wie uneins sich die SDL-Anwender bezüglich des Zeitbegriffs sind. Im folgenden klassifizieren wir die erhaltenen Erklärungen zur SDL-Zeit und diskutieren ihre Auswirkungen auf das Systemverhalten.

Grob lassen sich drei verschiedene Interpretationen der Zeit unterscheiden:

1. Die Systemzeit erhöht sich, wenn das System einen Stillstand erreicht hat. Die Systemzeit leitet sich somit aus dem Verhalten des Systems ab. Transitionen sind zeitlos.
2. Die Systemzeit schreitet für jeden Prozeß fort, sobald er sich in einem Zustand befindet. Die Zustandsübergänge sind zeitlos.
3. Die Systemzeit schreitet während des Systemablaufs unabhängig vom Ablaufverhalten des Systems fort. Transitionen und Ein- und Ausgabeaktionen verbrauchen Zeit.

Die folgende Tabelle gibt einen Überblick über die Antworten der befragten SDL-Anwender.

Zahl der Antworten:	19
Antwort I:	15
Antwort II:	4
Interpretation 1: Fortschritt der Zeit bei Systemstillstand	4
Interpretation 2: Fortschritt der Zeit in Zuständen	3
Interpretation 3: Permanenter Zeitfortschritt	9
Keine Begründung:	3

Interpretation 1

In der Empfehlung Z.100 wird von einer idealisierten Zeitvorstellung ausgegangen. Die Wartezeiten in einem System auf ein Signal sind um ein Vielfaches höher als die Ausführungszeiten der Aktionen in Zustandsübergängen. Die Ausführungszeiten von Aktionen können somit gegenüber den Wartezeiten vernachlässigt werden, so daß Zustandsübergänge in einem Prozeß zeitlos sind. Demzufolge verstreicht in einer SDL-Systemspezifikation Zeit, wenn das System einen Stillstand erreicht hat und alle Prozesse auf Signale warten, also wenn kein Zustandsübergang mehr ausführbar ist.
Bei dieser Interpretation des Begriffs Zeit handelt es sich nicht um Echtzeit, sondern um SDL-spezifische Systemzeit, die in Abhängigkeit des Systemverhaltens definiert ist. Wenn dieser Zeitbegriff zugrunde gelegt wird, so ist die Sprache SDL nicht dazu geeignet, Echtzeitanforderungen an ein System zu beschreiben.

An dem eben vorgestellten Zeitkonzept üben wir folgende Kritik:
Solange das System keinen Stillstand erreicht, schreitet die SDL-Systemzeit nicht fort. Ein SDL-System kann so modelliert sein, daß nie ein Stillstand erreicht wird. So können zum Beispiel zwei Prozesse existieren, die sich ständig gegenseitig Signale senden, so daß ein Pingpongeffekt hervorgerufen wird. Oder es liegt ein nichtterminierender Zustandsübergang vor, der eine unendliche Folge von Ausgabesignalen erzeugt. Wird kein Stillstand erreicht, so kann keine Zeit vergehen und auch kein Timer ablaufen. (Davon ausgenommen sind Timer, deren Ablaufzeitpunkt *NOW* ist.) Das hat zur Folge, daß sich bis auf einige wenige Prozesse alle Prozesse in einem Wartezustand befinden können, dieser Wartezustand aber nie durch den Ablauf eines Timers abgebrochen wird. Dies widerspricht der intuitiven Vorstellung, die mit Timern verbunden wird, nämlich Wartezeiten nach oben zu beschränken.

Hauptkritikpunkt an der Zeitinterpretation ist jedoch, daß diese Auffassung von Systemzeit der Idee eines verteilten Systems widerspricht. Wesentlicher Punkt bei der Modellierung eines verteilten Systems sollte der modulare Aufbau des Systems sein. Jede Komponente des Systems wird als abgeschlossene Einheit unabhängig von den anderen Komponenten beschrieben. Das Verhalten des Gesamtsystems leitet sich aus dem Verhalten der einzelnen Komponenten ab. Der eben vorgestellte Systemzeitbegriff hängt vom Verhalten der einzelnen SDL-Prozesse ab. Um diesen Zeitbegriff zu modellieren, ist ein „Beobachter“

erforderlich, der den Ablauf jedes einzelnen Prozesses verfolgt. Damit läßt sich das Verhalten des Gesamtsystems nicht mehr aus den lokalen Beobachtungen für jeden einzelnen Prozeß zusammensetzen. Das globale Wissen des Beobachters über den Zustand des Gesamtsystems läßt keine Kapselung der einzelnen SDL-Prozesse mehr zu - dies widerspricht der Idee eines verteilten Systems. Es handelt sich demnach nicht mehr um ein System, das aus unabhängig voneinander agierenden Prozessen besteht, die über Signale asynchron kommunizieren und lokale Zeitbegriffe verwenden, sondern um ein System, in dem die SDL-Prozesse einer zentralen Komponente für die Systemzeit untergeordnet sind.

Die eben vorgestellte Auffassung von SDL-Anwendern wird sicher durch die Verwendung von SDL-Werkzeugen beeinflußt, mit denen SDL-Spezifikationen erstellt und getestet werden. Das Werkzeug ObjectGEODE der Firma Verilog beispielsweise definiert die SDL-Zeit für die Simulation wie folgt ([Ver97]):

- Die Ausführung einer Transition erfolgt zeitlos.
- Die Zeit kann nur fortschreiten, wenn es mangels passender Eingabesignale keine ausführbare Transition gibt. Die Zeit schreitet dann solange fort, bis ein Timer abläuft.
- Wenn keine Transition ausführbar und kein Timer aktiv sind, kann die Zeit nicht fortschreiten; das System gerät in einen Verklemmungszustand.

Interpretation 2

Die Ausführung von Zustandsübergängen wird als zeitlos angesehen. Die Zeit erhöht sich für einen SDL-Prozeß, sobald der Prozeß einen Zustand erreicht hat. Dabei macht es keinen Unterschied, ob der Eingabepuffer beim Erreichen eines Zustands leer ist und der Prozeß damit sowieso auf das Eintreffen eines Signals warten muß oder ob bereits Signale im Eingabepuffer vorliegen, die einen weiteren Zustandsübergang ermöglichen. Mit dieser Interpretation ist das Fortschreiten der Zeit gesichert unter der Voraussetzung, daß die Zustandsübergänge des SDL-Prozesses terminieren.

Die Operation *NOW* liefert für jedes Vorkommen innerhalb eines Zustandsübergangs denselben Wert der Systemzeit. Die Zeit, die in einem Zustand verstreicht, ist endlich, aber unbestimmt und nicht Null. Andernfalls würde sich ein synchrones Verhalten der Prozesse ergeben, das nicht typisch für SDL ist.

Interpretation 3

Die Zeit schreitet ständig fort, d.h. die Zeit erhöht sich, während ein Prozeß Zustandsübergänge ausführt und während er in Zuständen ist. Es existiert eine globale Uhr, die die Zeit erhöht, für alle Prozesse zugänglich und maßgeblich für den Timermechanismus ist. Somit ist das Fortschreiten der Zeit unabhängig vom Verhalten der SDL-Prozesse; Zeit ist ein orthogonales Konzept zum Systemverhalten.

Daraus ergeben sich folgende Eigenschaften bezüglich des zeitlichen Verhaltens eines SDL-Systems:

- Die Systemzeit erhöht sich ständig während des Systemablaufs. Ein Prozeß, der während eines Zustandsübergangs mehrmals über den Ausdruck *NOW* die Systemzeit abfragt, kann verschiedene Zeitangaben erhalten.
- Die Ausführung eines Zustandsübergangs, genauer die Ausführung jeder Aktion in einem Zustandsübergang, verbraucht Zeit.
- Die Ausgabe eines Signals erfordert Zeit.
- Die Übertragung eines Signals ist zeitlos; ausgenommen sind Übertragungen über Kanäle mit Verzögerung.
- Ein Signal kann unbestimmte Zeit im Eingabepuffer eines Prozesses bleiben.
- Das Einlesen eines Signals aus dem Eingabepuffer in den Prozeß verbraucht Zeit.

Die Übertragung eines Signals über Signalwege erfolgt zwar zeitlos, die Ein- und Ausgabe eines Signals erfordert hingegen beliebig, aber endlich viel Zeit. Dieser Zeitverbrauch kann für jedes Signal und für jeden Prozeß bei jeder Ein- bzw. Ausgabe verschieden sein.
Die Auffassung von SDL-Anwendern, die SDL-Zeit mit Echtzeit gleichsetzen und in der SDL-Spezifikation Zeitangaben an physikalischen Gegebenheiten orientieren, ist ein Spezialfall dieser Zeitinterpretation.

Ergebnis der Umfrage

Anhand der Antworten auf unsere Umfrage wird deutlich, daß es hinsichtlich des Zeitbegriffs in SDL einen dringenden Klärungsbedarf gibt. SDL-Anwender ordnen einem mit SDL modellierten System unterschiedliche Verhaltensweisen zu. Dies ist für eine standardisierte Spezifikationssprache nicht tragbar. Einige SDL-Anwender haben noch auf folgenden Punkt hingewiesen:
Manche Aktionen eines SDL-Prozesses verbrauchen bei ihrer Ausführung so wenig Zeit, daß dieser Zeitverbrauch vernachlässigbar ist. Die Zeit, die ein Prozeß in Zuständen verbringt und auf Signale wartet, ist oft größer als die Zeit, die er für die Zustandsübergänge aufbringt. Dies ist charakteristisch für SDL-Prozesse. Daraus mag der falsche Eindruck entstanden sein, daß Zustandsübergänge zeitlos sind.

Einige SDL-Anwender sind der Meinung, daß zwischen der SDL-Spezifikation und der späteren Implementierung des Systems hinsichtlich der Zeitbehandlung ein Bruch entsteht. Deshalb sei es Aufgabe des SDL-Designers, das Verhalten der SDL-Prozesse und den Signalaustausch derart zu gestalten, daß das Systemverhalten unabhängig von der Ausführungsdauer der Zustandsübergänge eindeutig bestimmt werden kann.

In diesem Zusammenhang möchten wir auf folgendes Zitat aus G. J. Holzmann, Design and Validation of Computer Protocols (Seite 68, [Hol91]) aufmerksam machen:

Never make assumptions about the relative speeds of concurrent processes.

Das globale Zeitkonzept ist in unserer Umfrage nicht kritisiert worden. Lediglich eine Antwort enthielt die Bemerkung, daß für verteilte Systeme, wie sie SDL-Spezifikationen darstellen, lokale Uhren für die einzelnen Komponenten realistischer wären als eine globale Uhr.

Im nächsten Abschnitt werden wir ein Zeitkonzept vorstellen, das wir als adäquates Konzept für SDL-Spezifikationen betrachten.

2.3.4 Vorschlag für ein verbessertes SDL-Zeitkonzept

Im vorherigen Abschnitt haben wir verschiedene Interpretationen des SDL-Zeitbegriffs vorgestellt. Aufgrund der zahlreichen Mängel halten wir die erste Interpretationsmöglichkeit für keinen angemessenen Zeitbegriff für SDL. Insbesondere entspricht dieser Zeitbegriff nicht der Idee verteilter Systeme als Menge von unabhängigen, gekapselten Komponenten. Auch die zweite Interpretationsmöglichkeit weist Schwachstellen auf. Es erscheint nicht schlüssig, daß die Berechnungen während einer Transition ohne Zeitverbrauch erfolgen. Die meisten Befragten haben sich für die dritte Interpretationsmöglichkeit des Zeitbegriffs ausgesprochen. Dieser Zeitbegriff erscheint uns sinnvoll, da er der intuitiven Auffassung von Zeit entspricht und insbesondere nicht die Schwächen der anderen beiden Interpretationen aufweist. Die SDL-Systemzeit schreitet permanent und unabhängig vom Verhalten des Systems fort. Insbesondere erfordert die Ausführung von Zustandsübergängen Zeit. Der Zeitverbrauch bei der Interpretation eines SDL-Prozesses ist unbestimmt.

In diesem Abschnitt präsentieren wir eine Lösung des Zeitproblems in SDL, die auf der dritten Interpretation des Zeitbegriffs basiert.

Wir gehen von einer globalen, diskreten Systemzeit aus, die durch eine globale Uhr modelliert wird; die Zeit ist für alle SDL-Prozesse systemweit zugänglich. Die Zeit schreitet ständig fort. Die Ausführung eines SDL-Prozesses erfordert Zeit. Jedoch ist nichts über die Geschwindigkeit der Prozesse und über die Zeitdauer bekannt, die sie in einem Zustand oder in einem Zustandsübergang verbringen. Dies folgt auch aus der Tatsache, daß mit SDL die funktionalen Eigenschaften eines Systems auf hoher Abstraktionsebene unabhängig von der späteren konkreten Implementierung spezifiziert werden und somit die tatsächliche Rechenzeit auf der Zielplattform nicht bekannt ist. Daraus folgt, daß sich ein SDL-Prozeß hinsichtlich des Zeitverbrauchs nichtdeterministisch verhält. Betrachten wir einen SDL-Prozeß als Black-Box, so tritt eine beliebig lange Verzögerung größer Null zwischen der Eingabe eines Signals und der Ausgabe derjenigen Signale auf, die durch das Eingabesignal initiiert worden sind. Insbesondere liefert die Auswertung der Operation *NOW* in jedem Tasksymbol eines Zustandsübergangs einen anderen Wert der Systemzeit.

Es ist nicht vorhersehbar ist, wieviel Zeit die Ausführung von Zustandsübergängen und das Warten in Zuständen in Prozessen in Anspruch nehmen wird und zu welchem Zeitpunkt ein bestimmter Zustand im Prozeß erreicht sein wird. Deshalb ist es nicht sinnvoll, beim Setzen von Timern explizite Ablaufzeitpunkte anzugeben oder sich in Aktionen während eines Zustandsübergangs auf die aktuelle Systemzeit zu beziehen.

Timer sind Teil des Prozesses; sobald ein Timer abläuft, wird das Timersignal in den Eingabepuffer des Prozesses eingeordnet – die Zeit für die Zustellung ist somit Null. Andererseits werden Timersignale wie gewöhnliche Signale behandelt, d.h. sie werden am Ende des Eingabepuffers eingereiht und verbleiben unbestimmte Zeit im Puffer, bevor sie vom Prozeß als Eingabe verarbeitet werden. Somit vergeht zwischen dem Ablauf eines Timers und der Verarbeitung des Timersignals unbestimmt viel Zeit. Es ist auch deshalb nicht sinnvoll, den Ablaufzeitpunkt eines Timers relativ zur aktuellen Systemzeit anzugeben, da nach Ablauf des Timers nach dieser Zeitspanne nochmals eine beliebige Verzögerung hinzukommt.

Das Timerkonzept kann nur verwendet werden, um zu garantieren, daß nach einer endlichen Zeitdauer dem Prozeß ein Timersignal als Eingabe zur Verfügung steht. Somit kann durch die Verwendung von Timern die Zeit, die ein Prozeß in einem Zustand wartet, beschränkt und dadurch ein Aushungern von Prozessen verhindert werden. Wir schlagen vor, den komplizierten Timermechanismus zu vereinfachen, um ein derartiges Verhalten zu modellieren. Statt beim Setzen von Timern quantitative Ablaufzeitpunkte anzugeben, werden Timer nach dem Setzen beliebig lange verzögert, bevor sie ablaufen und in den Eingabepuffer des Prozesses eingereiht werden. Die Verzögerung der Timersignale ist dabei fair: es ist garantiert, daß ein gesetzter Timer nach endlicher Zeit abläuft und in den Eingabepuffer des SDL-Prozesses eingereiht wird.

In manchen Spezifikationssituationen mag eine Alternative zu diesem Timerkonzept darin bestehen, spontane Zustandsübergänge mittels des *none*-Konstrukts zu verwenden (siehe Abschnitt 2.1). Damit kann ebenfalls ein unendliches Warten auf ein Signal in einem Zustand verhindert werden. Allerdings entfällt damit die Vorstellung, daß sich ein Prozeß aktiv bemüht, einen Wartezustand zu vermeiden, indem er einen Timer setzt. Zudem ist in SDL nichts darüber ausgesagt, wie oft und nach welcher Zeit ein Prozeß einen spontanen Übergang ausführt – es kann nicht garantiert werden, daß solch ein Übergang tatsächlich in endlicher Zeit erfolgen wird (siehe kritische Anmerkung in Abschnitt 2.5).

Bei der Semantikdefinition für SDL in Kapitel 4 werden wir für die Zeitbehandlung das von uns vorgeschlagene vereinfachte Timerkonzept ohne quantitative Zeitangaben zugrunde legen.

2.4 Der Datenanteil in SDL

Ein SDL-Prozeß ist ein Zustandsautomat mit Ein- und Ausgabe, der zusätzlich über einen möglicherweise unendlichen Datenzustand verfügt. Der Datenzustand eines SDL-Prozesses setzt sich aus der Belegung der lokalen Variablen des Prozesses zusammen. Die Deklaration der Variablen erfolgt im Textsymbol des SDL-Prozesses. Die Variablen können bereits bei ihrer Deklaration oder während des Prozeßablaufs in einem Tasksymbol initialisiert werden. Während eines Zustandsübergangs kann der Datenzustand durch Anweisungen innerhalb von Tasksymbolen verändert werden. Das Entscheidungssymbol ermöglicht in Abhängigkeit des Datenzustands Verzweigungen des Kontrollflusses, so daß das Verhalten eines Prozesses durch seinen aktuellen Datenzustand beeinflußt werden kann.

Laut den offiziellen Empfehlungen der ITU-T basieren die SDL-Datentypen auf dem Konzept der abstrakten Datentypen (ADT). Das Konzept der abstrakten Datentypen wird von vielen Anwendern aufgrund seiner Komplexität abgelehnt. Es bietet allerdings den Vorteil, daß die Eigenschaften der Daten unabhängig von der späteren Implementierung des Systems beschrieben werden. Um den Umgang mit ADT für den SDL-Anwender einfacher zu gestalten, ist es mit SDL 92 möglich, bei der Definition eines ADT Funktionen graphisch in Form von sogenannten Operator Diagrams zu spezifizieren.

Bei genauerer Analyse des Datenaspekts in SDL stoßen wir auf mehrere kritische Punkte:

Lokale Datenräume:

Grundsätzlich sind Variablen lokal definiert, d.h. SDL-Prozesse haben keinen direkten Zugriff auf den Datenzustand anderer SDL-Prozesse. Dies ist ein charakteristisches Merkmal verteilter Systeme, deren Komponenten über Nachrichtenaustausch kooperieren – ein systemweiter globaler Datenraum ist nicht vorhanden. Es ist allerdings möglich, Signale mit Daten zu versehen, so daß über das Senden von Signalen ein Austausch von Daten zwischen den SDL-Prozessen möglich ist.
Über die Sprachkonstrukte *Export/Import* und *Reveal/View* wird es einem Prozeß ermöglicht, während eines Zustandsübergangs direkt auf die Werte von Variablen eines anderen Prozesses innerhalb des gleichen Blocks zuzugreifen[7]. Damit wird das Prinzip der prozeßeigenen lokalen Datenräume verletzt.

Partielle Funktionen:

Um Aussagen über das Verhalten eines Prozesses trefffen zu können, ist es wichtig zu untersuchen, welche Prozeßsymbole während der Interpretation des Prozesses erreicht und ausgeführt werden. In diesem Zusammenhang spielen die Terminierung von Anweisungen in Prozeßsymbolen und der Datenzustand eines Prozesses eine entscheidende Rolle – gerade wenn das Verhalten eines Prozesses während seiner Interpretation untersucht wird. Wichtiger Punkt beim Konzept der abstrakten Datentypen ist deshalb die Behandlung partiellen Verhaltens. Darunter verstehen wir die formale Behandlung von nichtterminierenden Berechnungen und von Funktionsanwendungen, denen kein definiertes Ergebnis zugeordnet werden kann. Ein typisches Beispiel hierfür ist die Division durch 0.
Die in SDL eingeführte Methode für die Definition abstrakter Datentypen unterstützt partielle Funktionen nur bedingt. In der informellen Beschreibung von SDL wird hierfür der Ausdruck *error* definiert:

„Ein error *taucht während der Interpretation einer SDL-Systembeschreibung auf, wenn eine der dynamischen Bedingungen von SDL verletzt wird. Sobald* error *aufgetreten ist, ist das weitere Verhalten des Systems nicht mehr definiert.“* (Seite 6 in Annex B, [IT93b])

Bei der Beschreibung der ADT in SDL wird *error* folgendermaßen verwendet: wird innerhalb eines Datentyps eine Funktion definiert, die für bestimmte Werte kein sinnvolles

[7] Der Gebrauch von *Reveal* und *View* wird in SDL 92 nicht mehr empfohlen.

Ergebnis liefert, so wird ihr für diese Eingabewerte der Ausdruck *error* als Ergebnis zugeordnet.

Beispiel: Division durch 0 in der Definition des Datentyps Integer:

```
mod : Integer, Integer -> Integer
...
a mod 0 == error!
```

Ergibt die Auswertung eines Ausdrucks in einem Tasksymbol *error*, so führt dies nicht nur zu einem undefinierten Verhalten des betroffenen SDL-Prozesses, sondern laut der Definition von *error* zu einem undefinierten Verhalten des gesamten Systems. Da sich das Systemverhalten aus den Verhalten der einzelnen SDL-Prozesse zusammensetzt, ist dies gleichbedeutend mit dem undefinierten Verhalten aller SDL-Prozesse. Diese Behandlung von partiellen Funktionen erscheint uns sehr ungünstig. Mit SDL werden verteilte Systeme spezifiziert, deren Komponenten, dargestellt durch die SDL-Prozesse, unabhängig voneinander agieren. Die SDL-Prozesse sind gekapselt und kommunizieren nur über Signale. Jeder Prozeß verfügt über einen eigenen Datenzustand; Information über den Datenzustand eines anderen Prozesses kann nur über Signalaustausch und das Exportieren/Importieren von Variablen erhalten werden. Es widerspricht somit dem Prinzip der Kapselung der einzelnen Komponenten, wenn die Interpretation eines Ausdrucks innerhalb einer Komponente unmittelbar Einfluß auf das Verhalten aller anderen Komponenten hat, ohne daß dazu ein Signalaustausch oder Variablenexport stattfindet. Denn dies setzt voraus, daß die Komponenten über Wissen bezüglich des Verhaltens und des lokalen Datenzustands einer anderen Komponente verfügen.

Gemäß den Eigenschaften von verteilten Systemen sollte das Verhalten von SDL-Prozessen nicht direkt von einer nichtterminierenden Berechnung eines anderen SDL-Prozesses beeinflußt werden. Es kann allerdings indirekt dadurch beeinflußt werden, daß der betroffene SDL-Prozeß nicht mehr auf Eingabesignale reagiert und somit keine Ausgabesignale mehr senden wird. Diese Situation ist bei der Spezifikation zu berücksichtigen. Die direkte Auswirkung auf andere Prozesse spiegelt zudem nicht das Verhalten wider, wie es bei verteilten Anwendungen in der Praxis auftritt. Fällt zum Beispiel in einem großen Vermittlungssystem eine Komponente aus, wird selbstverständlich erwartet, daß das restliche System korrekt weiterarbeitet.

Der zweite Fall für undefiniertes Verhalten eines SDL-Prozesses, nämlich Resultate von Abbildungen zu repräsentieren, für die ein assoziierter Algorithmus nicht terminiert, wird in der Beschreibung von SDL nicht angesprochen. Beispiele hierfür sind:

- Innerhalb eines Ausdrucks wird eine SDL-Prozedur aufgerufen, die rekursiv definiert ist und nicht terminiert.
- Innerhalb eines Ausdrucks wird eine SDL-Prozedur aufgerufen, in der eine Schleife vorkommt, die unendlich oft durchlaufen wird.
- Innerhalb eines Ausdrucks wird eine Funktion aufgerufen, die innerhalb eines Datentyps rekursiv definiert worden ist. Der Aufruf dieser Funktion terminiert nicht.

Ein wichtiges Ziel der semantischen Umsetzung des Datenanteils in SDL-Spezifikationen besteht darin, die eben vorgestellten Defizite des SDL-Datenanteils zu beheben. Dazu ist eine logische und mathematische Basis notwendig, in der sich partielle Funktionen und nichtterminierende Berechnungen explizit behandeln lassen. SPECTRUM ([BFG+93a]) bietet dazu geeignete Konzepte an.

Zugriff auf nicht-initialisierte Variablen:

Die Initialisierung von Variablen im Textsymbol eines SDL-Prozesses ist nicht zwingend festgelegt. Variablen können auch in Tasksymbolen während eines Zustandsübergangs initialisiert werden. Dies birgt eine Fehlerquelle. Erfolgt während der Prozeßinterpretation eine Zugriff auf eine Variable, der noch kein Wert zugewiesen ist, so führt dies laut der Z.100 zu einem nichtdefinierten Verhalten des gesamten Systems. Dies würde sich durch die Vorschrift, alle Variabeln bereits im Textsymbol zu initialisieren, leicht vermeiden lassen.

Informelle und programmiersprachliche Spezifikationsanteile:

Ein weiterer kritischer Punkt ist die Verwendung von informellen und programmiersprachlichen Anteilen in der Datenspezifikation.

- Obwohl SDL eine formale Beschreibungssprache ist, ist es zulässig, Teile einer SDL-Spezifikation informell anzugeben. Innerhalb eines SDL-Prozesses können in den Task- und Entscheidungssymbolen informelle Beschreibungsformen wie etwa natürlichsprachlicher Text verwendet werden.
- Neben den ADT können auch andere Techniken für die Definition der Datentypen in eine SDL-Spezifikation verwendet werden. So wird in der Empfehlung Z.105 ([IT95]) die Verbindung zwischen SDL und ASN.1 mit dem Ziel definiert, die Struktur und das Verhalten eines System mit SDL, den Datenanteil des Systems hingegen mit ASN.1 zu beschreiben. Programmiersprachliche Anteile, z.B. in C, können als externe Spezifikationen über das Schlüsselwort *alternative* in die Spezifikation des Datenanteils eingefügt werden. Innerhalb der SDL-Spezifikation entsprechen externe Datenanteile informellem Text (siehe Seite 164 in [IT93b]).

Sowohl informelle als auch programmiersprachliche Beschreibungen sind ein Teil einer Spezifikation, für den es keine formale Semantik gibt. Wir lehnen diese Beschreibungsformen in einer Spezifikation ab, da sie keine formale Semantik besitzen und somit keine Verifikation zulassen. Werden Programmiersprachen verwendet, so liegt zudem keine klare Trennung zwischen der Spezifikation und der Implementierung des Systems vor.

2.5 Analyse weiterer SDL-Sprachkonstrukte

Das Zeitkonzept und der Datenanteil von SDL beinhalten die gravierendsten Defizite der Spezifikationssprache SDL. Darüber hinaus weist SDL eine Reihe weiterer Schwachpunkte auf, die wir im folgenden aufführen.

- Das Konzept des Eingabepuffers und der Zugriff auf die Signale innerhalb des Puffers ermöglichen keine klare Vorstellung, welche Signale dem SDL-Prozeß in welcher Reihenfolge zur Verarbeitung zur Verfügung stehen. Signale, die nicht explizit als Eingabesignale spezifiziert sind, werden durch implizite Transitionen automatisch gelöscht, ohne daß dies in der Spezifikation explizit angegeben wird. Durch das Sprachelement *Save* kann das Löschen von Signalen zwar verhindert werden – dadurch wird jedoch das Prinzip der FIFO-Abarbeitung des Eingabepuffers durchbrochen.
  Es ist bei umfangreicheren Spezifikationen, bei denen ein Prozeß über mehrere Signalwege Signale erhält, schwer, einen Überblick zu behalten, welche Signale sich in welcher Reihenfolge im Eingabepuffer befinden können und in den Zustandsübergängen als Eingabesignale zu spezifizieren sind. Vorstellbar wäre die Einführung von Puffern für jeden einzelnen Signalweg, so daß ein selektiver Zugriff auf die Signale der einzelnen Signalwege ohne das Löschen von Signalen erfolgen kann. Dies würde zu eleganteren Spezifikationen führen.
- Es liegt keine Aussage hinsichtlich der Fairneßeigenschaften der nichtdeterministischen Sprachkonstrukte – spontane Übergänge und nichtdeterministische Entscheidungen – vor. Es ist nicht sichergestellt, daß bei einem Zustandsübergang mit einer nichtdeterministischen Entscheidung, der unendlich oft ausgeführt wird, jede vorliegende Entscheidungsmöglichkeit unendlich oft gewählt wird. Es ist möglich, daß bei jeder Ausführung dieselbe Entscheidung getroffen wird oder daß eine der Möglichkeiten nie ausgeführt wird. Gleiches gilt für Zustände mit spontanen Übergängen. Es ist nicht festgelegt, ob und wie oft ein SDL-Prozeß einen spontanen Übergang ausführt.
- Die graphische Darstellung eines Prozeßgraphen wird bei etwas umfangreicheren Spezifikationen rasch unübersichtlich. Sie erstreckt sich über mehrere Seiten. Ein Überblick über Zustände und Folgezustände und darüber, welche Eingabesignale zu welchen Folgezuständen führen, ist nicht möglich. Hier wäre ein Zustandsübersichtsdiagramm hilfreich.
- In den Prozessen fehlt eine klare Trennung zwischen Kontroll- und Datenaspekt. In den Prozeduren ist ein Zugriff auf den Eingabepuffer des Prozesses möglich – der Kontrollfluß teilt sich somit zwischen Prozeß und Prozedur auf. Der Datenanteil ist mit dem Interaktionsteil des Prozesses vermischt und im Kontrollflußdiagramm enthalten. Hier wäre eine stärkere Kapselung des Datenanteils wünschenswert.
- Ein SDL-Prozeß läßt sich nicht hierarchisch strukturieren, wie dies beispielsweise bei Statecharts ([Har87]) möglich ist. Mit dem Sprachkonstrukt *Service* wird nur eine horizontale Aufteilung des Prozesses erreicht, wobei die einzelnen Services auf einen Eingabepuffer zugreifen und nicht nebenläufig abgearbeitet werden.

- Die syntaktische Darstellung der Strukturierungsebenen eines Systems ist in SDL sehr starr. Selbst wenn ein System nur einen Block beinhaltet, ist die Trennung zwischen System- und Blockebene vorzunehmen, da für die unterschiedlichen Strukturierungsebenen verschiedene syntaktische Darstellungen zu verwenden sind.

Die Liste von Schwachpunkten macht deutlich, daß SDL einige Sprachkonzepte beinhaltet, die eine abstrakte und klare Spezifikationserstellung beeinträchtigen.

2.6 SDL – eine semiformale Spezifikationssprache

In den vorangegangenen Abschnitten haben wir die SDL-Semantikdefinition der Z.100 sowie den Zeit- und den Datenaspekt von SDL analysiert, wobei wir insbesondere die Frage untersucht haben, ob es sich bei SDL um eine formale Beschreibungssprache handelt. Zusammenfassend läßt sich diese Frage mit einem klaren Nein beantworten, wobei wir folgende Begründungen anführen:

- SDL besitzt keine formale, auf mathematischen und logischen Konzepten basierende Semantik (siehe Abschnitt 2.2).
- Die informelle Beschreibung des zeitlichen Verhaltens eines mit SDL spezifizierten Systems ist unklar und in sich widersprüchlich (siehe Abschnitt 2.3).
- Das Konzept der abstrakten Datentypen, wie es in SDL verwendet wird, beinhaltet keine adäquate Behandlung partieller Funktionen, woraus sich Verhaltensmuster ergeben, die die charakteristischen Eigenschaften verteilter Systeme verletzen (siehe Abschnitt 2.4).

Da SDL jedoch über eine klar definierte, sogar standardisierte Syntax sowohl für die graphische als auch für die textuelle Darstellung von Spezifikationen verfügt, erfüllt es die Voraussetzung einer semiformalen Beschreibungssprache.

Um SDL als formale Spezifikationssprache verwenden zu können, muß eine formale Semantik mit mathematisch präzisen Mitteln erstellt werden, wobei die Bedeutung der bislang nur unvollständig und widersprüchlich definierten Sprachanteile von SDL klar und im Sinne einer formalen Sprache für verteilte Systeme festzulegen ist. In Kapitel 4 werden wir eine solche formale Semantik im Rahmen der formalen Entwicklungsmethodik FOCUS definieren.

Kapitel 3

Focus – eine formale Entwicklungsmethodik

In diesem Kapitel stellen wir die semantische Basis vor, mit der wir Basic SDL formal fundieren werden. FOCUS ([BDD+93]) ist eine formale Entwicklungsmethodik für verteilte, reaktive Systeme, die über eine wohldefinierte formale Semantik verfügt und eine Reihe von formalen Spezifikations- und Beweistechniken anbietet. Die Semantik von Spezifikationen in FOCUS ist denotationell und beruht auf Konzepten der Bereichstheorie ([GS90]): stromverarbeitende Funktionen beschreiben das Verhalten einer Spezifikation.

In FOCUS werden verteilte Systeme als Netzwerke von Komponenten modelliert. Die Komponenten kommunizieren asynchron durch Nachrichtenaustausch über unbeschränkte, gerichtete Kanäle. Ströme werden verwendet, um die Kommunikationsgeschichte von Kanälen zu modellieren. Die Ströme bestehen dabei aus den Nachrichten, die die Komponenten untereinander und mit der Systemumgebung austauschen.

Für die formale Fundierung von SDL werden wir die Beschreibungssprache ANDL ([SS95]) verwenden, die eine Schicht über dem semantischen Modell von FOCUS liegt und den Anwendern somit eine Verwendung der Stromsemantik ohne direkten Zugriff auf die semantischen Konzepte ermöglicht. Bisher liegt mit ANDL eine Sprache vor, mit der das Verhalten von Systemen ohne zeitlichen Bezug spezifiziert werden kann. Da jedoch bei der Semantikdefinition von SDL die Modellierung von Zeitaspekten eine wesentliche Rolle spielen wird, erweitern wir im folgenden ANDL um die Möglichkeit, zeitliches Verhalten zu spezifizieren. Damit ist ein Wechsel des zugrundeliegenden semantischen Strommodells verbunden, den wir ebenfalls in diesem Kapitel vorstellen werden. Für die Spezifikation des Datenanteils in FOCUS-Spezifikationen werden wir die algebraische Spezifikationssprache SPECTRUM ([BFG+93a]) verwenden, deren Semantik wie FOCUS auf bereichstheoretischen Konzepten basiert und sich deshalb für eine Einbindung in FOCUS gut eignet.

Das Kapitel ist wie folgt aufgebaut: Zunächst stellen wir grundlegende Konzepte von FOCUS vor. Dazu zählen Ströme, stromverarbeitende Funktionen und das Zeitkonzept. Anschließend folgt eine kurze Beschreibung von Syntax und Semantik der um Zeitaspekte erweiterten Sprache ANDL. Den Abschluß bildet eine kurze Einführung in SPECTRUM.

Wir werden in diesem Kapitel nur solche Aspekte von FOCUS bzw. ANDL und SPECTRUM behandeln, die notwendig sind, um die Definition der SDL-Semantik und die darauf basierende Verifikationsmethode zu verstehen, und für darüber hinausgehende Aspekte Literaturhinweise angeben. Für eine grundlegende Einführung in FOCUS verweisen wir auf [BS98, BDD+93]; eine anwendungsorientierte Einführung in FOCUS gibt [Spi98].

3.1 Grundlegende Konzepte von Focus

Im FOCUS besteht ein System aus einem Netz von interagierenden Komponenten, die durch asynchronen Nachrichtenaustausch über gerichtete Kanäle kommunizieren. Dabei sind jedem Kanal genau eine sendende und eine lesende Komponente zugeordnet (point-to-point Kommunikation). Zur Modellierung der Kommunikation zwischen den Komponenten werden gezeitete Nachrichtenströme verwendet. Ein Strom enthält alle Nachrichten, die über einen Kanal gesendet werden, sowie Information darüber, in welchem Zeitintervall eine Nachricht übertragen wird. In FOCUS wird von einer globalen, diskreten Systemzeit ausgegangen. Das Voranschreiten der Zeit wird durch das Einfügen von speziellen Symbolen $\surd$, sogenannten Zeitticks, in den Nachrichtenströmen dargestellt. Die Zeitticks kennzeichnen innerhalb der Ströme das Ende der Zeitintervalle. Wir gehen davon aus, daß alle Zeitintervalle gleich groß sind und nur endlich viele Nachrichten enthalten. Abbildung 3.1 zeigt die Einteilung eines gezeiteten Stroms in eine Folge von Zeitintervallen, die durch $\surd$ abgeschlossen sind. Zwei unmittelbar aufeinanderfolgende $\surd$ bedeuten, daß während des Zeitintervalls, das durch diese $\surd$ umschlossen wird, keine Nachrichten auf dem Kanal übertragen werden. Da die Zeit nie stehenbleibt, wird die vollständige Kommunikationsgeschichte eines Kanals durch einen unendlich langen Strom mit unendlich vielen Zeitticks modelliert.

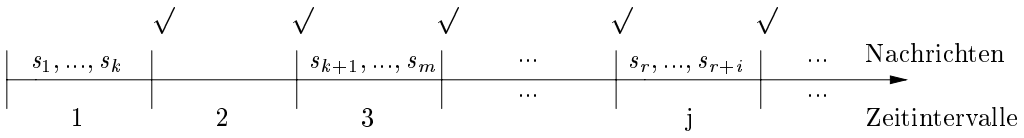


Abbildung 3.1: Gezeiteter Strom

Gezeitete Ströme über einer Nachrichtenmenge N werden definiert durch

$$N^{\underline{\omega}} = N^{\underline{*}} \cup N^{\underline{\infty}}$$

Dabei bezeichnet $N^{\underline{*}}$ die Menge aller endlichen gezeiteten Ströme, $N^{\underline{\infty}}$ die Menge aller unendlichen gezeiteten Ströme und $N^{\underline{\omega}}$ die Menge aller gezeiteten Ströme beliebiger Länge. Es gilt, daß ein unendlicher gezeiteter Strom unendlich viele Zeitticks enthält.

Spielt die Modellierung zeitabhängigen Verhaltens in einer Spezifikation keine Rolle, so genügt es, ungezeitete Ströme, also Ströme ohne Zeitticks, zu verwenden: $N^{\omega} = N^{*} \cup N^{\infty}$. Dabei bezeichnen N^{*} die Menge aller endlichen und N^{∞} die Menge aller unendlichen ungezeiteten Ströme; N^{ω} umfaßt die Menge aller ungezeiteten Ströme beliebiger Länge.

Für beliebige Ströme $s, t : N^\omega \cup N^{\underline{\infty}}$ definieren wir folgende Operationen:

$\langle\rangle$	bezeichnet den leeren Strom.
$\langle a \rangle$	bezeichnet den Strom, der nur aus der Nachricht $a : N \cup \{\surd\}$ besteht. Ist der Kontext klar, so wird statt $\langle a \rangle$ auch a geschrieben.
$s \circ t$	bezeichnet den Strom, der sich aus der Konkatenation der Ströme s und t ergibt. Es gilt: $s \circ t = s$, falls s unendlich ist.
$a\&s$	fügt die Nachricht $a : N \cup \{\surd\}$ an den Anfang des Stroms s an.
$ft(s)$	bezeichnet die erste Nachricht von Strom s. Es gilt: $ft(s)$ ist undefiniert, falls s der leere Strom ist.
$rt(s)$	bezeichnet den Rest von Strom s, also den Strom s ohne seine erste Nachricht $ft(s)$. Es gilt: $rt(s) = \langle\rangle$, falls $s = \langle\rangle$.
$M \copyright s$	bezeichnet den Strom, der nur aus denjenigen Nachrichten aus s besteht, die in der Menge M enthalten sind (Filteroperator).

Für gezeitete Ströme s stehen zusätzlich folgende Funktionen zur Verfügung:

$s\downarrow_j$	liefert für einen Strom $s : N^{\underline{\infty}}$ den Teilstrom bis zum Zeitpunkt j (kleinstes Präfix von s mit j Zeitticks)
$\overline{s}$	entfernt alle $\surd$ aus $s : N^{\underline{\omega}}$ (Zeitabstraktion)

Neben Strömen werden auch Tupel von Strömen betrachtet, um die Kommunikationsgeschichten mehrerer Kanäle zu erfassen. Wir schreiben hierfür $N_1^\omega \times \ldots \times N_m^\omega$ oder auch $(N^\omega)_{1:m}$ (siehe [Fuc94]). Eine analoge Schreibweise gilt für die anderen Typen von Strömen. Die oben eingeführten Funktionen auf Strömen werden entsprechend auf Tupel von Strömen erweitert.

Um das Verhalten einer Komponente mit m Eingabekanälen und n Ausgabekanälen zu spezifizieren, wird die Beziehung zwischen den Kommunikationsgeschichten ihrer Eingabe- und Ausgabekanäle spezifiziert. Dies erfolgt durch die Angabe einer stromverarbeitenden Funktion, die ein m-stelliges Tupel von Strömen auf ein n-stelliges Tupel von Strömen abbildet:

$f : (N^\omega)_{1:m} \to (N^\omega)_{1:n}$	ohne Berücksichtigung zeitlichen Verhaltens
$f : (N^{\underline{\infty}})_{1:m} \to (N^{\underline{\infty}})_{1:n}$	bei Berücksichtigung zeitlichen Verhaltens

Da die Zeit nie stehenbleibt, wird bei Spezifikationen mit Zeitaspekten über Funktionen mit unendlichen gezeiteten Strömen argumentiert.

Es ist möglich, stromverarbeitende Funktionen zusätzlich mit Argumenten zu parametrisieren, die keine Ströme sind, um damit zum Beispiel den lokalen Datenzustand einer Komponente zu modellieren.

Für stromverarbeitende Funktionen fordern wir semantische Eigenschaften wie Stetigkeit und Monotonie bzgl. der Präfixordnung auf Strömen. Durch die Monotonie wird garantiert, daß ausgegebene Nachrichten nicht mehr zurückgenommen oder verändert werden können.

Stetigkeit garantiert, daß sich das Verhalten einer Funktion vollständig durch ihr Verhalten auf endlichen Strömen beschreiben läßt. Speziell für stromverarbeitende Funktionen auf unendlichen gezeiteten Strömen fordern wir die Eigenschaft der Pulsgetriebenheit und sprechen dann von pulsgetriebenen Funktionen:

$$\text{starke Pulsgetriebenheit:} \quad x\downarrow_i = y\downarrow_i \implies f(x)\downarrow_{i+1} = f(y)\downarrow_{i+1}$$

$$\text{schwache Pulsgetriebenheit:} \quad x\downarrow_i = y\downarrow_i \implies f(x)\downarrow_i = f(y)\downarrow_i$$

Dies bedeutet, daß der Eingabestrom bis zum Zeitpunkt i die Ausgabe der Funktion bis zum Zeitpunkt $i+1$ bei starker bzw. i bei schwacher Pulsgetriebenheit vollständig festlegt. Daraus folgt, daß die Funktion ihre Eingabe nicht vorhersehen kann und die Zeit nicht rückwärts schreitet. Bei starker Pulsgetriebenheit benötigt die Komponente zur Verarbeitung von Nachrichten mindestens ein Zeitintervall, die Komponente reagiert also mit Verzögerung auf die Eingabenachrichten. Bei schwacher Pulsgetriebenheit verarbeitet die Komponente ihre Eingabenachrichten ohne Verzögerung.

3.2 Die logische Kernsprache Andl

ANDL ist eine leicht handhabbare, verständliche Spezifikationssprache, die in programmiersprachlicher Notation einen festen syntaktischen Rahmen für die Erstellung von Spezifikationen in FOCUS vorgibt. Damit ist es auch Anwendern, die mit formalen, mathematisch basierten Notationen und Formalismen nur wenig vertraut sind, möglich, in einem formalen Rahmen zu spezifizieren, ohne mathematisch komplexe Darstellungen verwenden zu müssen. Die Semantik von ANDL ist durch die formale Basis von FOCUS gegeben und wird in der Logik höherer Stufe für berechenbare Funktionen, HOLCF ([Reg94]), formalisiert. Dadurch ist für ANDL ein Anschluß an den Theorembeweiser Isabelle ([Pau94]) gegeben – Beweise über ANDL-Spezifikationen können maschinenunterstützt geführt werden.

Im folgenden definieren wir eine Erweiterung von ANDL, die die Spezifikation von Systemen mit zeitlichen Aspekten erlaubt und auf einem gezeiteten semantischen Strommodell basiert. Somit ist diese Variante von ANDL eine Erweiterung von [SS95] um zeitliche Aspekte auf syntaktischer und semantischer Ebene. Im weiteren bezeichnen wir diese Erweiterung mit ANDL. In Abschnitt 3.2.1 stellen wir die Spezifikationsformate von ANDL vor, in Abschnitt 3.2.2 beschreiben wir das gezeitete semantische Modell von ANDL. Eine Formalisierung dieses semantischen Modells in HOLCF führen in Abschnitt 6.3 durch.

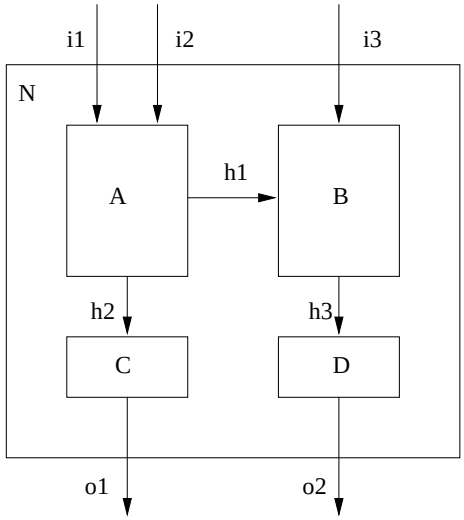
3.2.1 Andl – Syntax

Die Sprache ANDL unterscheidet zwischen Netzwerk- und Basiskomponenten. Netzwerkkomponenten beschreiben die Struktur eines Systems oder einer Komponente, Basiskomponenten das Verhalten einer Komponente, die nicht weiter strukturiert wird. Bei beiden Arten von Komponenten wird eine klare Beschreibung der Schnittstelle gegeben, die sich

aus den Ein- und Ausgabekanälen mit den dazugehörigen Nachrichtentypen zusammensetzt.

Netzwerkkomponenten

Die Beschreibung einer Netzwerkkomponente besteht aus der Schnittstellenbeschreibung sowie aus der Netzwerkstrukturbeschreibung der Komponente. Diese setzt sich aus einer Menge von Gleichungen der Form `<Ausgabekanäle> = Komponente <Eingabekanäle>` zusammen. Die Gleichungen geben die Kanalverbindungen der Komponenten des Netzwerkes untereinander und mit der Umgebung an.



agent N
input channel $i1 : I1,\ i2 : I2,\ i3 : I3$
output channel $o1 : O1,\ o2 : O2$
is network

$$\begin{aligned} < h1, h2 > &= A < i1, i2 >; \\ < h3 > &= B < i3, h1 >; \\ < o1 > &= C < h2 >; \\ < o2 > &= D < h3 > \end{aligned}$$

end N

Abbildung 3.2: Spezifikation einer Netzwerkkomponente mit ANDL

Abbildung 3.2 zeigt links ein sogenanntes Systemstrukturdiagramm ([HSS96]), das die Struktur der Komponente N, die sich aus den Komponenten A, B, C und D zusammensetzt, graphisch beschreibt. Diese Komponente wird in ANDL als Netzwerk spezifiziert. Die in der Schnittstelle aufgeführten Kanäle werden als Ein- und Ausgabekanäle bezeichnet und verbinden die Komponenten des Netzwerks mit der Umgebung. $I1$, $I2$, $I3$ bzw. $O1$, $O2$ sind die Nachrichtentypen der jeweiligen Ein- bzw. Ausgabekanäle. In den Netzwerkgleichungen stellen $h1$, $h2$ und $h3$ interne Kanäle dar, die nicht in der Schnittstellenbeschreibung des Netzwerks N aufgeführt sind und die Komponenten innerhalb des Netzwerks verbinden. Die Komponenten von N können selbst wieder Netzwerk- oder Basiskomponenten sein.

Basiskomponenten

Eine Komponente, die selbst nicht als verteiltes System beschrieben werden soll, stellt eine Basiskomponente dar. Eine Basiskomponente ist mittels Ein- und Ausgabekanälen mit ihrer Umgebung verbunden. In ANDL besteht die Beschreibung einer Basiskomponente aus einer Schnittstellen- und einer Verhaltensbeschreibung. Das Verhalten ist durch

die Beziehung zwischen den Ein- und Ausgabenachrichten der Komponente in Form von stromverarbeitenden Funktionen definiert. Üblicherweise geschieht dies durch die Angabe eines Prädikats, das eine Menge von stromverarbeitenden Funktionen festlegt, die die Eigenschaften der Basiskomponente spezifizieren.

In einem System mit zeitlichen Verhaltensanteilen brauchen nicht alle Komponenten zeitabhängiges Verhalten aufweisen, so daß innerhalb einer Systemspezifikation zeitabhängige und zeitunabhängige Komponenten nebeneinander existieren können. Wir unterscheiden deshalb zwei Spezifikationsformate für Basiskomponenten. In dem in Abbildung 3.3 gegebenen Schema ist A der Bezeichner der Basiskomponente. Durch die auf den Bezeichner A folgenden Schlüsselworte `time dependent` und `time independent` wird angegeben, ob es sich bei A um eine Komponente mit zeitabhängigem oder zeitunabhängigem Verhalten handelt. Mit `wp` bzw. `sp` wird zusätzlich angegeben, ob sich eine gezeitete Komponente schwach oder stark pulsgetrieben verhält.

agent A – [**time independent** | **time dependent** – [**wp** | **sp**]]
input channel $i_1 : I_1, \ \ldots, \ i_m : I_m$
output channel $o_1 : O_1, \ \ldots, \ o_n : O_n$
is basic

$R(i_1, \ldots, i_m, o_1, \ldots, o_n)$

end A

Abbildung 3.3: Spezifikation von Basiskomponenten mit ANDL

Für beide Varianten gilt: i_1 bis i_m bzw. o_1 bis o_n sind die Bezeichner der Ein- bzw. Ausgabekanäle und I_1 bis I_m bzw. O_1 bis O_n die Bezeichner der Nachrichtentypen. Das Verhalten der Komponente wird durch das Prädikat R definiert, das die Beziehung zwischen den Ein- und Ausgabeströmen der Komponente festlegt.

Bei *zeitunabhängigen* Basiskomponenten wird das Verhalten ohne Bezug auf die Zeitinformation in den Strömen spezifiziert. Innerhalb des Prädikats R werden nur ungezeitete Ströme verwendet. Demnach handelt es sich bei den Strömen in R um die Ein- und Ausgabeströme der Komponente ohne Zeitinformation, also um endliche oder unendliche ungezeitete Ströme. Das bedeutet, daß die Komponente gezeitete Eingabeströme erhält und gezeitete Ausgabeströme ausgibt, es jedoch ausreicht, ihr Ein-/Ausgabeverhalten unabhängig von der Zeitinformation in den Strömen zu spezifizieren. Damit darf die Komponenten zwischen einer Eingabenachricht und den dazugehörigen Ausgabenachrichten beliebig viel Zeit verstreichen lassen.

Bei *zeitabhängigen* Basiskomponenten wird in der Spezifikation Gebrauch von der Zeitinformation in den Strömen gemacht. Innerhalb des Prädikats R werden explizit Zeitticks verwendet, um zeitliches Verhalten zu spezifizieren; das Prädikat verwendet somit die gezeiteten Ein- und Ausgabströme der Komponente. Mit den Schlüsselworten `sp` bzw. `wp` wird festgelegt, ob die Verarbeitung von Eingabenachrichten mit oder ohne zeitliche Verzögerung erfolgt.

Anmerkung:

In der Schnittstellenbeschreibung der Komponente repräsentieren die Bezeichner $i_1, \ldots, i_m$ und $o_1, \ldots, o_n$ die Kanäle, über die die Basiskomponente Nachrichten erhält bzw. sendet. Damit ist jedem Kanal ein Nachrichtenstrom zugeordnet. Wird innerhalb des Prädikats R auf diese Nachrichtenströme Bezug genommen, so werden dazu ebenfalls die Bezeichner $i_1, \ldots i_m$ und $o_1, \ldots o_n$ verwendet.

Unabhängig vom Spezifikationsformat werden in der Verhaltensbeschreibung R die semantischen Eigenschaften der Komponente, also die Beziehung zwischen ihren Ein- und Ausgabenachrichten beschrieben. Dabei werden wir bei der Semantikdefinition von SDL überwiegend einen konstruktiven Spezifikationsstil auf der Basis von rekursiven Funktionsgleichungen verwenden (siehe Abschnitt 4.3.2).

3.2.2 Andl – Semantik

Das semantische Modell von ANDL beruht auf stark pulsgetriebenen Funktionen, d.h. auf semantischer Ebene wird ausschließlich über unendliche gezeitete Ströme argumentiert. Für jede Komponente der ANDL-Spezifikation wird die Semantik durch ein Prädikat bestimmt, das aus der Komponentenspezifikation abgeleitet wird. Wir bezeichnen die Semantik einer Komponente A mit $[\![A]\!]$. Das Prädikat definiert all diejenigen pulsgetriebenen Funktionen, die ein zulässiges Verhalten von A beschreiben. Der Vorteil des Modells liegt darin, daß sowohl Komponenten mit zeitabhängigem als auch mit zeitunabhängigem Verhalten integriert werden.
Wir beschreiben nun informell, wie aus den Spezifikationen der Netzwerke und Basiskomponenten die dazugehörigen Prädikate abgeleitet werden. Die Formalisierung des semantischen Modells in der Logik HOLCF wird in Abschnitt 6.3 vorgestellt.

Semantische Umsetzung von Netzwerkkomponenten:

Aus der Schnittstellenbeschreibung und den Netzwerkgleichungen sowie dem Verhalten der einzelnen Systemkomponenten wird ein Prädikat über stark pulsgetriebenen Funktionen abgeleitet. Dabei wird gefordert, daß für jede Belegung der Ein- und Ausgabekanäle des gesamten Netzwerks eine Belegung der internen Kanäle existiert, die das aus den Netzwerkgleichungen abgeleitete Gleichungssystem erfüllt. Die Semantik einer Netzwerkkomponente ist die Menge aller stark pulsgetriebenen Funktionen, die dieses Prädikat erfüllen.

Semantische Umsetzung von Basiskomponenten:

Aus der Beschreibung der syntaktischen Schnittstelle und aus dem Rumpf der Basiskomponente wird ein Prädikat abgeleitet, das diejenigen Funktionen charakterisiert, welche die in der Verhaltensbeschreibung der Komponente festgelegten Eigenschaften erfüllen. Dabei wird zwischen den Spezifikationsformaten unterschieden:

Die Semantik einer *zeitabhängigen Basiskomponente* A ist die Menge aller pulsgetriebenen Funktionen, welche die im Prädikat R festgelegten Eigenschaften der Komponente erfüllen.

Ist das Schlüsselwort `sp` angegeben, so handelt es sich um stark pulsgetriebene Funktionen, bei Angabe von `wp` um schwach pulsgetriebene Funktionen[1].

Die Semantik einer *zeitunabhängigen Basiskomponente A* ist die Menge aller stark pulsgetriebenen Funktionen, welche nach der Abstraktion der Zeitticks in ihren Ein- und Ausgabeströmen die im Prädikat R festgelegten Eigenschaften des Komponentenrumpfes erfüllen.

Semantik rekursiver Funktionen:

Die Semantik von rekursiv definierten Funktionen wird in der Bereichstheorie durch kleinste Fixpunkte definiert. Da unser semantisches Modell auf stark pulsgetriebenen Funktionen basiert, deren Ein- und Ausgabeströme per Definition unendliche gezeitete Ströme sind, gilt, daß stark pulsgetriebene Funktionen, die rekursiv definiert sind, nicht nur einen kleinsten, sondern einen einzigen und damit eindeutigen Fixpunkt besitzen (Satz von Knaster und Tarski sowie die partielle Ordnung $\sqsubseteq$ auf unendlichen Strömen). Diese Eigenschaft werden wir bei der Verifikation von SDL-Spezifikation verwenden (siehe Abschnitt 6.1.4).

3.3 Die Spezifikation des Datenanteils mit Spectrum

In FOCUS ist keine Beschreibungstechnik für den Datenanteil eines Systems vorhanden, da die Behandlung von Daten in FOCUS eine untergeordnete Rolle spielt. Deshalb werden wir bei der Semantikdefinition von SDL für den Datenaspekt die algebraische Spezifikationssprache SPECTRUM ([BFG+93a]) verwenden. SPECTRUM basiert auf den Konzepten der algebraischen Spezifikation, wie sie in [Gut75] entwickelt worden sind, und der Bereichstheorie ([GS90]). Im folgenden geben wir eine kurze Einführung in die wesentlichen Konzepte von SPECTRUM. Eine ausführliche Beschreibung findet sich in [BFG+93a], eine Einführung in abstrakte Datentypen in [PBB+82].

Die Spezifikation eines abstrakten Datentyps (kurz ADT) setzt sich aus der Signatur und einer Menge von Axiomen zusammen. In der Signatur werden die Sorten und Funktionen deklariert, im Axiomenteil wird das Verhalten der aufgeführten Funktionen durch eine Menge von Gesetzen abstrakt beschrieben. Dabei stehen in SPECTRUM nicht nur Gleichungen zur Verfügung, sondern die Prädikatenlogik erster Stufe und ein Termerzeugungsprinzip. Für die nachfolgende Semantikdefinition von SDL genügen jedoch Gleichungsspezifikationen, so daß wir uns auf diesen Aspekt von SPECTRUM beschränken.

Ein wichtiges Konzept von SPECTRUM ist die Behandlung partieller Funktionen. Dabei handelt es sich um Funktionen, denen für bestimmte Argumente kein sinnvolles Resultat zugewiesen werden kann. Dafür wird das Element $\perp$ eingeführt. $\perp$ ist ein Pseudo-Element und steht für nicht existierende Werte. Funktionen ohne sinnvolles Resultat sind zum Beispiel Funktionen, die für gewisse Eingaben nicht terminieren; sie erhalten für diese Eingaben als Ergebnis das Element $\perp$ zugewiesen.

[1] Die Komposition von schwach mit stark pulsgetriebenen Funktionen liefert stark pulsgetriebene Funktionen. Dadurch lassen sich schwach pulsgetriebene Funktionen in unser semantisches Modell integrieren, sofern sie in Komposition mit stark pulsgetriebenen Funktionen verwendet werden.

Um einen Eindruck von einer Spezifikation in SPECTRUM zu vermitteln, geben wir als Beispiel einen Ausschnitt des abstrakten Datentyps der natürlichen Zahlen *Nat* an.

SPEC Nat = {	Name des abstrakten Datentyps
enriches Bool;	Verwendung des Datentyps Bool
sort Nat;	Sortenvereinbarung
0 : Nat;	Angabe von Funktionen
succ, pred: Nat → Nat;	
iszero: Nat → Bool;	
⋮	
Nat **generated_by** 0, succ;	Termerzeugungsprinzip für Induktion
axioms ∀ *n* : *Nat* **in**	Verhalten der Funktionen, charakterisiert durch Gleichungen
iszero (0) = true;	
iszero (succ (n)) = false;	
pred (0) = ⊥;	
pred (succ (n)) = n;	
⋮	
endaxioms	
}	

Die Funktion *pred* ist eine partielle Funktion; angewandt auf das Argument 0 liefert sie den undefinierten Wert ⊥. Das Termerzeugungsprinzip drückt aus, daß jede natürliche Zahl durch die Konstante 0 und die Funktion *succ* erzeugbar ist. Dadurch läßt sich ein Induktionsprinzip ableiten, um Eigenschaften für den Datentypen zu beweisen.

In SPECTRUM stehen eine Reihe eingebauter Datentypen (z.B. natürliche Zahlen, Listen, Mengen) und Datentypen, die bereits in anderen Anwendungen definiert worden sind, zur Verfügung ([BFG+93b]). Daneben existiert ein Konstruktor *data* für die Definition rekursiver Datentypen.

Durch die Verbindung der formalen Entwurfsmethodik FOCUS mit der algebraischen Spezifikationssprache SPECTRUM steht uns nun ein mächtiges, wohldefiniertes semantisches Modell zur Verfügung, das die Aspekte Verhalten, Struktur und Daten verteilter, reaktiver Systeme umfaßt. Im folgenden Kapitel werden wir in diesem Modell die formale Semantik für SDL definieren.

Kapitel 4

Formale Fundierung von Basic SDL

In diesem Kapitel stellen wir eine denotationelle Semantik für SDL basierend auf stromverarbeitenden Funktionen im Rahmen von FOCUS vor. Unser Vorgehen ist wie folgt charakterisiert: Wir ordnen einer SDL-Spezifikation eine formale Spezifikation in FOCUS zu, wobei das in FOCUS beschriebene System das gleiche Verhalten wie das mit SDL beschriebene System aufweist. Für das FOCUS-System ist die formale Semantik basierend auf stromverarbeitenden Funktionen definiert. Dadurch erhalten wir eine wohldefinierte formale Semantik für die SDL-Beschreibung. Für die Systembeschreibung in FOCUS verwenden wir die logische Kernsprache ANDL, um eine intuitiv verständliche und nachvollziehbare Semantik für SDL zu erhalten. Unsere Semantikdefinition basiert auf den Ansätzen von [Bro91], die eine Überführung von SDL in ein funktionales Modell beschreiben.

Wir definieren die Semantik für eine Teilmenge von SDL, die im wesentlichen mit Basic SDL übereinstimmt. Diese Teilmenge enthält die meistverwendeten Sprachkonstrukte von SDL. Der hierarchische Aufbau eines Systems ist über mehrere Blockebenen möglich, im Gegensatz zu Basic SDL, bei dem nur eine Blockebene erlaubt ist. Auf Prozeßebene berücksichtigen wir folgende SDL-Sprachkonstrukte: Zustände, Ein- und Ausgabesymbole, Tasks und Entscheidungen, Savesymbole, Setzen und Rücksetzen von Timern, Start- und Stopsymbol sowie die Sprachkonstrukte *none* und *any* für die Modellierung nichtdeterministischen Prozeßverhaltens. Zusätzlich werden eingeschränkte Formen von Prozeduren behandelt.

In Hinblick auf die Rolle der Zeit in SDL-Spezifikationen legen wir das von uns in Abschnitt 2.3.4 vorgeschlagene Zeitkonzept zugrunde. Wir gehen von einer globalen, diskreten Systemzeit aus. Die Zeit schreitet ständig fort, der Zeitverbrauch eines SDL-Prozesses ist unbestimmt. Wie in Abschnitt 2.3.4 erläutert, ist die Verwendung von Timern mit quantitativen Ablaufzeitpunkten aufgrund des ungenügenden Zeitkonzepts von SDL nicht sinnvoll. Wir modellieren daher in der Semantikdefinition Timer, die nicht nach einer explizit vorgegebenen Zeitdauer, sondern nach einer unbestimmten, aber endlichen zeitlichen Verzögerung ablaufen.

Für die semantische Behandlung von dynamischen Strukturen, wie sie das Sprachkonstrukt *Create* für die SDL-Prozeßerzeugung modelliert, ist FOCUS in der bisher vorgestellten Form

nicht geeignet, das es für statische Systemstrukturen entwickelt worden ist. Es existiert jedoch eine Erweiterung des semantischen Modells von FOCUS, welche die Behandlung dynamischer Netzwerke ermöglicht. Basierend auf dieser FOCUS-Erweiterung werden wir in Kapitel 5 die Semantikdefinition der dynamischen SDL-Prozeßerzeugung vorstellen.

Wir gliedern unsere Semantikdefinition für SDL gemäß der Aspekte Struktur, Datenanteil und Verhalten. Zunächst beschreiben wir in Abschnitt 4.1 den Aufbau und die Eigenschaften der FOCUS-Systembeschreibung und geben eine formale Spezifikation der Systemstruktur in der logischen Kernsprache ANDL. Anschließend folgt in Abschnitt 4.2 die Definition der formalen Semantik für die SDL-Datentypen. Abschnitt 4.3 umfaßt den dritten und umfangreichsten Teil der Semantikdefinition für SDL – die semantische Fundierung der SDL-Prozesse. Diese schließt auch die Behandlung der SDL-Prozeduren mit ein, die parametrisierte Bereiche von SDL-Prozessen darstellen. Abschnitt 4.4 schließt das Kapitel mit einer Bemerkung zur formalen Fundierung von SDL.

4.1 Formale Fundierung der Systemstruktur von SDL-Spezifikationen

Für ein in SDL spezifiziertes System erstellen wir ein System in FOCUS, welches das gleiche Verhalten wie das SDL-System aufweist. Dazu setzen wir zuerst die Struktur des SDL-Systems nach FOCUS um. Jeder SDL-Komponente wird eine Komponente in FOCUS zugeordnet; die Hierarchieebenen sowie die Verbindungsstruktur aus dem SDL-System werden nach FOCUS übertragen und mit ANDL spezifiziert. Wir werden unser Vorgehen an einem Beispiel verdeutlichen. Abbildung 4.1 zeigt die SDL-Spezifikation eines Systems S, wobei wir auf die Angabe der Signalmengen, die über die Kanäle und Signalwege übertragen werden, der Übersichtlichkeit halber verzichten.

Wir setzen voraus, daß die vorliegenden SDL-Spezifikationen syntaktisch korrekt sind. Für die Signalmengen gelten folgende Bedingungen:

- Die Mengen der Eingabesignale der SDL-Prozesse sind jeweils paarweise disjunkt.
- Die Mengen der Ausgabesignale der SDL-Prozesse sind jeweils paarweise disjunkt.

Der Sender und der Empfänger eines Signals können somit durch die Zugehörigkeit des Signals zu einer bestimmten Signalmenge eindeutig bestimmt werden. Dies führt in einigen Fällen zu einfacheren Spezifikationen (siehe Abschnitt 4.3.1.1). Die obigen Bedingungen können durch Umbenennung der Signale in einer SDL-Spezifikation leicht erfüllt werden.

4.1.1 Spezifikation der Systemebene

Ein mit SDL spezifiziertes System wird in FOCUS durch eine Komponente der Art *System* modelliert. Mit dieser Komponente erhalten wir eine Black-Box-Sicht des Systems. Die Komponente *System* $<$ *systemname* $>$ erhält die entsprechenden Ein- und Ausgabekanäle

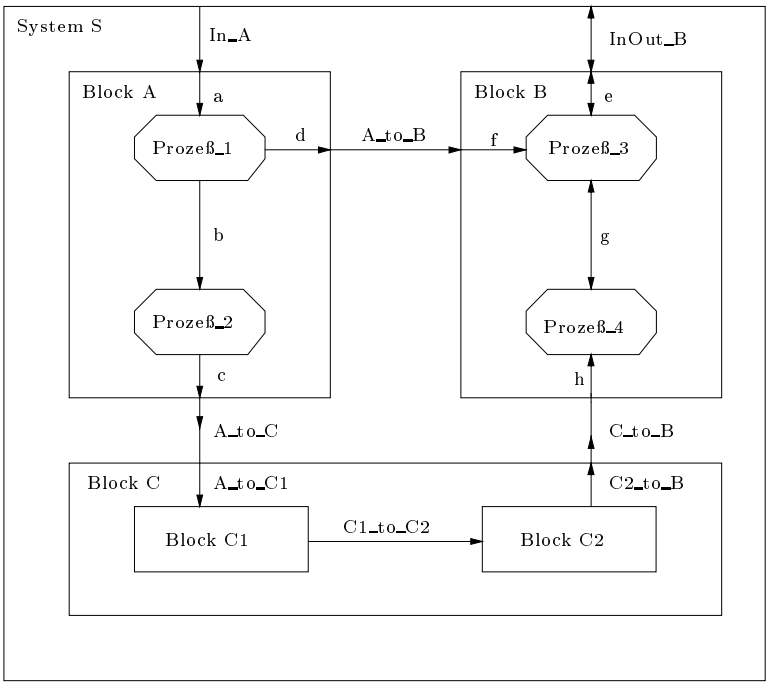


Abbildung 4.1: Beispiel für eine SDL-Spezifikation

aus der zugrundeliegenden SDL-Spezifikation. Es ist zu berücksichtigen, daß Kanäle in FOCUS gerichtet sind. Bidirektionale Kanäle aus der SDL-Spezifikation sind in zwei gerichtete Kanäle in der FOCUS-Systembeschreibung umzusetzen.

Die Schnittstelle der Komponente *System S* zur Systemumgebung besteht somit aus den Kanälen *In_A* und *In_B* für die Eingabe von Signalen aus der Systemumgebung und dem Kanal *Out_B* für die Ausgabe von Signalen (siehe Abb. 4.2).

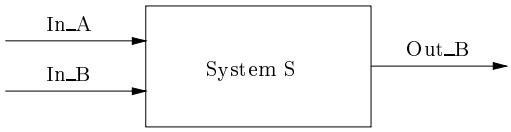


Abbildung 4.2: Die Komponente *System S*

In der FOCUS-Systembeschreibung sind für die Angabe der Kanäle zwischen den Komponenten die Kanalidentifikatoren mit den dazugehörigen Nachrichtentypen erforderlich. Dabei entsprechen die Nachrichtentypen Mengen von Signalen. Die Signalmengen lassen sich anhand der gegebenen SDL-Spezifikation bestimmen. Sei K ein Kanalidentifikator; SIG_K bezeichnet die zu K gehörenden Signale. Ein Signal, das Datenwerte der Sorten $sort_1, \ldots, sort_n$ überträgt, wird in SDL durch eine Deklaration der Art

`Signal <name> (sort`$_1$`, ..., sort`$_n$`)`

vereinbart. In der SDL-Semantikdefinition wird dieses Signal in folgende Nachrichtenmenge N umgesetzt:

$$N = \{(name,\ data_1,\ \ldots,\ data_n) \mid data_i : adt_i \text{ für } 1 \leq i \leq n\}$$

Dabei stellen $adt_1, \ldots, adt_n$ die abstrakten Datentypen aus SPECTRUM dar, auf welche die SDL-Datentypen abgebildet werden. Auf die semantische Fundierung der SDL-Datentypen gehen wir in Abschnitt 4.2 näher ein.

Die Struktur des gegebenen SDL-Systems wird in FOCUS durch ein Netzwerk beschrieben. Damit wird der strukturelle Aufbau der Komponente *System* < *systemname* > festgelegt. *System* wird als Netzwerk aus einer oder mehreren Komponenten der Art *Block* modelliert. Dabei entspricht jedem Block in der SDL-Spezifikation eine Komponente der Art *Block* im Netzwerk *System*. Falls nur verzögerungsfreie Kanäle in der SDL-Spezifikation vorliegen, können die Kommunikationsverbindungen zwischen den Komponenten der Art *Block* und der Systemumgebung direkt aus der SDL-Spezifikation übernommen werden, wobei bidirektionale Verbindungen wieder in unidirektionale Verbindungen zu überführen sind. In Hinblick auf Kanäle mit Verzögerung ist zu berücksichtigen, daß die Übertragung von Nachrichten über Kanäle in FOCUS ohne Zeitverbrauch erfolgt. Der Zeitverbrauch bei der Signalübertragung in SDL ist deshalb in FOCUS explizit zu modellieren. Wir führen für SDL-Kanäle mit Verzögerung Komponenten der Art *KV* ein, die die Verzögerung bei der Signalübertragung umsetzen. Signale, die über einen Kanal mit Verzögerung gesendet werden, werden zuerst an die für diesen Kanal eingeführte Komponente *KV* geschickt und erreichen anschließend den Empfängerblock bzw. die Systemumgebung.
Es ist zu beachten, daß mit SDL 92 auch verzögerungsfreie Kanäle als Sprachmittel zur Verfügung stehen. In vielen SDL-Spezifikationen werden deshalb keine Kanäle mit Verzögerung mehr verwendet, so daß sich die Einführung von Komponenten der Art *KV* erübrigt.

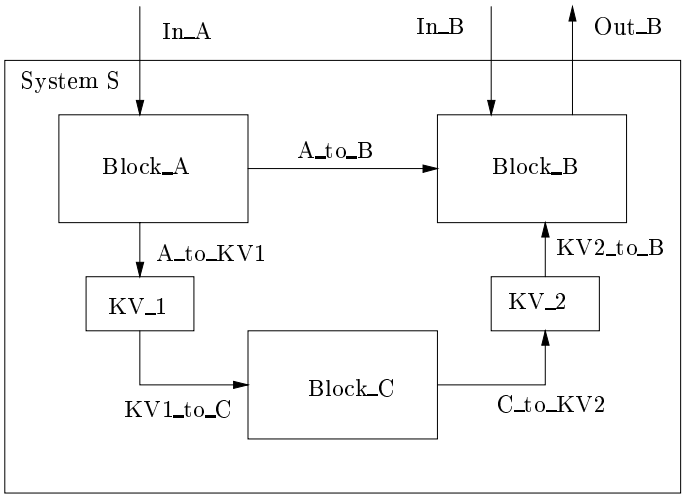


Abbildung 4.3: Struktureller Aufbau der Komponente *System S*

In unserem Beispiel *System S* sind die Kanäle, die *Block_A* bzw. *Block_B* mit *Block_C* verbinden, verzögernd, so daß insgesamt zwei Kanalverzögerungskomponenten einzuführen sind. Abbildung 4.3 zeigt den Aufbau von *System S*, das aus drei Komponenten der Art *Block* und zwei Kanalverzögerungskomponenten besteht. Die formale Spezifikation der Komponente *System S* als Netzwerk in ANDL lautet:

agent *System S*

input channel $In\_A : SIG_{In\_A},\ In\_B : SIG_{In\_B}$
output channel $Out\_B : SIG_{Out\_B}$

is network

$$\begin{aligned} < A\_to\_KV1,\ A\_to\_B > &= Block\_A < In\_A >; \\ < Out\_B > &= Block\_B < In\_B,\ A\_to\_B,\ KV2\_to\_B >; \\ < C\_to\_KV2 > &= Block\_C < KV1\_to\_C >; \\ < KV1\_to\_C > &= KV1 < A\_to\_KV1 >; \\ < KV2\_to\_B > &= KV2 < C\_to\_KV2 > \end{aligned}$$

end *System S*

4.1.2 Spezifikation von Kanälen mit Verzögerung

Die Kanalverzögerungskomponenten der Art KV modellieren die beliebig, aber endlich lange Verzögerung, die ein Signal während der Übertragung durch einen SDL-Kanal mit Verzögerung erfährt. Jedes Eingabesignal wird von der Kanalverzögerungskomponente KV um beliebig viele Zeitintervalle verzögert. Dabei bleibt die Reihenfolge der Eingabesignale erhalten. Abstrahieren wir in den Ein- und Ausgabeströmen von der zeitlichen Information, so folgt, daß Ein- und Ausgabestrom von KV identisch sind. Somit erhalten wir mit dem Spezifikationsformat für Basiskomponenten mit zeitunabhängigem Verhalten folgende einfache Spezifikation einer Kanalverzögerungskomponente; sei dabei SIG die Menge der Eingabe- bzw. Ausgabesignale der Komponente.

agent KV – **time independent**

input channel $in : SIG$
output channel $out : SIG$

is basic

$$in = out$$

end KV

In der Semantikdefinition wird jedem Kanal mit Verzögerung eine Komponente der Art KV zugeordnet. Dabei werden die Signalmenge SIG und die Kanalbezeichner in und out durch die aktuellen Bezeichner ersetzt. Diese ergeben sich aus der SDL-Spezifikation und der Komponente *System S*.

Anmerkung:

Diese Spezifikation von Kanalverzögerungskomponenten entspricht einer Spezifikation mit parametrisierter Schnittstelle, die jedoch in ANDL nicht unterstützt wird. Wir sprechen deshalb in Zukunft von Komponenten „der Art S“ und gehen davon aus, daß die Schnittstelle der Komponente S dabei aus den aktuellen Ein- und Ausgabekanäle gebildet wird und die Komponente einen eindeutigen Bezeichner erhält.

4.1.3 Spezifikation der Blockebene

Eine Komponente der Art *Block* wird wieder als Netzwerk beschrieben. Dabei ist zu unterscheiden, ob der entsprechende Block in der SDL-Spezifikation in Blöcke oder in Prozesse unterteilt ist.

Für den Fall, daß der SDL-Block in weitere Blöcke unterteilt ist, gehen wir wie bei der Umsetzung der Systemebene vor. Für jeden SDL-Block wird eine Komponente der Art *Block* in FOCUS eingeführt, für jeden Kanal mit Verzögerung eine Kanalverzögerungskomponente. Die Bezeichner für die Kanäle können sich an der Blockgrenze des übergeordneten Blocks ändern. Wir behalten für die Semantikdefinition die Kanalnamen aus der Blockumgebung bei. Für *Block_C* ergibt sich gemäß Abbildung 4.1 das in Abbildung 4.4 dargestellte Netzwerk, das durch folgende ANDL-Netzwerkkomponente beschrieben wird.

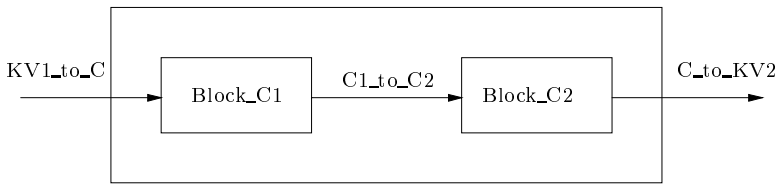


Abbildung 4.4: Struktureller Aufbau der Komponente *Block_C*

agent *Block_C*

input channel $KV1\_to\_C : SIG_{KV1\_to\_C}$

output channel $C\_to\_KV2 : SIG_{C\_to\_KV2}$

is network

$< C1\_to\_C2 > \;=\; Block\_C1 < KV1\_to\_C >;$
$< C\_to\_KV2 > \;=\; Block\_C2 < C1\_to\_C2 >$

end *Block_C*

Besteht ein SDL-Block aus mehreren SDL-Prozessen, so wird für jeden SDL-Prozeß eine Komponente der Art *Prozeß* eingeführt. Abbildung 4.5 zeigt die Umsetzung des SDL-Blocks *Block_A* aus Abbildung 4.1 nach FOCUS.

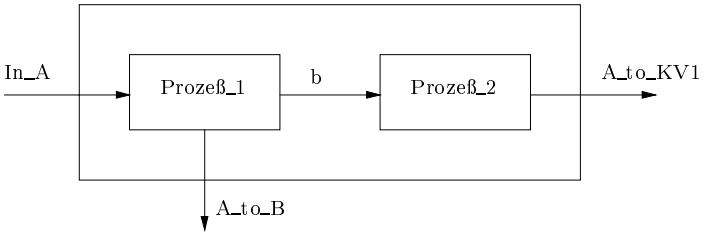


Abbildung 4.5: Struktureller Aufbau der Komponente *Block_A*

In SDL gehen an der Blockgrenze die Kanäle in Signalwege über, so daß sich die Bezeichnung der Verbindung ändern kann. In der Semantikdefinition behalten wir die Kanalnamen bei und verwenden nur für die Verbindungen zwischen den Prozessen die Namen der Signalwege aus der gegebenen SDL-Spezifikation.

In ANDL lautet die Spezifikation der Komponente *Block_A* wie folgt:

agent *Block_A*

input channel $In\_A : SIG_{In\_A}$

output channel $A\_to\_KV1 : SIG_{A\_to\_KV1},\ A\_to\_B : SIG_{A\_to\_B}$

is network

$< b,\ A\_to\_B > \ = \ Prozeß\_1 < In\_A >;$
$< A\_to\_KV1 > \ = \ Prozeß\_2 < b >$

end *Block_A*

Das Verhalten einer Komponente der Art *Prozeß* leitet sich aus dem dazugehörigen SDL-Prozeß ab und wird in Abschnitt 4.3 definiert.

4.1.4 Spezielle Kommunikationsstrukturen

Die bisher vorgestellte Umsetzung des Strukturanteils von SDL-Spezifikationen nach FOCUS ist in Bezug auf die Kommunikationsstruktur für zwei Fälle erweitern. Da in FOCUS über die gerichteten Kanäle Punkt-zu-Punkt-Verbindungen zwischen den Komponenten geschaffen werden, können Mehrpunkt-zu-Punkt- bzw. Punkt-zu-Mehrpunkt-Verbindungen aus SDL nicht direkt umgesetzt werden. Hierfür sind in der Semantikdefinition Misch- und Verteilkomponenten einzufügen. Im folgenden stellen wir diese Umsetzungen vor.

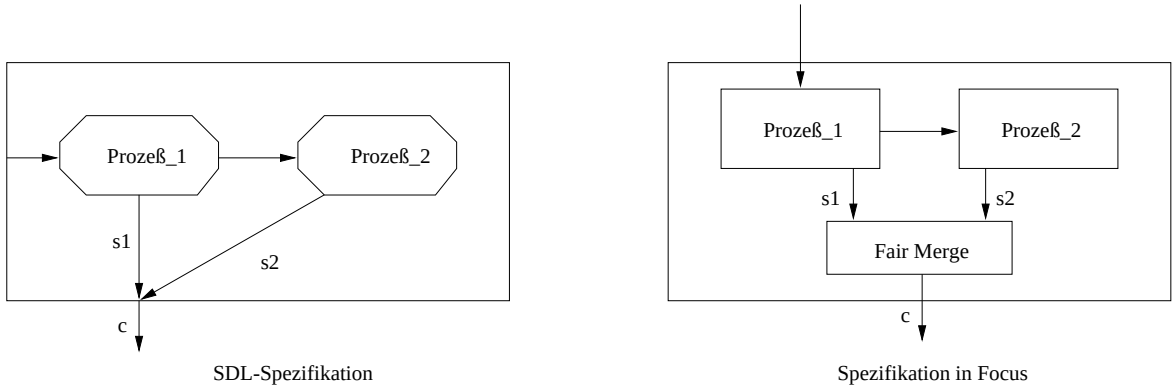


Abbildung 4.6: Zusammenführung mehrerer Signalwege in einen Kanal

- Mehrere Kanäle (bzw. Signalwege) werden zu einem Kanal zusammengeführt. In der FOCUS-Systembeschreibung wird diese Situation durch das Einfügen einer Mischkomponente *Fair Merge* explizit modelliert (siehe Abbildung 4.6). Diese fügt meh-

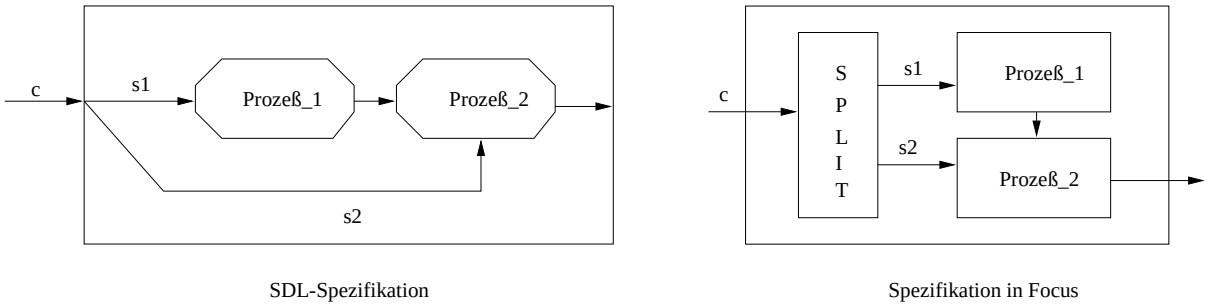


Abbildung 4.7: Verzweigung eines Kanals in mehrere Signalwege

rere Nachrichtenströme zu einem einzigen Strom zusammen. Die Spezifikation der Mischkomponente erfolgt in Abschnitt 4.3.1.1.

- Ein Kanal verzweigt sich in mehrere Kanäle (bzw. Signalwege). In der FOCUS-Systembeschreibung wird diese Situation durch das Einfügen einer Verteilkomponente *Split* explizit modelliert (siehe Abbildung 4.7). Diese teilt die Signale eines Kanals auf mehrere Kanäle auf. Die Spezifikation der Verteilkomponente wird in Abschnitt 4.3.1.3 vorgestellt.

4.1.5 Schematisches Erstellen der Focus-Systembeschreibung

Aus einer gegebenen SDL-Spezifikation läßt sich die Systembeschreibung des SDL-Systems, also die Struktur der Komponente *System*, schematisch ableiten:

- Für jeden SDL-Block wird eine Komponente der Art *Block* definiert.
- Ein SDL-Kanal wird durch einen FOCUS-Kanal modelliert, wobei für jeden bidirektionalen SDL-Kanal zwei unidirektionale FOCUS-Kanäle eingeführt werden.
- SDL-Kanäle mit Verzögerung werden durch Komponenten der Art KV modelliert, die zwischen dem sendenden und dem empfangenden Block bzw. der Systemumgebung liegen.
- Die Komponenten der Art *Block* bilden die oberste Strukturierungsebene der Komponente *System*.
- Eine Komponente *Block* wird entweder in weitere Komponenten der Art *Block* oder in Komponenten der Art *Prozeß* unterteilt.
- Für die Prozesse, aus denen sich ein SDL-Block zusammensetzt, werden in der entsprechenden Komponente *Block* Komponenten der Art *Prozeß* eingeführt.

Die Anzahl der vorhandenen Komponenten *Block* und *Prozeß* entspricht der Anzahl der entsprechenden SDL-Komponenten in der gegebenen SDL-Spezifikation. Falls jedoch ein SDL-Block nur einen SDL-Prozeß enthält, so kann die Einführung der Komponente *Block* entfallen, da durch den Block keine sinnvolle Hierarchisierung gegeben wird. In diesem Fall

stimmt die Zahl der SDL-Blöcke nicht mit der Anzahl der Komponenten *Block* überein (siehe auch kritische Anmerkung auf Seite 33).

Um den Zusammenhang zwischen einer SDL-Spezifikation und der zugehörigen Systembeschreibung in FOCUS zu verdeutlichen, erhalten die Block- und Prozeßkomponenten die entsprechenden Namen aus der SDL-Spezifikation. Die Vergabe der Kanalnamen orientiert sich an den Kanalnamen der SDL-Spezifikation, wobei eine direkte Übernahme durch die Einführung von unidirektionalen Kanälen und von Kanalverzögerungskomponenten nicht immer möglich ist.

4.2 Formale Fundierung der SDL-Datentypen

Im folgenden beschreiben wir, wie sich der Datenanteil von SDL nach SPECTRUM umsetzen läßt. Der SDL-Datenanteil umfaßt die Definition abstrakter Datentypen, die Deklaration von lokalen Variablen in den SDL-Prozessen und den Zugriff auf die Variablen während des Prozeßablaufs. Wir zeigen, wie sich die in SDL vorgegebenen Standarddatentypen in SPECTRUM definieren lassen, und stellen einen Datentypen für die formale Fundierung der lokalen Datenanteile in den Prozessen vor. Die formale Umsetzung des Zugriffs auf die Variablen während des Prozeßablaufs wird in Abschnitt 4.3.2 behandelt.

Wie wir in Abschnitt 2.4 erläutert haben, ist das Datentypkonzept von SDL unzureichend, da es keine adäquate Behandlung für nichtterminierende Berechnungen bietet. Ein wichtiges Ziel der semantischen Umsetzung des Datenanteils in SDL-Spezifikationen besteht darin, Fragen nach der Terminierung von Anweisungen zu beantworten. Dazu ist eine logische und mathematische Basis notwendig, in der sich dieser Aspekt explizit behandeln läßt. Deshalb verwenden wir in der Semantikdefinition für den Datenanteil das Konzept der abstrakten Datentypen, wie es der algebraischen Spezifikationssprache SPECTRUM zugrunde liegt (siehe Abschnitt 3.3). Die in SDL-Spezifikationen vereinbarten Datentypen bilden wir auf entsprechende Datentypen in SPECTRUM ab.

4.2.1 Umsetzung von SDL-Datentypen nach Spectrum

In SPECTRUM existiert eine Reihe bereits vorhandener Datentypen (z.B. in [BFG+93b] oder [Reg95b]), auf die sich die Standarddatentypen von SDL abbilden lassen. Dabei werden nichtterminierende Berechnungen und Funktionsaufrufe, denen kein sinnvolles Ergebnis zugeordnet werden kann, mit dem Symbol $\perp$ aus SPECTRUM modelliert, das für die Behandlung partieller Funktionen eingeführt wurde. Somit ist die unzulängliche Modellierung mit dem Ausdruck *error* (vergleiche Abschnitt 2.4) in den SDL-Datentypen aufgehoben; die Terminierung von Ausdrücken in SDL-Prozeßsymbolen kann in der formalen SDL-Semantik überprüft werden (siehe Abschnitt 4.3.2.2). Damit eine möglichst direkte Umsetzung von SDL nach SPECTRUM erfolgen kann, sind einige syntaktische Änderungen an den Bezeichnungen und Funktionsnamen bestehender SPECTRUM-Datentypen vorzunehmen. Dies stellt jedoch keine Schwierigkeit dar, so daß wir darauf nicht näher eingehen werden.

In SDL besteht die Möglichkeit, in der Definition von Datentypen Funktionen graphisch zu spezifizieren. Dazu sind in SDL 92 *Operator Diagrams* eingeführt worden. Die semantische Umsetzung von derartigen Funktionsspezifikationen erfolgt analog zur Umsetzung von SDL-Prozeduren und wird in Abschnitt 4.3.2.7 vorgestellt.

Die nachfolgende Tabelle zeigt die Zuordnung von Datentypen in SDL zu Datentypen in SPECTRUM:

SDL	SPECTRUM
Boolean	Bool
Character	Char
Charstring	String
Natural	Nat
Integer	Int
Array (Generator)	Grex
Powerset (Generator)	Set
Duration	Nat
Time	Nat
Pid	Nat

Tabelle 4.1: Zuordung von SDL- zu SPECTRUM-Datentypen

Da wir für unsere Semantikdefinition ein diskretes Zeitmodell voraussetzen, eignet sich für die Modellierung der SDL-Datentypen *Time* und *Duration*, die in SDL als Teilmenge der rationalen Zahlen definiert sind, der SPECTRUM-Datentyp der natürlichen Zahlen. Für die Modellierung der eindeutigen SDL-Prozeßnummern verwenden wir ebenfalls den Datentypen der natürlichen Zahlen. Für den SDL-Generator *String* und den Datentyp der reellen Zahlen liegen noch keine SPECTRUM-Datentypen vor. Sie sind bei Bedarf zu definieren. Dabei ist die Bezeichnung des Datentyps *Real* in SDL irreführend. Der Datentyp definiert nicht die reellen, sondern die rationalen Zahlen.

Für die Umsetzung von Verbundtypen, die mittels des SDL-Konstruktors *struct* definiert werden, verwenden wir das Sprachkonstrukt *data* für Datentypdeklarationen aus SPECTRUM. Die Definition eines Verbundtyps in SDL lautet:

```
newtype s struct
     sel_1 sort_1
          ⋮
     sel_n sort_n
endnewtype
```

Auf diesem ADT stehen in SDL implizit Konstruktor- und Selektorfunktionen sowie Funktionen für selektives Ändern der einzelnen Komponenten zur Verfügung.

Der oben definierte Datentyp wird auf folgenden abstrakten Datentypen in SPECTRUM abgebildet:

$$data\ s = make\ (sel_1 : sort_1,\ \ldots,\ sel_n : sort_n)$$

Der Sortenkonstruktor s verfügt über die Konstruktorfunktion $make : sort_1 \times ... \times sort_n \rightarrow s$ und die Selektorfunktionen $sel_1, \ldots, sel_n$, über die auf die einzelnen Elemente des Verbundtyps s zugegriffen werden kann.

Für die Umbenennung von Datentypen, wie sie in SDL durch das Konstrukt *syntype* möglich ist, steht in SPECTRUM das Sprachkonstrukt *sortsyn* zur Verfügung.

4.2.2 Definition des Datenzustands eines SDL-Prozesses

Um den lokalen Datenanteil eines SDL-Prozesses formal zu fundieren, werden zunächst die Sorten der innerhalb des SDL-Prozesses deklarierten Variablen auf SPECTRUM-Datentypen abgebildet. Anschließend wird der lokale Datenanteil, der sich aus der Belegung aller lokalen Variablen zusammensetzt, als SPECTRUM-Datentyp modelliert.

Seien $v_1, \ldots, v_n$ die lokalen Variablen des SDL-Prozesses mit den Sorten $sort_1, \ldots, sort_n$. Seien $adt_1, \ldots, adt_n$ die zugehörigen Datentypen aus SPECTRUM. Der Datentyp D für den prozeßlokalen Datenzustand wird mittels des Sprachkonstrukts *data* aus SPECTRUM definiert:

$$data \;\; D = md \;\; (!\, v_1 : adt_1, \;\; \ldots, \;\; !\, v_n : adt_n).$$

Damit stehen die Konstruktorfunktion $md : adt_1, \ldots, adt_n \rightarrow D$ und Selektorfunktionen für jede Sorte adt_i zur Verfügung: $v_i : D \rightarrow adt_i$. Die Selektorfunktionen und die Konstruktorfunktion sind strikt[1].

Für jede Sorte adt_i definieren wir eine Updatefunktion up_i auf D:

$$\begin{aligned} up_1 &: D \times adt_1 \rightarrow D \\ &\vdots \\ up_n &: D \times adt_n \rightarrow D \end{aligned}$$

Statt der Bezeichnungen up_i verwenden wir eine Notation, bei der wir den Variablennamen v_i miteinbeziehen. Seien $\sigma : D$ und $d : adt_i$ gegeben. $\sigma\ [v_i := d]$ bezeichnet den Datenzustand, der sich ergibt, wenn der Wert der Variablen v_i durch d ersetzt wird. ($\sigma\ [v_i := d]$ ist gleichwertig zu $up_i\ (\sigma,\ d)$). $v_i(\sigma)$ liefert den Wert der Variablen v_i.

Es gilt weiterhin:

$$v_j\ (\sigma\ [v_i := d]) = \begin{cases} d & \text{falls } i = j \\ v_j(\sigma) & \text{sonst} \end{cases}$$

Mit dem Datentypen D für die lokal deklarierten Variablen eines SDL-Prozesses stehen nun Funktionen für den Zugriff auf die Werte der einzelnen Variablen zur Verfügung.

[1] In der Datentypdeklaration mit *data* werden strikte Funktionen mit einem Ausrufezeichen „!“ gekennzeichnet. Eine strikte Funktion liefert den undefinierten Wert $\perp$, wenn mindestens eines ihrer Argumente undefiniert ist.

Diese Funktionen sind erforderlich, um den datenabhängigen Verhaltensanteil eines SDL-Prozesses, der in den Task- und Entscheidungssymbolen spezifiziert ist, formal zu fundieren (siehe Abschnitt 4.3.2).

4.3 Formale Fundierung von SDL-Prozessen

Die Definition der formalen Semantik für SDL-Prozesse stellt den umfangreichsten Teil der gesamten Semantikdefinition von SDL in FOCUS dar. Das SDL-Prozeßkonzept enthält neben dem graphischen Anteil des Prozeßgraphen einige implizite Konzepte, die in der formalen Semantik zu berücksichtigen sind. Dazu zählen der Eingabepuffer und die unbestimmte Verzögerung von Timersignalen. Um eine strukturierte Semantikdefinition und dadurch eine schematische Umsetzung eines SDL-Prozesses nach FOCUS zu erhalten, modellieren wir einen SDL-Prozeß als ein Netzwerk von Komponenten in FOCUS. Jede Komponente setzt dabei einen Verhaltensanteil des SDL-Prozesses um. Die Komponenten für den Eingabepuffer und die Verzögerung von Timersignalen können dabei wie Bausteine bei jeder SDL-Prozeßumsetzung verwendet werden. Die Komponente, die dem SDL-Prozeßgraphen entspricht und die Umsetzung der Zustandsübergänge enthält, ist dagegen individuell für jeden SDL-Prozeß anzugeben.

Zunächst beschreiben wir in Abschnitt 4.3.1 die Umsetzung eines SDL-Prozesses in ein FOCUS-Netzwerk und geben die Spezifikation der einzelnen Komponenten an. Anschließend folgt in Abschnitt 4.3.2 die Umsetzung der Zustandsübergänge des SDL-Prozeßgraphen in Funktionsgleichungen, wobei wir die verschiedenen Prozeßsprachkonstrukte behandeln. Dies schließt auch die formale Fundierung von SDL-Prozeduren in Abschnitt 4.3.2.7 mit ein, die innerhalb einer SDL-Prozeßspezifikation definiert werden.

4.3.1 Umsetzung eines SDL-Prozesses in ein Focus-Netzwerk

Um das Verhalten eines SDL-Prozesses formal zu fundieren, genügt es nicht, allein die graphische Darstellung des erweiterten Zustandsautomaten zu betrachten. Auch die Konzepte, die nicht explizit in einer SDL-Prozeßspezifikation dargestellt sind, sind maßgeblich am Verhalten eines SDL-Prozesses beteiligt. Dazu zählen der unbeschränkte Eingabepuffer des Prozesses, in den die ankommenden Signale eingereiht werden, und die Verzögerung von Timersignalen.

Unser Ziel ist es, das komplexe Verhalten eines SDL-Prozesses auf in sich abgeschlossene Verhaltensanteile zu verteilen. Solche Verhaltensanteile sind etwa der Eingabepuffer eines Prozesses oder die Verzögerung von Timersignalen. Wir modellieren die Komponente *Prozeß*, die wir in der Strukturbeschreibung einem SDL-Prozeß zugeordnet haben, deshalb nicht als eine Basiskomponente, sondern als ein Netzwerk aus mehreren Komponenten. Jede dieser Komponenten modelliert einen Verhaltensanteil des SDL-Prozesses und dient als Baustein für die Semantikdefinition der SDL-Prozesse. Die Funktionalität der einzelnen Komponenten wird klar und einfach gehalten.

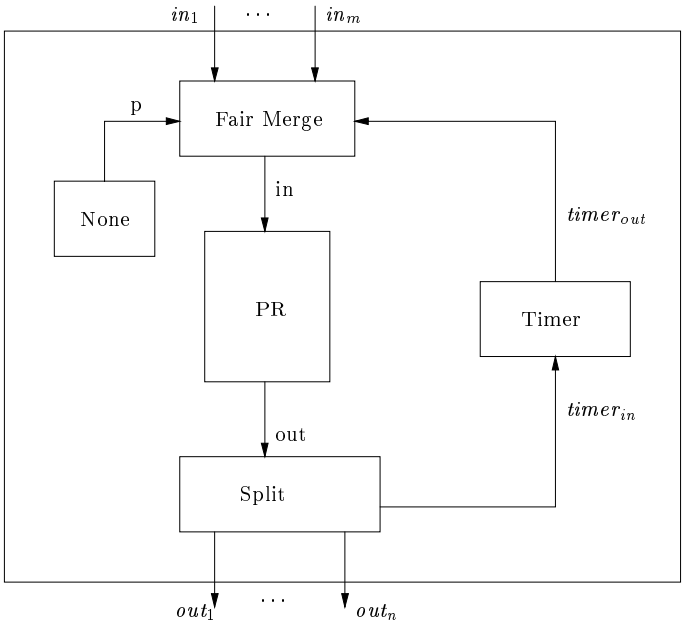


Abbildung 4.8: Struktureller Aufbau einer Komponente der Art *Prozeß*

Betrachten wir das durch einen SDL-Prozeß spezifizierte Verhalten:
Ein SDL-Prozeß erhält über verschiedene Signalwege Signale, die in seinen Eingabepuffer eingereiht werden. Aus diesem werden die Signale sequentiell ausgelesen und initiieren Zustandsübergänge, die in der graphischen Prozeßbeschreibung angegeben sind. Während eines Zustandsübergangs werden Signale erzeugt, die über verschiedene Signalwege an die Umgebung gesendet werden. Zusätzlich kann der Prozeß einen Timer setzen; die dazugehörigen Timersignale werden nach einer beliebig langen, aber endlichen Verzögerung in den Eingabepuffer eingereiht.

Ausgehend von dieser Verhaltensbeschreibung gelangen wir zu der in Abbildung 4.8 dargestellten Aufteilung der Komponente *Prozeß* in Basiskomponenten. Die einzelnen Teilkomponenten der Komponente *Prozeß* übernehmen folgende Aufgaben:

- Die Mischkomponente **Fair Merge** modelliert den Eingabepuffer des SDL-Prozesses.
- **PR** modelliert das Ein-/Ausgabeverhalten des SDL-Prozesses, das sich aus dem Zustandsautomaten des SDL-Prozesses ableitet.
- Die Verteilkomponente **Split** verteilt die Ausgabesignale auf die verschiedenen Ausgabekanäle, die sich aus den Signalwegen des SDL-Prozesses und der Verbindung zur Komponente **Timer** zusammensetzen.
- **Timer** setzt einen Teil des Timerkonzepts von SDL-Prozessen um. Das Setzen von Timern wird in **PR** durch die Ausgabe von Timernachrichten modelliert. Diese werden an die Komponente **Timer** gesendet und dort unbestimmt, aber endlich lange verzögert, bevor sie an den Eingabepuffer des SDL-Prozesses weitergesendet werden.

- **None** erzeugt einen Strom von *none*-Nachrichten, die für die Modellierung spontaner Übergänge im SDL-Prozeß erforderlich sind.

Es ergibt sich folgende Netzwerkbeschreibung in ANDL:

agent *Prozeß*

input channel $in_1 : In_1,\ \ldots,\ in_m : In_m$
output channel $out_1 : Out_1,\ \ldots,\ out_n : Out_n$

is network

$$\begin{aligned} < in > &= FairMerge < in_1, \ldots, in_m, timer_{out}, p >; \\ < out > &= PR < in >; \\ < timer_{out} > &= Timer < timer_{in} >; \\ < p > &= None < \ >; \\ < out_1, \ldots, out_n, timer_{in} > &= Split < out > \end{aligned}$$

end *Prozeß*

Die Kanäle $in_1, \ldots, in_m$ und $out_1, \ldots, out_n$ ergeben sich aus der Umsetzung des SDL-Blocks, in dem der betrachtete SDL-Prozeß unmittelbar enthalten ist. Sie modellieren die Signalwege, über die der SDL-Prozeß mit seiner Umgebung kommuniziert.

Die Komponenten *FairMerge*, *Split*, *Timer*, *None* und *PR* sind Basiskomponenten, d.h. sie werden selbst nicht weiter in Netzwerke strukturiert. Die Komponenten *Timer* und *None* sind nur dann Bestandteil der Komponente *Prozeß*, wenn der zu spezifizierende SDL-Prozeß Timer bzw. spontane Übergänge enthält. Die Misch- und Verteilkomponenten sind nur notwendig, wenn die Schnittstelle der Komponente *PR* über mehr als einen Eingabekanal bzw. mehr als einen Ausgabekanal verfügt.

Die Komponente *PR* beschreibt den Zustandsautomaten, der durch den SDL-Prozeßgraphen angegeben wird. Sie erhält einen Strom von Eingabenachrichten, verarbeitet diesen und liefert einen Strom von Ausgabenachrichten. Das Verhalten von *PR* wird durch eine Menge von Funktionsgleichungen spezifiziert, die sich aus den SDL-Zustandsübergängen des zugrundeliegenden SDL-Prozesses ableiten lassen. Die Ableitung der Funktionsgleichungen aus dem SDL-Prozeßgraphen stellen wir in Abschnitt 4.3.2 vor. Im folgenden beschreiben wir das Verhalten und das Zusammenspiel der einzelnen Basiskomponenten und geben die dazugehörigen Spezifikationen in ANDL an.

4.3.1.1 Modellierung des Eingabepuffers

Jeder SDL-Prozeß besitzt einen unbeschränkten Eingabepuffer, in den alle Signale, die den Prozeß erreichen, der Reihe nach eingetragen werden. Signale, die gleichzeitig eintreffen, werden gemäß des Interleavingprinzips in beliebiger Reihenfolge eingetragen (siehe Seite 14). Die Reihenfolge der Signale bezüglich eines Eingabekanals bleibt dabei jedoch unverändert. Der Prozeß arbeitet den Eingabepuffer als FIFO-Warteschlange ab, sofern diese Reihenfolge nicht durch die Verwendung von Savesignalen aufgehoben wird.

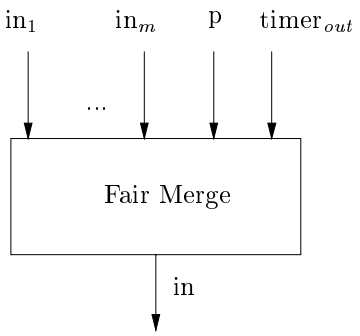


Abbildung 4.9: Modell des Eingabepuffers

In der FOCUS-Systembeschreibung wird die Aufgabe des Eingabepuffers durch eine Mischkomponente *FairMerge* realisiert. Diese Komponente mischt die Nachrichten auf den Eingabekanälen in_1 bis in_m, die Timernachrichten von Kanal $timer_{out}$ und die *none*-Nachrichten von Kanal p (siehe Abschnitt 4.3.1.5) zu einem Nachrichtenstrom zusammen. Das Mischen der Nachrichten wird anhand des Zeitticks $\surd$ synchronisiert: alle Nachrichten, die innerhalb eines bestimmten Zeitintervalls auf den Eingabekanälen eintreffen, werden gemischt und auf den Ausgabekanal geschrieben. Bei *FairMerge* handelt es sich somit um eine Basiskomponente mit zeitabhängigem Verhalten. Das Mischen selbst erfolgt allerdings ohne zeitliche Verzögerung; die Verzögerung, die beim Verarbeiten eines Eingabesignals im SDL-Prozeß auftritt, findet ausschließlich in der Komponente *PR* statt.

Betrachten wir die Beziehung zwischen einem Eingabestrom in_i und dem Ausgabestrom *out* der Mischkomponente: Der Strom, den wir erhalten, wenn alle Nachrichten, die in in_i vorkommen, und die Zeitinformation $\surd$ aus dem Ausgabestrom gefiltert werden, ist gleich dem ursprünglichen Eingabestrom. Voraussetzung ist dabei, daß die Nachrichtenmengen der Eingabekanäle paarweise disjunkt sind.

Seien $i_1, \ldots, i_k$ mit den Nachrichtenmengen $I_1, \ldots, I_k$ die Eingabekanäle der Mischkomponente, wobei gilt: $\forall\, a, b \in \{1, \ldots, k\},\ a \neq b :\ I_a \cap I_b = \varnothing$. Das Mischen der k Ströme zu einem Strom o ist wie folgt spezifiziert:

agent *Fair Merge* – **time dependent** – **wp**

input channel $i_1 : I_1,\ \ldots,\ i_k : I_k$
output channel $o : O$

is basic

$$\begin{aligned} i_1 &= (I_1 \cup \{\surd\}) \copyright\, o \wedge \\ &\vdots \qquad\quad \vdots \\ i_k &= (I_k \cup \{\surd\}) \copyright\, o \end{aligned}$$

end *Fair Merge*

Die Ein- und Ausgabekanäle mit den zugehörigen Nachrichtenmengen von *FairMerge* sind jeweils an die konkret vorliegende Schnittstelle der Komponente *Prozeß* anzupassen. Die

Kanäle $timer_{out}$ und p gehören nur zu den Eingabekanälen von *FairMerge*, wenn der gegebene SDL-Prozeß über Timer bzw. spontane Übergänge verfügt.

Für die Spezifikation der Misch- und Verteilkomponenten (siehe Abschnitt 4.3.1.3) ist es wesentlich, daß die Menge der Ein- und Ausgabesignale der einzelnen SDL-Prozesse jeweils paarweise disjunkt sind. Andernfalls lassen sich die Nachrichten des Ausgabestroms nicht eindeutig den Eingabeströmen zuordnen, so daß komplexere Spezifikationen erforderlich sind.

Anmerkung:

Das Verhalten der Komponente wird sehr abstrakt beschrieben und ist nichtdeterministisch. Es wird kein Mischverfahren für die Eingabeströme angegeben. Es wird offen gelassen, in welcher Reihenfolge die Nachrichten, die während eines Zeitintervalls eintreffen, gemischt werden. Es gilt nur, daß die Reihenfolge von Nachrichten eines Eingabekanals erhalten bleibt. Im semantischen Modell von ANDL wird durch diese Spezifikation somit eine Menge von pulsgetriebenen Funktionen bestimmt; jede Funktion beschreibt die angegebene Ein-/Ausgabebeziehung unter Verwendung eines beliebigen Mischverfahrens.

4.3.1.2 Formale Fundierung des SDL-Prozeßgraphen

Das Verhalten von *PR* leitet sich aus dem Zustandsautomaten des SDL-Prozesses ab. Die Komponente *PR* erhält von der Komponente *FairMerge* einen Strom von Eingabenachrichten *in* und erzeugt einen Strom von Ausgabenachrichten *out*. Die Relation zwischen Ein- und Ausgabestrom ergibt sich aus dem erweiterten Zustandsautomaten, der durch den SDL-Prozeßgraphen beschrieben ist.

Wie in Abschnitt 2.3.4 erläutert, ist der Zeitverbrauch eines SDL-Prozesses für Wartezeiten in Zuständen und das Ausführen von Zustandsübergängen beliebig und nimmt bei jeder Ausführung andere Werte an. Betrachten wir die Komponente *PR* als Black Box (siehe Abbildung 4.10), also ihr von außerhalb beobachtbares Ein-/Ausgabeverhalten, so ist während der Eingabe einer Nachricht a und der Ausgabe der dadurch initiierten Nachrichten o_1 und o_2 eine beliebige Verzögerung zu beobachten. An welchen Stellen der Berechnung in der Komponente *PR* die Zeit fortschreitet, kann von außerhalb der Komponente nicht festgestellt werden. Wir gehen davon aus, daß die Zeit sowohl in den Zuständen als auch in den Zustandsübergängen vergeht.

In der formalen Semantik für SDL-Prozesse modellieren wir das zeitliche Verhalten wie folgt: Da das zeitliche Verhalten eines SDL-Prozesses völlig beliebig ist, abstrahieren wir auf der Spezifikationsebene von zeitlichen Betrachtungen und wählen das Spezifikationsformat für zeitunabhängige Basiskomponenten. Wie in Abschnitt 3.2.2 definiert, wird bei zeitunabhängig spezifizierten Basiskomponenten der zeitliche Aspekt auf der semantischen Ebene eingebracht. Jede pulsgetriebene Funktion, die die Spezifikation des SDL-Prozesses erfüllt, stellt ein mögliches zeitliches Verhalten des SDL-Prozesses dar. Dabei legt jede Funktion ein anderes Verzögerungsverhalten des Prozesses fest, so daß wir durch die

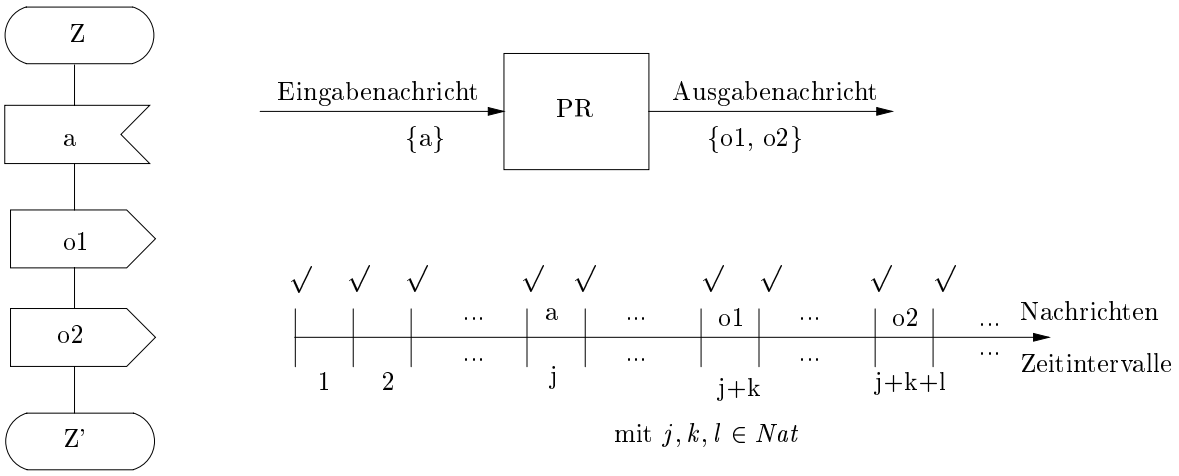


Abbildung 4.10: Zeitliches Verhalten der Komponente *PR*

zeitunabhängige Spezifikationsform jedes mögliche zeitliche Verhalten des SDL-Prozesses erfassen, das die festgelegte Beziehung zwischen Ein- und Ausgabesignalen erfüllt.

Die Spezifikation der Schnittstelle einer Komponente *PR* lautet:

agent *PR* – **time independent**

input channel $in : In$
output channel $out : Out$

is basic

$$Q\ (in,\ out)$$

end *PR*

Mittels des Prädikats $Q\ (in,\ out)$ wird die Beziehung zwischen dem Eingabestrom in und dem Ausgabestrom out von *PR* festgelegt. Das Prädikat setzt sich aus einer Reihe von Funktionsgleichungen zusammen, die aus den Zustandsübergängen des SDL-Prozesses abgeleitet werden. In Abschnitt 4.3.2 werden wir diese Umsetzung vorstellen.

4.3.1.3 Modellierung der Signalverteilung

Das Verteilen der Ausgabenachrichten von *PR* auf die entsprechenden Ausgabekanäle des SDL-Prozesses wird von der Verteilkomponente *Split* übernommen. *Split* bildet das Gegenstück zur Komponente *FairMerge*. Die Komponente *Split* filtert für jeden Ausgabekanal die dazugehörigen Ausgabenachrichten aus ihrem Eingabestrom heraus. Somit ist die Spezifikation von *Split* ähnlich zur Spezifikation von *FairMerge*. Wie dort setzen wir voraus, daß die Nachrichtenmengen der Ausgabekanäle paarweise disjunkt sind $(\forall\, a, b \in \{1, \ldots, k\}, a \neq b :\ O_a \cap O_b = \varnothing)$.

agent *Split* – **time dependent** – **wp**

input channel $i : I$
output channel $o_1 : O_1 \ \ldots \ o_k : O_k$

is basic

$$\begin{aligned} o_1 &= (O_1 \cup \{\surd\}) \copyright i \wedge \\ &\vdots \qquad \vdots \\ o_k &= (O_k \cup \{\surd\}) \copyright i \end{aligned}$$

end *Split*

Die Ein- und Ausgabekanäle sowie die Nachrichtenmengen von *Split* sind jeweils an die konkrete Netzwerkstruktur der Komponente *Prozeß* anzupassen. Der Kanal $timer_{in}$ gehört nur zu den Ausgabekanälen von *Split*, wenn der gegebene SDL-Prozeß über Timer verfügt.

4.3.1.4 Modellierung der Verzögerung von Timersignalen

In Abschnitt 2.3 haben wir das Timerkonzept von SDL diskutiert und uns entschieden, statt Timern mit quantitativen Ablaufzeitpunkten ein Timerkonzept zu verwenden, bei dem die gesetzten Timer nach beliebig langer, aber endlicher Zeit ablaufen und in den Eingabepuffer des Prozesses eingereiht werden. Das Setzen eines Timers wird innerhalb der Komponente *PR* durch die Ausgabe einer Nachricht modelliert, die den Namen des Timers trägt (siehe Abschnitt 4.3.2.6). Die unbestimmte Verzögerung der Timernachrichten wird durch die Komponente *Timer* modelliert, die zwischen den Komponenten *Split* und *FairMerge* eingefügt wird. Sie leitet die Timernachrichten, die sie von *Split* erhält, nach endlich, aber beliebig langer Zeit an *FairMerge* weiter. Dabei bleibt die Reihenfolge der Nachrichten erhalten (FIFO-Prinzip). Die Timer-Komponente weist somit das gleiche Verhalten wie eine Kanalverzögerungskomponente auf (vgl. Abschnitt 4.1.2) und wird als zeitunabhängige Basiskomponente spezifiziert. Sei *timeout* der Name des Timers im SDL-Prozeß und damit auch der Name der dazugehörigen Timernachrichten; die Spezifikation von *Timer* lautet:

agent *Timer* – **time independent**

input channel $timer\_in : \{timeout\}$
output channel $timer\_out : \{timeout\}$

is basic

$$timer\_in = timer\_out$$

end *Timer*

Falls ein Anwender in einer SDL-Prozeßspezifikation verschiedene Timer verwendet, so gilt das eben vorgestellte Verzögerungsprinzip für jeden einzelnen Timer; die Verzögerungen

für die einzelnen Timer sind voneinander unabhängig. Statt für jeden Timer eine eigene Verzögerungskomponente anzugeben, modellieren wir die Verzögerungen für die unterschiedlichen Timer in einer Komponente *MultipleTimer*. Abbildung 4.11 zeigt einen Ausschnitt aus dem strukturellen Aufbau einer Komponente der Art *Prozeß*, wenn innerhalb des SDL-Prozesses mehrere Timer definiert sind (vgl. Abbildung 4.8).

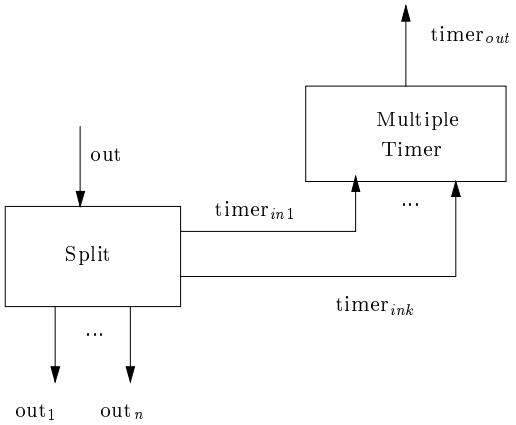


Abbildung 4.11: Ausschnitt des strukturellen Aufbaus der Komponente *Prozeß* mit *MultipleTimer*

Für jeden Timer $timer_i$ existiert ein Kanal $timer_{in_i}$ zwischen der Verteilkomponente *Split* und *MultipleTimer*. Diese mischt die Eingabeströme zu einem Ausgabestrom *timer_out*, wobei die Timernachrichten während des Mischvorgangs eine beliebige zeitliche Verzögerung erfahren.

agent *MultipleTimer* – **time independent**

input channel $timer_{in_1} : \{timeout_1\}, \ldots, timer_{in_k} : \{timeout_k\}$
output channel $timer_{out} : \{timeout_1\} \cup \ldots \cup \{timeout_k\}$

is basic

$$\begin{aligned} timer_{in_1} &= \{timeout_1\} \copyright timer_{out} \wedge \\ &\vdots \qquad \vdots \\ timer_{in_k} &= \{timeout_k\} \copyright timer_{out} \end{aligned}$$

end *MultipleTimer*

4.3.1.5 Modellierung von spontanen Übergängen

Die Komponente *None* wird innerhalb der Komponente *Prozeß* hinzugefügt, wenn im zugrundeliegenden SDL-Prozeß spontane Zustandsübergänge, gekennzeichnet durch das Schlüsselwort *none* in einem Eingabesymbol, vorliegen. Die Komponente *None* erhält keine Eingabenachrichten und erzeugt auf nichtdeterministische Weise einen Strom von Ausgabenachrichten *none* (vgl. [HS94]). Diese Nachrichten werden über den Kanal p an die Komponente *FairMerge* gesendet und in den Eingabestrom der Komponente *PR* eingefügt. *PR*

behandelt *none*-Nachrichten wie gewöhnliche Eingabenachrichten. Liegt in einem Zustand ein spontaner Übergang vor, so wird dieser ausgeführt, falls die nächste zu verarbeitende Nachricht *none* ist.

Im Rumpf der Basiskomponente *None* wird durch den Wahrheitswert *true* festgelegt, daß es keine Bedingungen an die Ausgabeströme, die nur aus *none*-Nachrichten bestehen, gibt. Somit ist jeder beliebig gezeitete Strom mit *none*-Nachrichten ein gültiger Ausgabestrom der Komponente. Über die Wahrscheinlichkeit der spontanen Übergänge wird in [IT93b] keine Aussage getroffen, so daß wir an die Zusammensetzung des Stroms keine weiteren Bedingungen stellen. Somit sind auch ein Strom ohne *none*-Nachrichten und ein Strom, der unendlich viele *none*-Nachrichten enthält, gültige Ausgabeströme der Komponente *none* (siehe auch Abschnitt 2.5).

agent *None* – **time independent**

input channel
output channel $p : \{none\}$

is basic

true

end *None*

4.3.2 Ableitung von Funktionsgleichungen aus einem SDL-Prozeß

Das Verhalten der Komponente *PR* leitet sich aus dem Zustandsautomaten ab, der durch den SDL-Prozeßgraphen spezifiziert ist. Um das Verhalten von *PR* zu spezifizieren, werden aus dem SDL-Prozeßgraphen eine Menge von Funktionsgleichungen abgeleitet. Mittels dieser Gleichungen wird eine stromverarbeitende Funktion definiert, die die Beziehung zwischen dem Ein- und dem Ausgabestrom von *PR* beschreibt.

Wir spezifizieren das Verhalten einer Prozeßkomponente durch eine stromverarbeitende Funktion *behaviour*, die einen Strom von Eingabenachrichten erhält und diesen auf einen Strom von Ausgabenachrichten abbildet. Wie in Abschnitt 4.3.1.2 erläutert, werden wir das ungezeitete Spezifikationsformat für Basiskomponenten wählen. Die Komponente *PR* erhält von *FairMerge* zwar einen gezeiteten Eingabestrom, in ihrer Spezifikation wird diese Zeitinformation jedoch nicht berücksichtigt. Die Funktion *behaviour* wird über ungezeiteten Strömen definiert und verfügt über einen Parameter der Sorte D für den Datenzustand des SDL-Prozesses. Dieser Parameter umfaßt, wie in Abschnitt 4.2 beschrieben, die aktuelle Belegung der lokalen Variablen des Prozesses. Es ergibt sich folgende Funktionalität für die Funktion *behaviour*:

$$behaviour : D \times In^{\omega} \rightarrow Out^{\omega}$$

Dabei bezeichnen In die Menge der Eingabenachrichten und Out die Menge der Ausgabenachrichten von *PR*, die sich aus den Ein- bzw. Ausgabesignalen des SDL-Prozesses sowie möglichen Timer- und *none*-Nachrichten zusammensetzen.

In ANDL lautet die Spezifikation der Komponente *PR* nun folgendermaßen:

agent *PR* – **time independent**

input channel $in : In$
output channel $out : Out$

is basic

$\exists\ behaviour : D \times In^{\omega} \rightarrow Out^{\omega}.\ behaviour\ [\sigma_{init}]\ (in)\ = out$

where

Funktionsgleichungen

end *PR*

Es wird gefordert, daß eine Funktion *behaviour* existiert, die die Beziehung zwischen dem Eingabestrom *in* und dem Ausgabestrom *out* in Abhängigkeit des initialen Datenzustands σ_{init} beschreibt. Wie die Funktion aus dem Eingabestrom den Ausgabestrom berechnet, wird durch die Funktionsgleichungen definiert. Um eine bessere Lesbarkeit der Funktionsgleichungen zu erreichen, schließen wir – sofern vorhanden – die Variablen des Datenzustands und eventuelle weitere Parameter in eckige Klammern ein.

Die Funktion *behaviour* berechnet den Strom von Ausgabenachrichten, indem sie für jede Eingabenachricht die Berechnungen durchführt, die dem jeweiligen Zustandsübergang aus dem zugrundeliegenden SDL-Prozeß entsprechen. Dabei hängt das Verhalten von *behaviour* davon ab, in welchem Zustand sich der SDL-Prozeß beim Verarbeiten des Eingabesignals befindet. Wir führen für jeden Zustand Z des SDL-Prozesses eine Funktion ein, die mit dem Namen des Zustands, also mit Z, bezeichnet wird. Diese Funktion verfügt über die gleiche Funktionalität wie die Funktion *behaviour* und beschreibt das Verhalten des Prozesses im Zustand Z.

$$Z : D \times In^{\omega} \rightarrow Out^{\omega}$$

Die Zustandsübergänge, die von diesem Zustand ausgehen, werden in Funktionsgleichungen umgesetzt. Eine Funktionsgleichung für einen Zustandsübergang von Z nach Z' entspricht folgendem Schema:

$$Z\ [\sigma]\ (a\ \&\ in)\ =\ b_1\ \&\ \ldots\ \&\ b_n\ \&\ Z'\ [\sigma']\ (in)$$

Die Funktion Z erhält den Eingabestrom $a\&in$, erzeugt abhängig von der ersten Nachricht a eine Folge von Ausgabenachrichten $b_1, \ldots, b_n$ und ruft mit dem restlichen Eingabestrom *in* die Funktion Z' auf, die den Folgezustand repräsentiert. Mögliche Veränderungen des Datenzustands werden durch den Übergang von σ nach σ' erfaßt.

Die Funktion *behaviour* legt die Beziehung zwischen dem vollständigen Eingabestrom *in* und dem vollständigen Ausgabestrom *out* fest. Sie stützt sich dabei auf die Funktionen, die den einzelnen Prozeßzuständen zugeordnet sind. Der Eingabestrom wird nachrichtenweise abgearbeitet, wobei die jeweiligen Funktionen für die Zustände aufgerufen und die Zustandsübergänge in Form der obigen Funktionsgleichungen durchgeführt werden. Zu den

aus dem SDL-Prozeßgraphen abgeleiteten Funktionsgleichungen kommen weitere Funktionsgleichungen für implizite Transitionen und die Behandlung von Timern hinzu.

Ferner ist das Verhalten des SDL-Prozesses zu modellieren, wenn der leere Strom $\langle\rangle$ als Eingabe für die Funktion *behaviour* bzw. die einzelnen Zustandsfunktionen Z vorliegt. Ein leerer Eingabestrom bedeutet, daß keine weiteren Eingabenachrichten mehr vorliegen und die Komponente PR keine Ausgabenachrichten mehr erzeugen kann. Es erfolgt kein weiterer Funktionsaufruf, sondern die Ausgabe des leeren Stroms. Dies entspricht folgender Funktionsgleichung (für beliebige Zustandsfunktionen Z bzw. *behaviour*):

$$Z\ [\sigma]\ (\langle\rangle)\ =\ \langle\rangle$$

Anstatt diese Gleichung für jeden Zustand explizit anzugeben, setzen wir voraus, daß die Funktion *behaviour* und die Zustandsfunktionen strikt sind. Eine stromverarbeitende Funktion ist strikt, wenn sie bei Eingabe des leeren Stroms den leeren Strom als Ausgabe liefert[2].

Das Ableiten der Funktionsgleichungen kann durch eine bestimmte Form der SDL-Prozeßspezifikation vereinfacht werden. Damit können schematisierte Gleichungen erzielt werden, was zu einem besseren Verständnis der Semantikdefinition beiträgt. Zugleich unterstützen die folgenden Bedingungen auch eine übersichtlichere Form der SDL-Spezifikation.

- Die lokalen Variablen des Prozesses werden bereits bei ihrer Deklaration im Textsymbol und nicht erst während des Prozeßablaufs initialisiert (vgl. Seite 32).
- Berechnungen auf den lokalen Variablen werden ausschließlich in den Tasksymbolen durchgeführt und nicht in den Entscheidungssymbolen oder bei der Ausgabe von Signalen mit Datenwerten. Alternativen bei Entscheidungen sind entweder Variablen oder Konstanten.

In den nächsten Abschnitten stellen wir die semantische Behandlung der einzelnen Prozeßkonstrukte aus dem SDL-Prozeßgraphen in den Funktionsgleichungen vor. Neben dem Parameter für den Datenzustand eines SDL-Prozesses sind einige weitere Parameter in der Funktion *behaviour* und somit für die Funktionen, die den Zuständen zugeordnet sind, erforderlich, so zum Beispiel für die Modellierung nichtdeterministischer Entscheidungen und die Behandlung von Timernachrichten innerhalb der Komponente PR. Wir werden die entsprechenden Erweiterungen der Funktionalität bei der Behandlung dieser Sprachkonstrukte angeben.

Die Werte folgender Variablen in den Funktionsgleichungen setzen wir als beliebig voraus: $in : In^{\omega},\ \sigma : D,\ c : Nat,\ b : Bool$. Die Kennzeichnung „'“ einer Variablen drückt eine mögliche, nicht näher definierte Änderung des Werts der Variablen aus.

[2]Für Ströme des Typs N^{ω} ist der leere Strom $\langle\rangle$ das kleinste Element in der Präfixordnung der Ströme.

4.3.2.1 Startsymbol, Ein-/Ausgabe von Signalen, Implizite Transitionen

Die Anfangstransition des SDL-Prozesses erfolgt ohne Verbrauch eines Eingabesignals und führt vom Startsymbol bis zum darauffolgenden Zustandssymbol Z. Die Funktion *behaviour* modelliert die Anfangstransition des Prozesses und ruft anschließend die Funktion Z für den Folgezustand Z auf. Während dieser Transition können Ausgabesignale erzeugt und der Datenzustand verändert werden. Der Datenparameter σ_{init} der Funktion *behaviour* bildet den initialen Datenzustand des Prozesses und wird, wie im vorherigen Abschnitt beschrieben, aus den initialen Zuweisungen an die Prozeßvariablen bestimmt, die im Textsymbol erfolgen.

Die Anfangstransition entspricht folgender Funktionsgleichung:

$$behaviour\ [\sigma_{init}]\ (in)\ =\ o_1\ \&\ \ldots\ \&\ o_n\ \&\ Z\ [\sigma'_{init}]\ (in)$$

Wir veranschaulichen das Vorgehen anhand des Prozeßausschnitts in Abbildung 4.12:

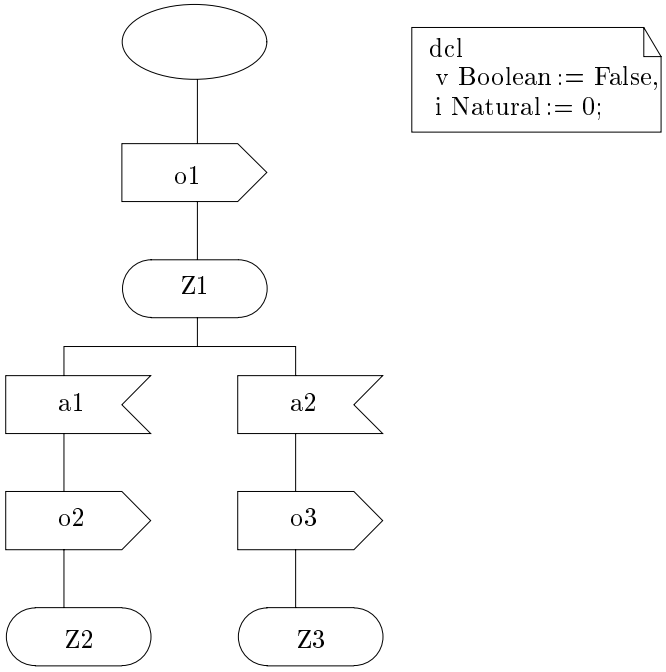


Abbildung 4.12: SDL-Prozeßausschnitt mit Startsymbol

Die im Prozeß graphisch spezifizierten Zustandsübergänge werden in folgende Funktionsgleichungen umgesetzt:

Anfangstransition:
$$behaviour\ [\sigma_{init}]\ (in)\ =\ o1\ \&\ Z1\ [\sigma_{init}]\ (in)$$
mit $\sigma_{init} = md\ (False,\ 0)$

Alle darauffolgenden Funktionsgleichungen gelten nun für beliebige Datenzustände σ:

$Z1$ nach $Z2$:
$$Z1\ [\sigma]\ (a1\ \&\ in)\ =\ o2\ \&\ Z2\ [\sigma]\ (in)$$

$Z1$ nach $Z3$:
$$Z1\ [\sigma]\ (a2\ \&\ in)\ =\ o3\ \&\ Z3\ [\sigma]\ (in)$$

Zusätzlich sind in jedem Zustand die **impliziten Transitionen** zu modellieren. Alle Signale, die zur Menge In der Eingabesignale des Prozesses gehören und nicht als Eingabe für einen Zustand Z spezifiziert sind, werden in einer impliziten Transition verarbeitet, die zurück in den ursprünglichen Zustand Z führt. Der Datenzustand bleibt dabei unverändert.

Für unser Beispiel lautet die Gleichung für die implizite Transition in Zustand $Z1$:

$$\alpha \in (In \setminus \{a1, a2\}) \implies Z1\ [\sigma]\ (\alpha\ \&\ in) = Z1\ [\sigma]\ (in)$$

4.3.2.2 Tasks, Entscheidungen und Signale mit Daten

Ein wesentlicher Punkt bei der semantischen Fundierung des datenabhängigen Verhaltens von SDL-Prozessen ist die Frage nach der Terminierung von Datenausdrücken. Um die SDL-Semantikdefinition möglichst einfach zu gestalten, gehen wir von der Vereinbarung aus, daß Ausdrücke ausschließlich in den Tasksymbolen verwendet werden. Damit kann sichergestellt werden, daß die Ausführung von Entscheidungen und die Ausgabe von Signalen mit Daten terminiert. Die Terminierung von Ausdrücken muß somit nur bei der semantischen Behandlung der Tasksymbole und ihrer Inhalte geprüft werden.

Anweisungen in Tasks (Abbildung 4.13) manipulieren den Datenzustand des Prozesses.

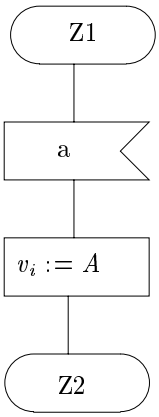


Abbildung 4.13: Zustandsübergang mit Task

Eine Anweisung der Form $v_i := A$, wobei $v_i : adt_i$ gilt und A ein Ausdruck der Sorte adt_i ist, entspricht folgender Veränderung von σ:

σ geht über in $\sigma[v_i := A]$.

Zustandsübergänge mit Tasks werden nun wie folgt umgesetzt:

$$Z1\ [\sigma]\ (a\ \&\ in)\ =\ \mathsf{if}\ A_\sigma = \bot\ \mathsf{then}\ \langle\rangle\ \mathsf{else}\ Z2\ [\sigma\ [v_i := A]]\ (in)\ \mathsf{endif}$$

A_σ bezeichnet den Ausdruck A, in dem die in A frei vorkommenden Variablen $v_1, \ldots, v_n$ durch ihre aktuellen Belegungen in σ (also durch $v_1(\sigma), \ldots, v_n(\sigma)$) ersetzt werden. Das Verhalten des Prozesses hängt davon ab, ob der Ausdruck A in Abhängigkeit des aktuellen

Datenzustands einen definierten Wert oder $\perp$ liefert. Trifft die Bedingung $A_\sigma = \perp$ zu, so ist das weitere Verhalten der Komponente *PR* undefiniert. Die Komponente reagiert nicht mehr auf den restlichen Strom von Eingabenachrichten und produziert keine weiteren Ausgabenachrichten. Der Zustandsübergang wird mit der Ausgabe des leeren Stroms abgeschlossen.
Mit der Überprüfung der Terminierung von Ausdrücken in unserer Semantikdefinition ist nun die ungenügende Behandlung partieller Funktionen und nichtterminierender Ausdrücke in SDL behoben (siehe Abschnitt 2.4).

Entscheidungen werden durch bedingte Anweisungen umgesetzt. Abhängig vom Wert des Entscheidungsausdrucks wird eine der vorgegebenen Alternativen gewählt. Da wir fordern, daß Ausdrücke nur in Tasksymbolen berechnet werden, handelt es sich beim Entscheidungsausdruck um eine der lokalen Variablen des Prozesses. Bei den Alternativen handelt es sich entweder um eine lokale Variable oder einen Datenwert (Konstante), wobei diese vom gleichen Typ wie die Variable in der Entscheidung sind.

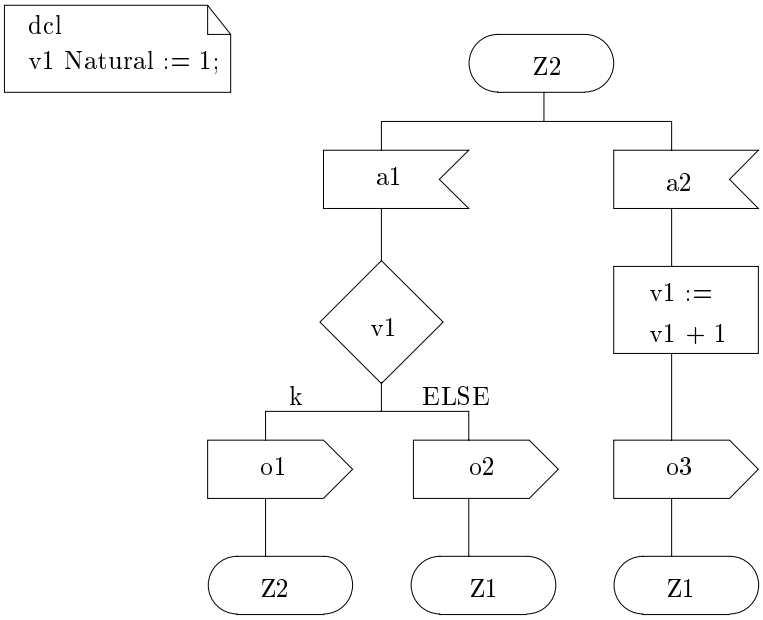


Abbildung 4.14: SDL-Prozeßausschnitt mit Task und Entscheidung

Die Funktionsgleichungen für den Prozeßausschnitt in Abbildung 4.14 lauten:

Modellierung der Anweisung im Tasksymbol:

$$Z2\ [\sigma]\ (a2\ \&\ in) =$$
$$\mathsf{if}\ (v1+1)_\sigma = \perp\ \mathsf{then}\ \langle\rangle\ \mathsf{else}\ o3\ \&\ Z1\ [\sigma\ [v1 := v1+1]]\ (in)\ \mathsf{endif}$$

Modellierung der Entscheidung:

$$Z2\ [\sigma]\ (a1\ \&\ in) = \mathsf{if}\ v1(\sigma) = k\ \mathsf{then}\ o1\ \&\ Z2\ [\sigma]\ (in)\ \mathsf{else}\ o2\ \&\ Z1\ [\sigma](in)\ \mathsf{endif}$$

Anmerkung:
Eine explizite Abfrage nach der Terminierung der Auswertung einer Anweisung ist notwendig und kann nicht durch die Forderung ersetzt werden, daß die Zustandsfunktionen im ersten Argument strikt sind. Dies würde für den rechten Zustandsübergang in Abbildung 4.14 zu folgender Gleichung führen:

$$Z2\ [\sigma]\ (a2\ \&\ in) = o3\ \&\ Z1\ [\sigma\ [v1 := v1 + 1]]\ (in)$$

Die Ausgabe von $o3$ durch den SDL-Prozeß erfolgt nur, wenn die vorherige Anweisung terminiert; dies kann mittels der Striktheit der Funktion nicht adäquat modelliert werden. Die Auswertung obiger Gleichung würde auch zur Ausgabe von $o3$ führen, wenn die Anweisung $v1 := v1 + 1$ nicht terminiert.

Bei Entscheidungen mit mehr als zwei Alternativen kann statt verschachtelter bedingter Anweisungen das *case*-Konstrukt verwendet werden, um eine übersichtlichere Darstellung der Funktionsgleichung zu erhalten. Eine Entscheidung über der Variablen v mit l Alternativen, die in die Zustände $Z1$ bis Zl führen, läßt sich in folgendes Schema umsetzen:

$$\begin{aligned} Z\ [\sigma]\ (a\ \&\ in) = \mathsf{case}\ v\ \mathsf{of}\ & k1: \ o1\ \&\ Z1\ [\sigma']\ (in) \\ & k2: \ o2\ \&\ Z2\ [\sigma']\ (in) \\ & \vdots \\ & kl: \ ol\ \&\ Zl\ [\sigma']\ (in) \\ \mathsf{endcase}& \end{aligned}$$

Signale können **mit Daten** versehen sein. Führt ein SDL-Eingabesignal Daten mit sich, so werden diese an lokale Variablen des Prozesses übergeben. Im Eingabesymbol sind die Namen dieser Variablen in Klammern hinter dem Signalnamen aufgeführt. Das Verarbeiten eines Signals mit Daten führt zu einer Veränderung des Datenzustands des Prozesses. Bei der Ausgabe von Signalen mit Daten werden die aktuellen Datenwerte des Signals anhand der Variablenangaben im Ausgabesymbol des SDL-Prozesses bestimmt.

SDL-Signale mit Daten `signal <name> (sort`$_1$`, ..., sort`$_n$`)` werden in der FOCUS-Semantikdefinition in eine Nachrichtenmenge umgesetzt (vgl. Abschnitt 4.1). Dabei werden die SDL-Typen $sort_i$ auf SPECTRUM-Datentypen adt_i abgebildet.

$$N = \{(name,\ data_1, \ldots,\ data_n) \mid data_i : adt_i \text{ für } 1 \leq i \leq n\}$$

In der Semantikdefinition werden die Daten, die Signale mit sich führen, explizit modelliert. Die Komponente PR erhält in ihrem Eingabestrom somit Signale der Art $(name, d_1, \ldots, d_n)$. Anhand des Eintrags im dazugehörigen Eingabesymbol $(name(v_1, \ldots, v_n))$ des gegebenen SDL-Prozesses kann festgestellt werden, welchen lokalen Variablen des Prozesses die Daten $d_1, \ldots, d_n$ zuzuweisen sind.

Es ist zu berücksichtigen, daß Parameter von Signalen in SDL nicht zwingend mit Daten belegt sein müssen. In diesem Fall erfolgt beim Verarbeiten des Signals keine Zuweisung der Datenwerte der Parameter an die entsprechenden lokalen Variablen des Prozesses. In

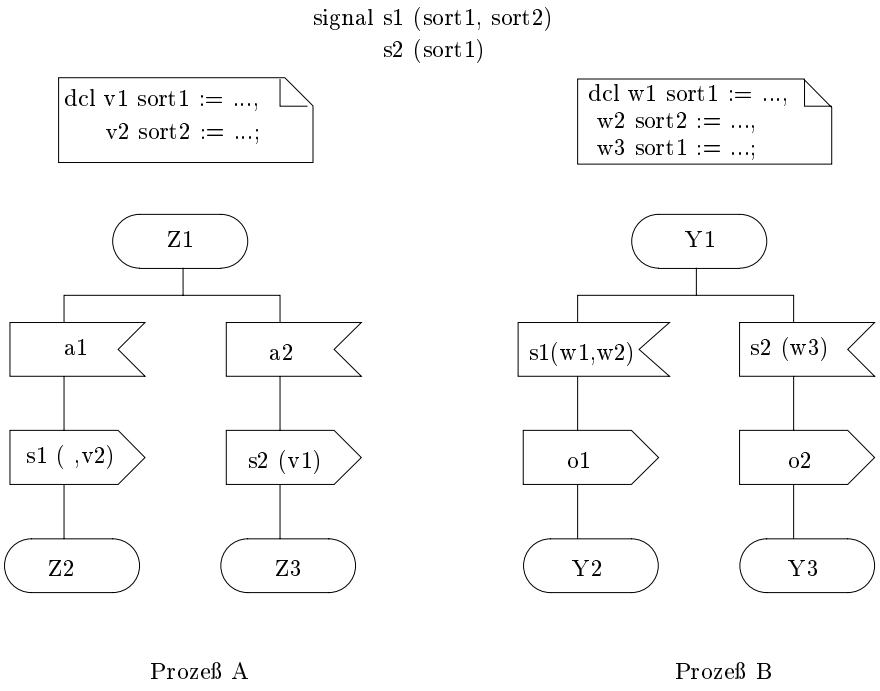


Abbildung 4.15: SDL-Prozeßausschnitte mit Signalen mit Parametern

der Semantikdefinition kann dies nicht mit dem Wert $\perp$ für undefinierte Werte modelliert werden. Statt dessen ist die Einführung eines gesonderten Wertes erforderlich, der den Wert nichtbelegter Parameter repräsentiert. Wir führen den polymorphen[3] Wert ϕ ein. Liegt in einer SDL-Spezifikation ein Signal vor, dessen Parameter nicht durchgehend mit Datenwerten belegt sind, so werden die Datentypen der betroffenen Parameter um das Symbol ϕ erweitert. In den Funktionsgleichungen ist beim Empfang eines solchen Signals zu prüfen, ob der Wert ϕ oder ein anderer Wert als Parameter vorliegt. Nur in letzterem Fall erfolgt eine Zuweisung an die lokalen Variablen. Werden bei der Ausgabe eines Signals mit Parametern keine Datenwerte angegeben, so ist in den zugehörigen Funktionsgleichungen das Symbol ϕ als aktueller Wert für die betroffenen Parameter zu verwenden.

Abbildung 4.15 zeigt zwei Prozeßausschnitte, in denen das Senden und Empfangen von Signalen mit Parametern spezifiziert wird. Dabei sind bis auf Ausnahme des ersten Parameters von Signal $s1$ alle Parameter mit Datenwerten belegt. Die SDL-Datentypen $sort1$ bzw. $sort2$ werden auf die SPECTRUM-Typen adt_1 bzw. adt_2 abgebildet. Die Funktionsgleichungen für Abbildung 4.15 lauten (mit $\forall\, d, d1 : adt_1,\ \ d2 : adt_2$):

Spezifikation der Zustandsübergänge in Prozeß A:

$$Z1\ [\sigma]\ (a1\ \&\ in)\ =\ (s1,\ \phi,\ v2(\sigma))\ \&\ Z2\ [\sigma]\ (in)$$

$$Z1\ [\sigma]\ (a2\ \&\ in)\ =\ (s2,\ v1(\sigma))\ \&\ Z3\ [\sigma]\ (in)$$

[3]Ein polymorpher Wert ist für eine Menge unterschiedlicher Datentypen verfügbar.

Spezifikation der Zustandsübergänge in Prozeß B:

$Y1\ [\sigma]\ ((s1,\ d1,\ d2)\ \&\ in) =$
if $d1 = \phi$
  then $o1\ \&\ Y2\ [\sigma[w2 := d2]]\ (in)$
  else $o1\ \&\ Y2\ [\sigma[w1 := d1,\ w2 := d2]]\ (in)$
endif

$Y1\ [\sigma]\ ((s2,\ d)\ \&\ in)\ \ =\ \ o2\ \&\ Y3\ [\sigma[w3 := d]]\ (in)$

Anmerkung:
In SDL-Prozessen tritt häufig der Fall auf, daß auf die Daten von Signalparametern nur in dem Zustandsübergang Bezug genommen wird, der durch das Einlesen des Signals initiiert wurde. In darauffolgenden Transitionen wird auf die Werte der Parameter nicht mehr zugegriffen. In diesem Fall kann darauf verzichtet werden, eigene Variablen für die Signalparameter in den Funktionen vorzusehen und diesen die Parameterwerte zuzuweisen. Dadurch werden die Funktionsgleichungen übersichtlicher. Allerdings ist es bei umfangreichen Prozeßspezifikationen schwierig, die Verwendung von Signalparametern vollständig zu analysieren, so daß in diesem Fall die Definition sämtlicher Prozeß- und Signalvariablen zu erfolgen hat.

4.3.2.3 Stopsymbol

Nach der Interpretation eines Stopsymbols existiert ein SDL-Prozeß nicht mehr, d.h. er erzeugt keine Ausgabesignale mehr und reagiert nicht länger auf Eingabesignale. Die zugeordnete Komponente in FOCUS weist somit kein nach außen beobachtbares Verhalten in Form von Ein- und Ausgabenachrichten mehr auf.

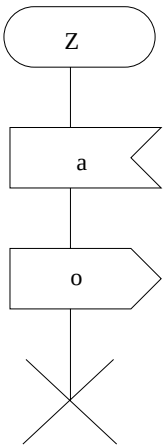


Abbildung 4.16: SDL-Prozeßausschnitt mit Stopsymbol

Bei der Ableitung der Funktionsgleichung werden die Prozeßsymbole bis zum Stopsymbol wie bisher beschrieben umgesetzt. Es erfolgt jedoch kein weiterer Funktionsaufruf; die Gleichung endet mit der Ausgabe des leeren Stroms.

Die Funktionsgleichung für den Übergang von einem Zustand Z in das Stopsymbol, wie er in Abbildung 4.16 dargestellt ist, lautet:

$$Z\ [\sigma]\ (a\ \&\ in)\ =\ o\ \&\ \langle\rangle$$

4.3.2.4 Savesymbol

In den Savesymbolen eines Zustands sind Signale eingetragen, die nicht durch implizite Transitionen zerstört werden, sondern für nachfolgende Zustände im Eingabepuffer verbleiben. Für unsere Semantikdefinition bedeutet dies folgendes: Die Funktion, die einem Zustand mit Savesignalen S = $\{s_1,\ \ldots,\ s_k\}$ zugeordnet ist, darf nicht auf das erste Element des Eingabestroms zugreifen und es in einer impliziten Transition zerstören, falls das Element in der Menge der Savesignale enthalten ist. Vielmehr wird der Eingabestrom nach dem ersten Vorkommen eines Signal durchsucht, das kein Savesignal ist und somit einen Zustandsübergang initiieren kann. Ist ein solches Signal im Strom gefunden, so wird die entsprechende Funktionsgleichung wie für Zustände ohne Savesignale durchgeführt. Das Signal wird dabei aus dem Eingabestrom gelöscht.

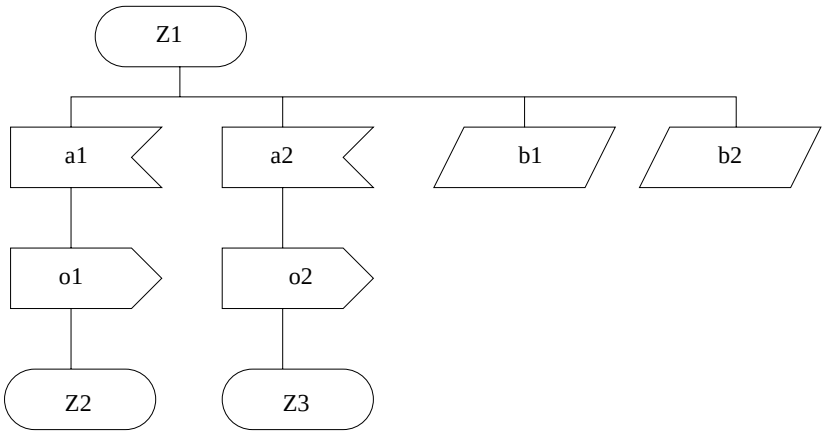


Abbildung 4.17: SDL-Prozeßausschnitt mit Savesignalen

Wir führen für Zustände mit Savesignalen eine Funktion *search* ein, die im Eingabestrom nach dem ersten Element sucht, das nicht in der Menge der Savesignale liegt[4] und einen Zustandsübergang initiiert, und eine Funktion *del*, die dieses Element anschließend aus dem Eingabestrom löscht.

$$search : Set(In) \times In^{\omega} \rightarrow In$$

$$del : In \times In^{\omega} \rightarrow In^{\omega}$$

[4] $Set(In)$ bezeichnet den Datentypen der Mengen, die aus Nachrichten des Typs In bestehen.

$search(M, m \;\&\; s) =$ if $m \in M$ then m else $search(M, s)$ endif

$del(e, m \;\&\; s) =$ if $e = m$ then s else $m \;\&\; del(e, s)$ endif

Wir veranschaulichen die Behandlung von Savesignalen am Prozeßausschnitt in Abbildung 4.17. Die Signale $b1$ und $b2$ sind die Savesignale des Zustands $Z1$, $a1$ und $a2$ Signale, die einen Zustandsübergang auslösen. Ferner nehmen wir an, daß das Signal $a3$ zu den gültigen Eingabesignalen des Prozesses gehört und in Zustand $Z1$ in einer impliziten Transition verbraucht wird.

$Z1\ [\sigma]\ (in) =$
case $search\ (\{In \setminus \{b1, b2\}\},\ in)$ of

$a1:$	$o1 \;\&\; Z2\ [\sigma]\ (del(a1, in))$	Gültiges Eingabesignal $a1$
$a2:$	$o2 \;\&\; Z3\ [\sigma]\ (del(a2, in))$	Gültiges Eingabesignal $a2$
$a3:$	$Z1\ [\sigma]\ (del(a3, in))$	Implizite Transition

endcase

Falls der Eingabestrom in im Zustand $Z1$ nur aus Savesignalen besteht, bedeutet dies, daß der SDL-Prozeß keinen Zustandsübergang mehr durchführen kann. In unserer Semantikdefinition liefert die Funktion *search* in diesem Fall den Wert $\perp$. Bedingt durch die Striktheit des *case*-Konstrukts und der Funktion $Z1$ ergibt $Z1$ angewandt auf in somit den leeren Strom (also das $\perp$-Element der Ströme). Dies ist damit gleichzusetzen, daß der SDL-Prozeß kein Verhalten mehr nach außen zeigt.

4.3.2.5 Spontane Übergänge und nichtdeterministische Entscheidungen

Nichtdeterministisches Verhalten in SDL-Prozessen kann durch spontane Übergänge und nichtdeterministische Entscheidungen modelliert werden. Da in der Definition von SDL ([IT93b]) über die Fairneß bei nichtdeterministischen Auswahlmöglichkeiten keine klare Aussage vorliegt, werden wir in unserer Semantikdefinition keine Fairneßbedingungen bei *none* und *any* berücksichtigen (siehe auch Abschnitte 2.5 und 4.3.1.5).

Spontane Übergänge, die durch das Schlüsselwort *none* in einem Eingabesymbol gekennzeichnet sind, modellieren wir durch das Einfügen von *none*-Nachrichten in den Eingabestrom der Prozeßkomponente *PR*. Dazu führen wir im Netzwerk *Prozeß* die Komponente *None* ein, die einen Strom mit beliebig vielen *none*-Nachrichten erzeugt (siehe Abschnitt 4.3.1.5). Spontane Übergänge werden somit in der Semantikdefinition bei der Angabe der Funktionsgleichungen wie Zustandsübergänge behandelt, die durch das Verarbeiten eines Signals initiiert werden. Die Menge der Eingabenachrichten der Komponente *PR* wird um das Signal *none* erweitert: $In \cup \{none\}$.

Für den linken Prozeßausschnitt in Abbildung 4.18 lauten die Funktionsgleichungen:

Z1 nach Z2:	$Z1\ [\sigma]\ (a \;\&\; in) = o1 \;\&\; Z2\ [\sigma]\ (in)$
Z1 nach Z3:	$Z1\ [\sigma]\ (none \;\&\; in) = o2 \;\&\; Z3\ [\sigma]\ (in)$
Implizite Transition:	$\forall \alpha \notin \{a, none\} : Z1\ [\sigma]\ (\alpha \;\&\; in) = Z1\ [\sigma]\ (in)$

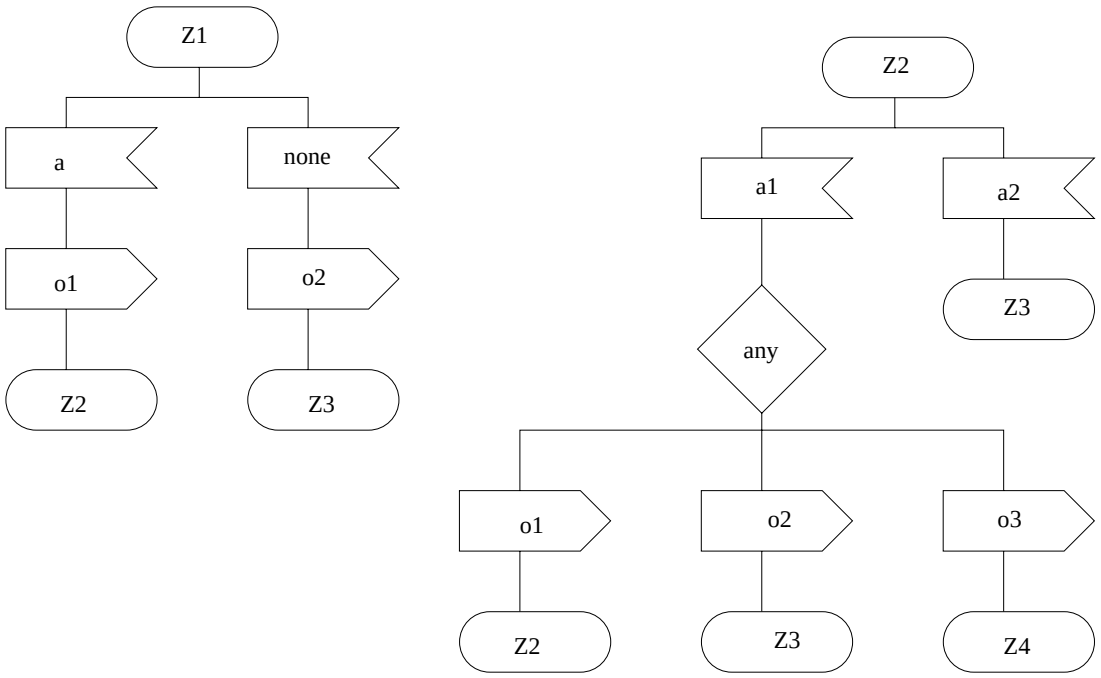


Abbildung 4.18: SDL-Prozeßausschnitt mit *none* und *any*

Für die Umsetzung **nichtdeterministischer Entscheidungen**, gekennzeichnet durch das Schlüsselwort *any* im Entscheidungssymbol, führen wir ein Orakel ein (vgl. [HS94]). Dieses bestimmt, welche der möglichen Alternativen bei der Durchführung der Entscheidung gewählt wird. Das Orakel ist ein unendlich langer Strom von natürlichen Zahlen. Bei der Auswertung einer nichtdeterministischen Entscheidung mit i Alternativen wird die Alternative $(r \ mod \ i)+1$ gewählt, wobei r das erste Element des Orakels darstellt. Anschließend wird das erste Element des Orakels gelöscht. Bei Zustandsübergängen, die keine nichtdeterministische Entscheidung enthalten, bleibt das Orakel unverändert. Da über die Fairneß der Auswertung von *any* in [IT93b] keine Aussage zu finden ist, verzichten wir auf Bedingungen bezüglich der Zusammensetzung des Orakels. Die Funktionalität der Funktion *behaviour* in der Definiton der Basiskomponente PR wird um einen zusätzlichen Parameter für das Orakel erweitert:

$$\exists\ or \in Nat^{\infty},\ \exists\ behaviour \in Nat^{\infty} \times D\ \times In^{\omega} \rightarrow Out^{\omega}\ :$$
$$behaviour\ [or,\ \sigma_{init}]\ (in)\ =\ out$$

Die Funktionsgleichungen für den rechten Prozeßausschnitt in Abbildung 4.18 lauten:

$$Z2\ [r\ \&\ or,\ \sigma]\ (a1\ \&\ in)\ =$$
case $(r\ mod\ 3)$ of
$0:\ o1\ \&\ Z2\ [or,\ \sigma]\ (in)$
$1:\ o2\ \&\ Z3\ [or,\ \sigma]\ (in)$
$2:\ o3\ \&\ Z4\ [or,\ \sigma]\ (in)$
endcase

$$Z2\ [r\ \&\ or,\ \sigma]\ (a2\ \&\ in)\ =\ Z3\ [r\ \&\ or,\ \sigma]\ (in)$$

4.3.2.6 Timer

Wie in Abschnitt 2.3.4 erläutert, modellieren wir den Timermechanismus von SDL ohne Angabe von Ablaufzeitpunkten für die Timer – ein gesetzter Timer läuft nach beliebig langer, aber endlicher Zeit ab. Ein gesetzter Timer kann durch eine *Reset*-Anweisung deaktiviert werden und steht dem SDL-Prozeß somit nicht mehr als Eingabesignal zur Verfügung. Wird ein Timer mehrfach hintereinander gesetzt, so wird nur das Timersignal des zuletzt gesetzten Timers als Eingabesignal verarbeitet. Mittels des Ausdrucks *active* kann innerhalb eines Zustandsübergangs geprüft werden, ob ein Timer gesetzt ist.

In unserer Semantikdefinition haben wir bei der Modellierung der Komponente *Prozeß* eine Komponente *Timer* eingeführt, durch die die beliebige Verzögerung der Timersignale modelliert wird. Innerhalb der Komponente *PR* sind nun das Setzen und Rücksetzen von Timern sowie das Verarbeiten von Timersignalen zu definieren. Das Setzen eines Timers wird durch das Senden eines Timersignals an die Komponente *Timer* modelliert, die das Timersignal nach einer beliebig langen, jedoch endlichen Verzögerung an den Eingabepuffer *FairMerge* sendet. Ein Timersignal, das durch die Komponente *FairMerge* in den Eingabestrom der Komponente *PR* eingereiht wird, stellt nicht in allen Fällen ein gültiges Timersignal dar. Es kann, wie oben erläutert, durch das erneute Setzen oder durch das Rücksetzen des Timers ungültig geworden sein. In diesen Fällen wird das Signal in einer impliziten Transition vernichtet. Im Gegensatz zu SDL erfolgt das Löschen des Signals nicht, während der Timer abläuft (also in der *Timer*-Komponente) oder sich im Eingabepuffer des Prozesses befindet, sondern direkt in *PR*.

Für die Überprüfung, ob ein Timersignal zum Zeitpunkt der Verarbeitung in der Komponente *PR* ein gültiges Signal darstellt, führen wir zwei zusätzliche Zustandsparameter ein:

- Ein Zähler speichert die Differenz zwischen der Anzahl der gesendeten und der Anzahl der empfangenen Timersignale. Beim Setzen des Timers wird der Zähler um 1 erhöht, nach Verarbeiten eines Timersignals um 1 verringert. Nur wenn beim Empfang eines Timersignals der Wert des Zählers 1 ist, wurde der Timer nach Absenden dieses Timersignals nicht erneut gesetzt: es handelt sich um ein gültiges Timersignal. Ist der Wert des Zählers hingegen größer als 1, so wird das Timersignal in einer impliziten Transition vernichtet.
- Eine boolesche Variable gibt an, ob der Timer gesetzt und damit aktiv ist. Wird der Timer durch *Reset* zurückgesetzt, so erhält die Variable den Wert *false*; wird der Timer gesetzt, so erhält die Variable den Wert *true*. Der Ausdruck *active* wird durch die Abfrage des Werts dieser Variablen semantisch fundiert. Liegt ein gültiges Timersignal als Eingabenachricht für *PR* vor (obiger Zähler hat den Wert 1), so wird mittels der Variablen geprüft, ob der Timer noch aktiv oder mit *Reset* deaktiviert worden ist. Nach Verarbeiten eines gültigen Timersignals wird der Variablen der Wert *false* zugewiesen.

Für einen SDL-Prozeß, in dem der Timer *timeout* definiert ist, lautet die Funktionalität der Funktion *behaviour*:

$$behaviour \;:\; D \times Nat \times Bool \times (In \cup \{timeout\})^{\omega} \to (Out \cup \{timeout\})^{\omega} :$$
$$behaviour\ [\sigma_{init},\ 0,\ false]\ (in) \;=\; out$$

Das Timersignal *timeout* wird zur Menge der Ein- und Ausgabenachrichten von PR hinzugenommen. Der Timer ist zu Beginn der Prozeßinterpretation noch nicht gesetzt; der Zähler wird deshalb mit dem Wert 0, die boolesche Variable mit dem Wert *false* initialisiert.

Es ist zu beachten, daß auch in Zuständen, denen keine Übergänge mit Setzen und Rücksetzen von Timern folgen, Funktionsgleichungen für die Verarbeitung von Timernachrichten hinzuzufügen sind, da auch das implizite Verarbeiten einer Timernachricht den Zählerstand verringert.

Anhand des Beispiels in Abbildung 4.19 geben wir nun die Funktionsgleichungen für Zustandsübergänge an, in denen Timer gesetzt oder zurückgesetzt werden oder Timersignale als Eingabenachrichten vorliegen.

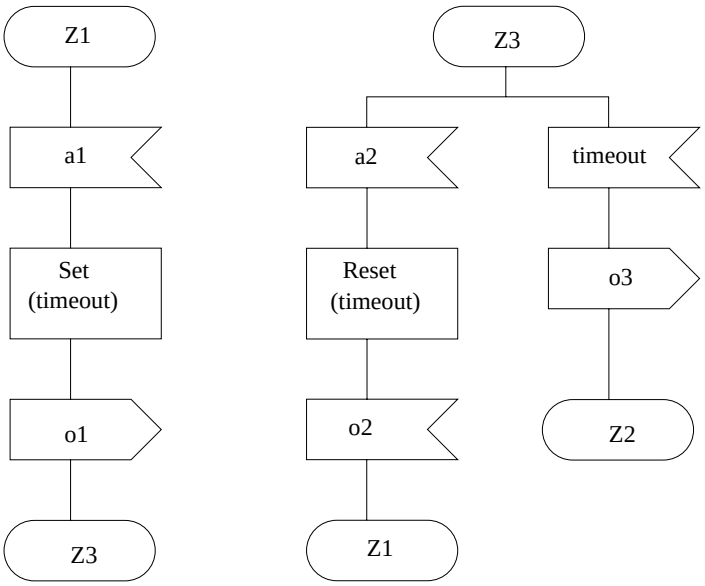


Abbildung 4.19: SDL-Prozeßausschnitt mit Timern

Timer setzen:

$$Z1\ [\sigma,\ c,\ b]\ (a1\ \&\ in) \;=\; timeout\ \&\ o1\ \&\ Z3\ [\sigma,\ c+1,\ true]\ (in)$$

Timer rücksetzen:

$$Z3\ [\sigma,\ c,\ b]\ (a2\ \&\ in) \;=\; o2\ \&\ Z1\ [\sigma,\ c,\ false]\ (in)$$

Timersignal *timeout* verarbeiten

$Z3\ [\sigma,\ c,\ b]\ (timeout\ \&\ in) =$
if $c > 1 \vee b = false$ — Timer erneut gesetzt oder zurückgesetzt
  then $Z3\ [\sigma,\ c-1,\ b]\ (in)$ — Implizite Transition
  else $o3\ \&\ Z2\ [\sigma,\ 0,\ false]\ (in)$ — Timersignal verarbeiten
endif

Jedem Zustand Z des Prozesses, in dem das Signal *timeout* nicht als Eingabenachricht spezifiziert ist, ist folgende Funktionsgleichung hinzuzufügen:

$Z\ [\sigma,\ c,\ b]\ (timeout\ \&\ in) =$
if $c > 1 \vee b = false$ — Timer erneut gesetzt oder zurückgesetzt
  then $Z\ [\sigma,\ c-1,\ b]\ (in)$ — ungültiges Timersignal - implizite Transition
  else $Z\ [\sigma,\ 0,\ false]\ (in)$ — gültiges Timersignal - implizite Transition
endif

Bisher haben wir den Fall betrachtet, daß nur ein Timer in der Prozeßspezifikation vorliegt. Werden in einem SDL-Prozeß mehrere Timer definiert, so müssen für jeden dieser Timer ein Zähler und eine boolesche Variable eingeführt werden. Es bietet sich an, ähnlich wie bei der Definition des Datenzustands, einen SPECTRUM-Datentypen T einzuführen, der sämtliche Variablen für alle Timer umfaßt.

Seien in einem SDL-Prozeß die Timer $t_1,\ \ldots,\ t_k$ definiert:

$$data\ \ T = mt\ (t_1 : (c : Nat, b : Bool),\ \ldots,\ t_k : (c : Nat, b : Bool))$$

Die Namen der Timer bilden die Selektoren, mt den Konstruktor der Sorte T. Ein Eintrag für einen Timer t_j ist von der Sorte $(Nat, Bool)$.

Die Funktionsgleichungen für mehrere Timer unterscheiden sich von denen für einen Timer nur darin, daß der Zugriff auf die einzelnen Zählerwerte mittels der Selektoren des Datentyps T erfolgt und die unterschiedlichen Timer getrennt zu behandeln sind. Deshalb verzichten wir darauf, auf diese Gleichungen näher einzugehen.

4.3.2.7 SDL-Prozeduren

Prozeduren stellen einen parametrisierten Teil einer SDL-Prozeßspezifikation dar, der über einen eigenen Gültigkeitsbereich für Zustandsnamen und lokale Variablen verfügt, jedoch auf die Variablen und den Eingabepuffer des aufrufenden Prozesses zugreifen kann. Für die semantische Fundierung ist zwischen Prozeduren mit und ohne Ergebniswert zu unterscheiden. Der Aufruf einer Prozedur ohne Ergebniswert erfolgt in einem eigenen Prozeßsymbol, die Ausführung des SDL-Prozesses wird gestoppt, bis die Prozedur beendet ist. Der Aufruf einer Prozedur mit Ergebniswert erfolgt hingegen als Teil eines Ausdrucks innerhalb einer Anweisung in einem Tasksymbol.

Prozeduren ohne Ergebniswert:

Für die semantische Fundierung von Prozeduren ohne Ergebniswert gilt: Vor der Erstellung der Semantikdefinition des Prozesses wird der Prozedurgraph anstelle des Prozeduraufrufs in den Prozeßgraphen einkopiert. Dabei werden Prozeduranfangs- und Prozedurendesymbol entfernt. Verfügt die Prozedur über lokale Variablen, so werden diese zu den lokalen Variablen des Prozesses hinzugefügt. Anschließend werden die Funktionsgleichungen für die Zustandsübergänge wie bisher beschrieben erstellt.

Prozeduren mit Ergebniswert:

Prozeduren mit Ergebnis sind in SDL 92 eingeführt worden. Ihr Aufruf erfolgt in den Tasksymbolen; lokale Prozeßvariablen können als aktuelle Parameter dienen. Da Prozeduren mit Ergebniswert überwiegend verwendet werden, um komplexe Berechnungen auf dem Datenzustand des Prozesses durchzuführen, und damit keine Signale verarbeiten, beschränken wir uns auf Prozeduren ohne Zustände, d. h. auf Prozeduren mit nur einer Transition. Die Prozeduren entsprechen im mathematischen Sinne Funktionen, die Eingabewerte erhalten und ein Ergebnis liefern. Allerdings werden diese Funktionen über den Datentypen der lokalen Variablen des SDL-Prozesses definiert und nicht wie bei der bisher erfolgten Semantikdefinition über den Eingabeströmen des SDL-Prozesses. Um SDL-Prozeduren formal zu fundieren, werden wir aus einer SDL-Prozedur eine Funktion (Rechenvorschrift) ableiten. Unser Vorgehen erklären wir am Beispiel der SDL-Prozedur in Abbildung 4.20.

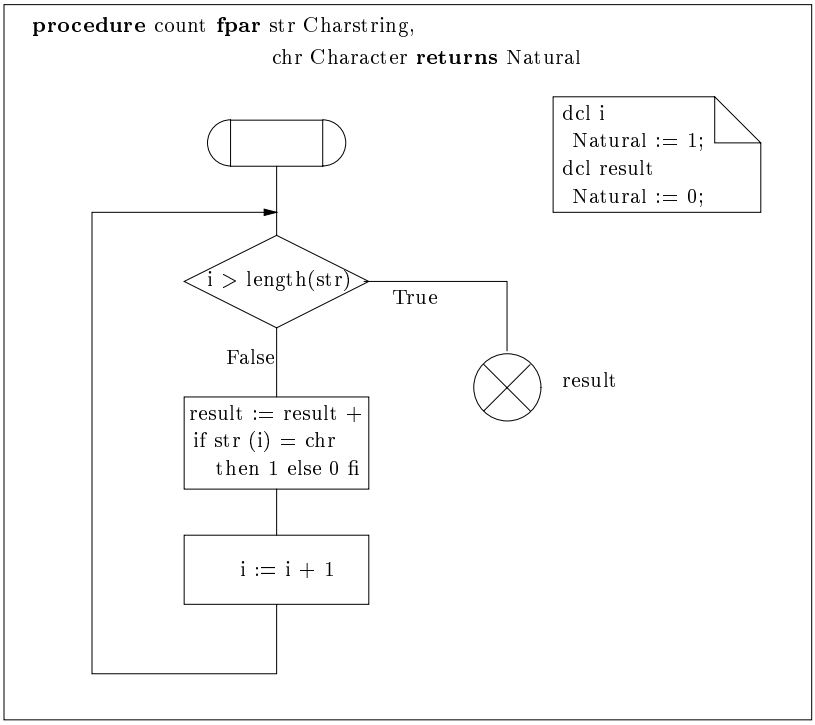


Abbildung 4.20: SDL-Prozedur mit Ergebniswert

Aus den Prozedurdiagrammen kann anhand der Sortenangaben der formalen Parameter und des Ergebnisses die Funktionalität der Funktion definiert werden. Dabei werden die SDL-Datentypen auf die entsprechenden Datentypen von SPECTRUM abgebildet, wie in Abschnitt 4.2 erklärt. Die Funktion erhält den Namen der SDL-Prozedur *proc*. Für eine Funktionsdeklaration schreiben wir:

$$\mathsf{function}\ proc = (x_1 : adt_1,\ \ldots, x_n : adt_n)\ adt_{n+1} : E$$

Dabei sind $x_1, \ldots, x_n$ mit den Sorten $adt_1, \ldots, adt_n$ die Argumente der Funktion *proc*. Die Funktion liefert ein Ergebnis der Sorte adt_{n+1}. E ist ein Ausdruck der Sorte adt_{n+1} und stellt den Rumpf der Funktion dar.

In der SDL-Prozedur *count* aus Abbildung 4.20 werden neben den formalen Parametern lokale Variablen verwendet. Wir setzen diese Variablen ebenfalls in formale Parameter um. Die Funktionsdeklaration lautet somit:

$$\textsf{function } \mathit{count} = (\mathit{String}\ \mathit{str},\ \mathit{Char}\ c,\ \mathit{Nat}\ i,\ \mathit{Nat}\ \mathit{result})\ \mathit{Nat}$$

Dabei sind *String*, *Char* und *Nat* abstrakte Datentypen aus SPECTRUM.

Beim Aufruf der Funktion *count* innerhalb eines Ausdrucks in einem Tasksymbol werden den formalen Parametern *str* und *c* die beim Aufruf angegeben Werte zugewiesen. Die Parameter *i* und *result* werden mit den Werten 1 und 0 belegt. Diese Belegung ergibt sich aus der Initialisierung der Variablen *i* und *result* im Textsymbol des Prozedurdiagramms.

Der Graph der SDL-Prozedur wird in den Rumpf der Funktion umgesetzt. Dabei entspricht die Kante, die von dem Tasksymbol mit der Anweisung $i := i + 1$ zurück an den Beginn des Graphen führt, einem rekursiven Aufruf der Funktion *count*.

$$\begin{array}{l}\textsf{if } i > \mathit{length}(\mathit{str}) \textsf{ then } \mathit{result}\\ \textsf{elif } \mathit{str}(i) = \mathit{chr} \textsf{ then } \mathit{count}\ (\mathit{str},\ c,\ i+1,\ \mathit{result}+1)\\ \qquad\qquad\qquad\quad \textsf{else } \mathit{count}\ (\mathit{str},\ c,\ i+1,\ \mathit{result})\\ \textsf{endif}\end{array}$$

Die Funktion *length* und der Zugriff auf das i-te Element eines Strings $\mathit{str}(i)$ sind entweder Bestandteile des Datentyps *String* oder an anderer Stelle definiert worden.

Operator Diagrams:

In der Definition von abstrakten Datentypen in SDL kann das Verhalten von Funktionen anstelle durch Gleichungen auch graphisch spezifiziert werden. Dazu sind in SDL 92 Operator Diagrams eingeführt worden, die dem Sprachkonstrukt der Prozeduren mit Ergebniswert sehr ähnlich sind. Operator Diagrams dürfen nur eine Transition enthalten und verfügen möglicherweise über lokale Variablen. Da wir uns bei der formalen Umsetzung der SDL-Prozeduren auf Prozeduren ohne Zustände beschränkt haben, können Operator Diagrams in der gleichen Weise wie SDL-Prozeduren mit Ergebniswert in eine Funktion umgesetzt werden.

4.4 Abschließende Bemerkung

In diesem Kapitel wurde eine formale Semantik für SDL definiert. Dazu haben wir beschrieben, wie sich eine SDL-Spezifikation in eine adäquate FOCUS-Systembeschreibung umsetzen läßt, die das gleiche Verhalten wie das mit SDL spezifizierte System aufweist.

Implizite Konzepte und Mechanismen aus SDL, wie z.B. der Zeitbegriff, die Eingabepuffer von Prozessen oder die Verzögerung bei der Signalübertragung, sind in der Semantikdefinition explizit modelliert und nun greifbar. Die Semantikdefinition wurde bewußt modular aufgebaut; gerade für die Semantik eines SDL-Prozesses liegt nun eine Reihe von Semantikbausteinen vor, die in sich abgeschlossene Verhaltensanteile beschreiben und leicht zu verstehen sind.

Es ist deutlich geworden, daß sich ein einfacher SDL-Prozeß, der einem Zustandsautomaten mit Ein- und Ausgabe und Datenanteil entspricht, schematisch in eine Menge einfacher Funktionsgleichungen umsetzen läßt. Die Funktionsgleichungen werden jedoch komplexer, wenn die Zustandsübergänge Sprachkonstrukte enthalten, die das Konzept des Zustandsautomaten erweitern, wie z.B. Savesignale, Setzen und Rücksetzen von Timern oder Konstrukte für Nichtdeterminismus. In diesen Fällen sind zusätzliche Parameter und Hilfsfunktionen erforderlich, um das Verhalten adäquat modellieren zu können. Dadurch werden die Funktionsgleichungen wesentlich komplexer. Die Komplexität steigt zusätzlich, wenn die Länge der Zustandsübergänge zunimmt. Die Komplexität einer SDL-Spezifikation spiegelt sich somit in der zugehörigen Semantikdefinition wider.

Gerade die semantische Umsetzung des Timerkonzepts hat zu komplexen Funktionsgleichungen geführt. Wir haben dabei das mehrfache Setzen und Rücksetzen von Timern berücksichtigt, da diese ein wesentliches Modellierungskonzept von SDL sind. Es stellt sich jedoch die Frage, welchen Vorteil eine explizite Spezifikation von mehrfachem Setzen von Timern und dem Rücksetzen mit sich bringt. Es handelt sich hier um einen Mechanismus, der sehr stark an einer operationellen Vorstellungsweise orientiert ist. Auf abstrakterer Spezifikationsebene würde wohl auch das alleinige Setzen von Timern genügen ohne die Möglichkeit, Timer rückzusetzen. Das Erkennen von Timersignalen, die nicht mehr aktuell sind, könnte stattdessen durch die Einführung unterschiedlicher Zustände und die Verwendung von impliziten Zustandsübergängen spezifiziert werden.

Wir haben die Semantikdefinition für SDL auf der Spezifikationsebene von FOCUS durchgeführt und die Beschreibungssprache ANDL verwendet, um eine verständliche Darstellung der Semantikdefinition zu erhalten. Diese Vorgehensweise haben wir bewußt gewählt, da damit das semantische Modell von FOCUS mit den mathematisch-logischen Konzepten verborgen bleibt und auch Anwender, die mit mathematisch-logischen Formalismen weniger vertraut sind, die Umsetzung von SDL nach FOCUS und damit die Semantikdefinition nachvollziehen können.
Im semantischen Modell von FOCUS entspricht jede Basiskomponente und jedes Netzwerk einem Prädikat, das all diejenigen Funktionen definiert, die ein zulässiges Verhalten der Komponente bzw. des Netzwerks beschreiben. Dadurch ordnet die SDL-Semantik jeder SDL-Spezifikation durch die Überführung in eine FOCUS-Spezifikation eine Menge von

pulsgetriebenen Funktionen zu. Die Menge von möglichen Verhalten ist bedingt durch das nichtdeterministische Verhalten von SDL-Systemen. Grund dafür ist in erster Linie das Zeitkonzept von SDL, das einem SDL-Prozeß ein beliebiges zeitliches Verhalten zuordnet. Daneben tragen Kanäle mit Verzögerung, das Interleavingkonzept bei den Eingabepuffern der SDL-Prozesse, spontane Übergänge sowie nichtdeterministische Entscheidungen zum nichtdeterministischen Verhalten von SDL-Systemen bei. Somit führt ein Eingabestrom eines SDL-Systems zu einer Menge von möglichen Ausgabeströmen. Auf die Betrachtung der mathematisch-logischen Ebene der SDL-Semantikdefinition werden wir bei der Verifikation von SDL-Spezifikationen in Kapitel 6 zurückkommen.

Um dem Anwender den Umgang mit der formalen Semantik zu erleichtern, ist eine werkzeugunterstützte Erstellung der formalen Semantik für SDL-Spezifikationen wünschenswert. Zu diesem Zweck haben wir unter UNIX mit Hilfe der GNU-Compilerwerkzeuge flex und bison einen Prototypen implementiert ([Hin96b]). Dieser setzt auf der textuellen PR-Darstellung von SDL-Spezifikationen auf und generiert für diese die zugehörigen FOCUS-Spezifikationen. Die textuelle Darstellung von SDL-Spezifikationen kann bei vielen SDL-Werkzeugen aus der graphischen Darstellung generiert werden. Mit diesem Prototypen ist die Machbarkeit einer automatischen Überführung von SDL-Spezifikationen in die zugehörige FOCUS-Systemspezifikation nachgewiesen.

Mit der von uns eingeführten Semantikdefinition in FOCUS erfüllt SDL nun die Anforderungen einer formalen Spezifikationssprache. Eigenschaften eines mit SDL spezifizierten Systems können basierend auf dieser mathematisch-logischen Semantikdefinition verifiziert werden. Am Beispiel des Alternating Bit Protokolls werden wir dies in Kapitel 6 demonstrieren und eine Verifikationsmethode entwickeln.

Kapitel 5

Formale Fundierung der dynamischen Prozeßerzeugung

In der formalen Semantikdefinition von SDL (siehe Kapitel 4) haben wir die meist verwendeten Sprachkonstrukte von SDL mit Ausnahme der dynamischen Prozeßerzeugung behandelt. Durch das Erzeugen von SDL-Prozessen mittels des Sprachkonstrukts *Create* wächst die Anzahl der vorhandenen Prozesse. Ferner entstehen zusätzliche Kommunikationsverbindungen zwischen den Prozessen. Die Struktur eines SDL-Systems ist somit nicht länger statisch, sondern unterliegt einer dynamischen Veränderung. Eine Modellierung dieses dynamischen Verhaltens ist im klassischen FOCUS ([BDD+93]) nicht unmittelbar möglich.

Das semantische Modell des klassischen FOCUS ist für statische verteilte Systeme ausgerichtet. Damit zielt es auf die Modellierung von Systemen ab, deren Komponenten feste Schnittstellen haben, deren Netzwerkstruktur während des Systemablaufs nicht verändert wird und in denen während des Systemablaufs keine Komponenten erzeugt bzw. gelöscht werden können. Ist die Zahl der gleichzeitig existierenden Prozeßinstanzen beschränkt, so kann eine Umsetzung des *Create*-Konstrukts bereits im Rahmen der in Kapitel 4 erfolgten Semantikdefinition durchgeführt werden, indem das dynamische Erzeugen durch das Aktivieren bereits vorhandener Prozeßinstanzen modelliert wird. Dabei sind allerdings die Zahl der Prozeßinstanzen und die Kommunikationsverbindungen während des gesamten Systemablaufs statisch. Dies widerspricht der Idee von SDL, Prozeßinstanzen während des Systemablaufs dynamisch zu kreieren und zu löschen.

Mit [GS97] liegt mittlerweile eine semantische Erweiterung von FOCUS für die Spezifikation mobiler, dynamischer Systeme vor. Diese ermöglicht es, Systeme zu spezifizieren, deren Kommunikationsstruktur sich während des Systemablaufs verändert und in denen neue Komponenten dynamisch kreiert und in die bestehende Systemstruktur integriert werden können. Unter Verwendung dieser FOCUS-Variante werden wir die Semantik für das *Create*-Konstrukt definieren. Wir beschreiben, wie sich die Prozeßerzeugung in der FOCUS-Erweiterung für mobile, dynamische Systeme semantisch fundieren läßt. Dabei wird deutlich, welche komplexen Eingriffe in die System- und Kommunikationsstruktur mit einer SDL-Prozeßerzeugung verbunden sind. Um unsere Semantikdefinition an einem Beispiel vorzustellen, geben wir für die SDL-Spezifikation des *Daemon Game* in Anhang A die for-

male Semantik an. Das *Daemon Game* ist ein SDL-Standardbeispiel für die dynamische Prozeßerzeugung

Die Erweiterung des semantischen Modells von FOCUS um mobile und dynamische Konzepte ist in [GS97] beschrieben; in [HS96] geben wir eine für den Anwender verständliche Einführung in die semantischen Grundlagen dieser Erweiterung und Leitfäden für die Spezifikation von dynamischen, mobilen Systemen. Wir verzichten deshalb im folgenden auf eine ausführliche Vorstellung der semantischen Basis von mobilen, dynamischen Systemen.

Das Kapitel gliedert sich wie folgt: In Abschnitt 5.1 erläutern wir die Erweiterung des klassischen FOCUS-Ansatzes, soweit dies für das Verständnis der anschließenden Semantikdefinition erforderlich ist. Darauf aufbauend definieren wir in Abschnitt 5.2 die formale Semantik für das SDL-Sprachkonstrukt *Create*. In Anhang A stellen wir die SDL-Spezifikation des Daemon Game vor und geben die formale Semantik für diese Spezifikation an. Dabei gehen wir ausführlich auf die dynamische Veränderung der Struktur des SDL-Systems ein, die durch die Prozeßerzeugung hervorgerufen wird.

5.1 Mobile, dynamische Focus-Systeme

Die Erweiterung des klassischen FOCUS umfaßt zwei Modellierungskonzepte: Mobilität und Dynamik. Unter Mobilität verstehen wir, daß eine Komponente aufgrund von Interaktionen und Berechnungen ihre aktuelle Kanalschnittstelle verändern kann und die Kommunikationsverbindungen zwischen den Komponenten somit variabel sind. Unter dem Begriff Dynamik verstehen wir, daß eine Komponente während des Systemablaufs weitere Komponenten erzeugen und in die Systemstruktur einbinden kann sowie daß sich Komponenten während des Systemablaufs beenden können.

Wie im klassischen FOCUS werden auch im erweiterten FOCUS Systeme als Netzwerke von Komponenten beschrieben, die durch Nachrichtenaustausch kommunizieren. Die Komponenten sind durch eine Menge gerichteter Kanäle fest verbunden, wobei grundsätzlich jede Komponente jeden Kanal als Ein- oder Ausgabekanal verwenden kann. Auf welche Kanäle eine Komponente tatsächlich lesend oder schreibend zugreift, wird durch die Vergabe von Lese- und Schreibrechten für die Kanäle geregelt. Besitzt eine Komponente das Leserecht zu einem Kanal, so gehört dieser Kanal zu ihren Eingabekanälen, und sie darf von diesem Kanal Nachrichten lesen. Verfügt eine Komponente über das Schreibrecht zu einem Kanal, so gehört dieser Kanal zu ihren Ausgabekanälen, und sie darf über ihn Nachrichten senden. Komponenten, die ihre aktuelle Schnittstelle, also ihre Ein- und Ausgabekanäle verändern, heißen mobile Komponenten. Die Mobilität der Komponenten ist der wesentliche Unterschied zum klassischen FOCUS-Ansatz.

Zu Beginn verfügt jede Komponente über eine initiale Schnittstelle, die aus einer Menge I von Eingabe- und einer Menge O von Ausgabekanälen besteht. Jeder Komponente steht zusätzlich eine Menge von privaten Kanalnamen P für das Erzeugen neuer Kommunikationsverbindungen zur Verfügung. Während des Systemablaufs wächst bzw. verringert sich die Menge der verfügbaren Ein- und Ausgabekanäle der einzelnen Komponenten, in-

dem sich die Komponenten sogenannte Ports schicken und somit neue Rechte an Kanälen erlangen oder vorhandene Rechte abtreten. Ein Port ist ein Kanalname, der zusätzlich mit einem Zugriffsrecht auf den Kanal (Lesen („?“) oder Schreiben („!“)) versehen ist. Neue Kommunikationsverbindungen zwischen Komponenten können nur erzeugt werden, wenn die Komponenten bereits direkt verbunden sind oder – falls keine direkte Verbindung vorliegt – eine indirekte Kommunikationsverbindung über andere Komponenten existiert. Andernfalls kann zwischen den beiden Komponenten kein Port übermittelt werden. Die Kommunikation zwischen den Komponenten erfolgt nach dem many-to-many Prinzip, d.h. mehrere Komponenten dürfen gleichzeitig lesend oder schreibend auf einen Kanal zugreifen.

Wenn die Komponente einen Port $!j$ als Nachricht an eine andere Komponente verschickt, wobei sie den Kanal j selbst erzeugt hat ($j \in P$), so darf sie später den Inhalt des Kanals j lesen. Schickt sie einen Port $?p$, wobei sie den Kanal p selbst erzeugt hat, so darf sie Nachrichten auf diesem Kanal senden. Erhält die Komponente den Port $!q$, so darf sie anschließend auf den Kanal q schreiben; erhält sie den Port $?q$, so darf sie von Kanal q lesen. Durch das semantische Modell wird sichergestellt, daß eine Komponente nur auf diejenigen Kanäle lesend oder schreibend zugreift, zu denen sie aktuell die entsprechenden Ports besitzt. Jede Komponente erhält als Eingabe die Nachrichtenströme aller innerhalb des Systems vorhandenen Kanäle. Allerdings greift die Komponente nur auf diejenigen Kanäle lesend bzw. schreibend zu, für die sie die Zugriffsrechte besitzt.

Um die dynamische Veränderung der Schnittstelle einer mobilen Komponente zu erfassen, werden die Mengen ap und pp eingeführt. Die Menge ap (active ports) enthält die Ports der Kanäle, auf die die Komponente aktuell lesend oder schreibend zugreifen darf. Die Menge pp (private ports) enthält die Menge aller privaten Ports; das sind diejenigen Ports, die nur der Komponente selbst bekannt sind und die sie für das Erzeugen neuer Kommunikationsverbindungen verwenden kann. Im semantischen Modell wird das Verhalten einer mobilen Komponente durch eine Menge von mobilen Funktionen beschrieben. Diese bilden wie stromverarbeitende Funktionen Eingabe- auf Ausgabeströme ab und gewährleisten dabei, daß sich die Mengen ap und pp verändern, sobald die Komponente Ports empfängt oder versendet.

In [Gro96a] wird das Modell für mobile Systeme um die dynamische Erzeugung von Komponenten erweitert. Damit kann eine Komponente neue Komponenten erzeugen und in die bestehende Systemstruktur einfügen. Dazu wird der Komponente bei ihrer Erzeugung eine Menge von Ports übergeben, über die sie Kommunikationsverbindungen zu den vorhandenen Komponenten aufbaut. Auf semantischer Ebene entspricht das Erzeugen einer neuen Komponente während des Systemablaufs einem Verfeinerungsschritt.

Die Spezifikation von Netzwerken mobiler Komponenten mit many-to-many Kommunikation erfolgt nicht durch die Angabe von Netzwerkgleichungen wie in ANDL[1], sondern durch die Verwendung eines Kompositionsoperators $\oplus$. Dieser Operator verbindet Komponenten durch parallele oder sequentielle Komposition sowie durch Rückkopplung zu Netzwerken. Er wird sowohl auf Komponenten- als auch auf Funktionsebene verwendet.

[1]Eine Beschreibung der Netzstruktur mit ANDL ist nicht möglich, da ANDL ausschließlich die Spezifikation von Systemen mit point-to-point Kommunikation unterstützt.

5.2 Semantikdefinition von Create

Ziel dieses Kapitels ist die Semantikdefinition für die dynamische Prozeßgenerierung in SDL. Wir konzentrieren uns deshalb im folgenden auf das Sprachkonstrukt *Create*. Die Semantik der übrigen SDL-Sprachkonstrukte kann ohne Probleme aus der Semantikdefinition für statische SDL-Systeme übernommen und an das erweiterte semantische FOCUS-Modell angepaßt werden. Für die SDL-Spezifikation *Daemon Game* geben wir die formale Semantikdefinition in Anhang A vollständig an, so daß deutlich wird, wie die übrigen SDL-Konstrukte in der Semantikdefinition für dynamische SDL-Systeme behandelt werden.

5.2.1 Dynamische Prozeßerzeugung in SDL

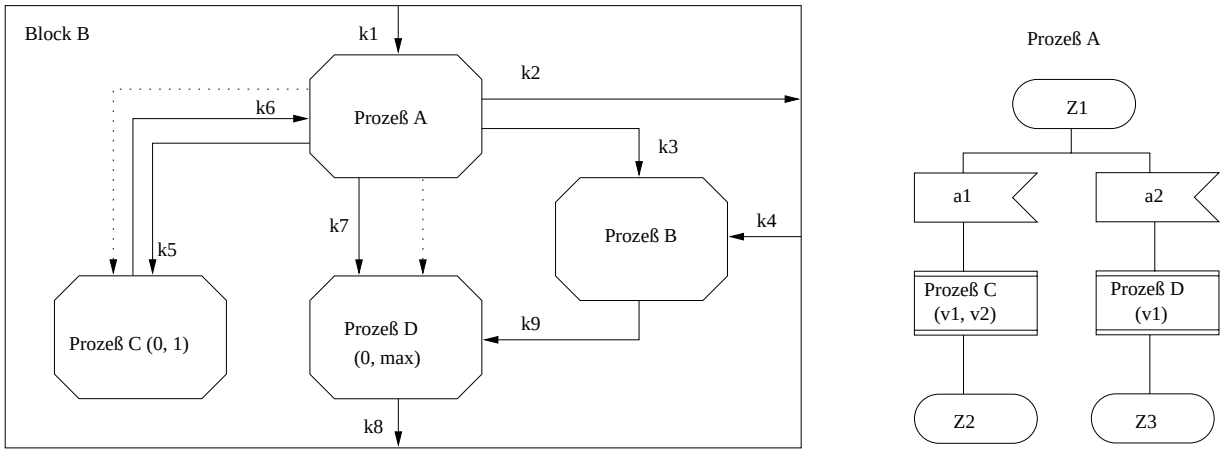


Abbildung 5.1: SDL-Prozeßausschnitt mit Prozeßerzeugung

Das linke Diagramm in Abbildung 5.1 zeigt einen SDL-Block mit vier Prozessen. Die SDL-Prozesse `C` und `D` werden dynamisch von SDL-Prozeß `A` erzeugt (gekennzeichnet durch die gestrichelte Linie) und sind anschließend über Signalwege mit dem SDL-Prozeß `A` bzw. mit den SDL-Prozessen `A` und `B` sowie der Blockumgebung verbunden. Die Angaben in Klammern, die in den Prozeßsymbolen von `C` und `D` enthalten sind, legen fest, wieviele Instanzen des jeweiligen SDL-Prozesses zu Systembeginn und wieviele Instanzen während des Systemablaufs maximal gleichzeitig existieren dürfen. Von Prozeß `C` existiert beim Systemstart keine, während des Systemablaufs höchstens eine Instanz. Von Prozeß `D` ist beim Systemstart ebenfalls keine Instanz vorhanden; während des Systemablaufs kann eine Menge von höchstens *max* Prozeßinstanzen erzeugt werden. Wird keine Obergrenze angegeben, so ist die Anzahl der gleichzeitig existierenden Instanzen eines Prozesses nicht nach oben beschränkt; während des Systemablaufs kann eine beliebig große Menge von Prozeßinstanzen entstehen. In SDL werden die Instanzen einer Prozeßmenge durch eindeutige Prozeßidentifikatoren unterschieden. Die Adressierung eines Signals an eine Prozeßinstanz erfolgt über die Angabe ihres Prozeßidentifikators. Die Signalwege, die einer Menge von Instanzen als Ein- bzw. Ausgabesignalwege zugeordnet sind (z.B. die Signalwege `k7, k8` und `k9` für die Menge von Prozeßinstanzen `D`), legen die Ein- und Ausgabesignale fest, die

die Prozeßinstanzen empfangen bzw. senden dürfen. Es bestehen keine Signalwege zwischen jeder einzelnen Prozeßinstanz `D` und den Prozessen `A` und `B` sowie der Blockumgebung. Ebenfalls in Abbildung 5.1 ist der entsprechende Zustandsübergang innerhalb des SDL-Prozesses `A` abgebildet, in dem mittels des *Create*-Konstrukts Prozeßinstanzen der SDL-Prozesse `C` und `D` erzeugt werden.

Wird in einem SDL-Prozeß ein Zustandsübergang mit Prozeßerzeugung ausgeführt, wenn die maximale Anzahl der gleichzeitig vorhandenen Prozeßinstanzen bereits erreicht ist, so erfolgt der Zustandsübergang ohne Prozeßerzeugung. Erst wenn sich eine Instanz durch Ausführung eines Stopsymbols beendet hat, kann eine neue Instanz kreiert werden.

In SDL wird keine Aussage getroffen, ob und welche Komponente des Systems Information über die Zahl der aktuell existierenden Instanzen eines Prozesses sammelt und wie die SDL-Prozesse bei der Erzeugung einer neuen Instanz auf diese Information zugreifen können. Es wird auch nicht angegeben, auf welche Weise die eindeutigen Prozeßidentifikatoren vergeben werden. In der Semantikdefinition in FOCUS werden wir diese impliziten Konzepte modellieren.

5.2.2 Aufbau einer Prozeßkomponente in Focus

Bei der formalen Fundierung einer Prozeßkomponente gehen wir ähnlich wie bei der Semantikdefinition für statische SDL-Systeme vor. Das Verhalten eines SDL-Prozesses wird durch das Zusammenwirken mehrerer Komponenten in einem Netzwerk *Prozeß* erbracht (vergleiche Abschnitt 4.3.1). Im Unterschied zur Semantikdefinition für statische SDL-Systeme sind jedoch keine Misch- und Verteilkomponenten mehr enthalten, da im semantischen Modell für mobile, dynamische FOCUS-Spezifikationen das Kommunikationsprinzip many-to-many verwendet wird. Bei dieser Art der Kommunikation wird die Interferenz, die auftritt, wenn mehrere Komponenten auf einen Kanal gleichzeitig schreiben, im semantischen Modell aufgelöst. Dies wird erreicht, indem in den Kompositionsoperator $\oplus$ implizite Mischkomponenten eingebaut sind. Diese mischen die Ströme ohne zeitliche Verzögerung und mit beliebigem Interleaving der Signale, die innerhalb eines Zeitintervalls gesendet werden. Für unsere Umsetzung eines SDL-Prozesses bedeutet dies, daß sämtliche Signalwege, die einen SDL-Prozeß erreichen, nun in einen Kanal münden, wobei dieser Kanal den Eingabepuffer des Prozesses darstellt und Eingabekanal der Komponente *PR* ist. Die explizite Spezifikation von Verteilkomponenten der Art *Split* erübrigt sich ebenfalls, da ein Ausgabesignal der Komponente *PR* einem bestimmten Kanal direkt zugeordnet werden kann. Es ergibt sich der in Abbildung 5.2 dargestellte Aufbau der Komponente *Prozeß*.

Die Spezifikation der Komponenten *None* und *Timer* erfolgt wie bei der statischen Semantikdefinition und kann von dort übernommen werden. Innerhalb der Komponente *PR* verwenden wir Funktionsgleichungen, die wir aus den Zustandsübergängen des SDL-Prozesses ableiten. Für Zustandsübergänge, in denen keine Prozeßerzeugung durchgeführt wird, erfolgt die Ableitung wie bei der statischen Semantikdefinition. Es ist aber zu beachten, daß aufgrund der veränderten Kommunikationsbehandlung explizit auf die Kanalnamen zugegriffen wird.

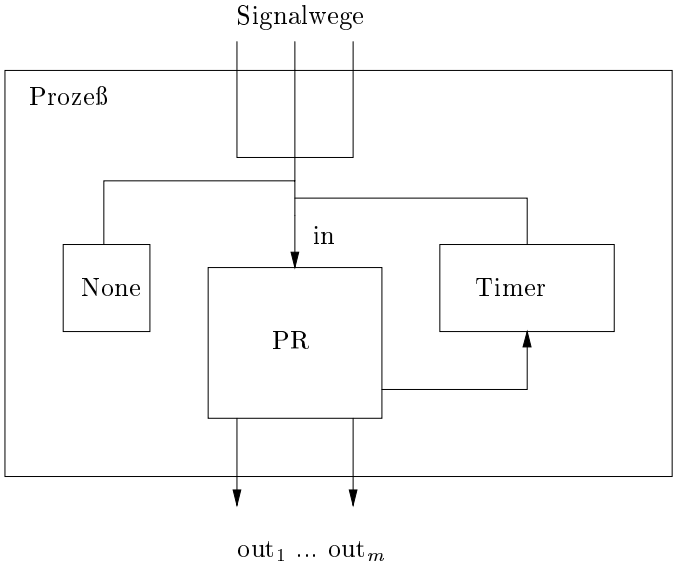


Abbildung 5.2: Aufbau einer Komponente *Prozeß*

Eine Funktionsgleichung genügt im wesentlichen folgendem Schema:

$$Z\ [\sigma]\ (\{in \mapsto a\}\ \&\ s) = \{out_1 \mapsto b_1,\ out_2 \mapsto b_2,\ \ldots,\ out_m \mapsto b_m\}\ \&\ Z'\ [\sigma']\ (s)$$

Die Funktion Z modelliert das Verhalten des SDL-Prozesses im Zustand Z. Sie erhält die Nachricht a auf dem Eingabekanal in, reagiert darauf mit der Ausgabe der Nachrichten $b_1, \ldots, b_m$ über die Kanäle $out_1, \ldots, out_m$ und ruft für die Abarbeitung des restlichen Eingabestroms s die Funktion Z' auf, die das Verhalten des SDL-Prozesses im Zustand Z' modelliert. Zusätzlich verfügt die Funktion Z über einen Zustandsparameter σ, der die aktuellen Belegungen der lokalen Prozeßvariablen enthält und durch die Verarbeitung der Eingabenachricht a in σ' übergeht.

5.2.3 Modellierung der SDL-Prozeßerzeugung in Focus

Wir geben in diesem Abschnitt zunächst die Semantikdefinition für eine SDL-Prozeßerzeugung an, bei der genau eine Instanz kreiert wird, und erklären daran unsere Vorgehensweise. Anschließend behandeln wir die Erzeugung einer Menge von Prozeßinstanzen. Wir setzen voraus, daß die Semantikdefinition der SDL-Prozesse `B`, `C` und `D` in Form von FOCUS-Komponenten *Prozeß B*, *Prozeß C* und *Prozeß D* vorliegt, und geben im folgenden die Funktionsgleichungen an, die für die Prozeßerzeugung innerhalb des SDL-Prozesses `A` aus Abbildung 5.1 relevant sind.

Abbildung 5.3 zeigt die Netzstruktur in FOCUS für den SDL-Ausschnitt aus Abbildung 5.1 nach der Erzeugung der Komponente *Prozeß C* sowie von zwei Komponenten der Art *Prozeß D*. Dabei ist zu beachten, daß aufgrund der Kommunikationsform many-to-many mehrere Komponenten auf einen Kanal schreiben. Dadurch brauchen nicht alle Kanäle und Signalwege aus der SDL-Spezifikation nach FOCUS umgesetzt werden. So werden zum

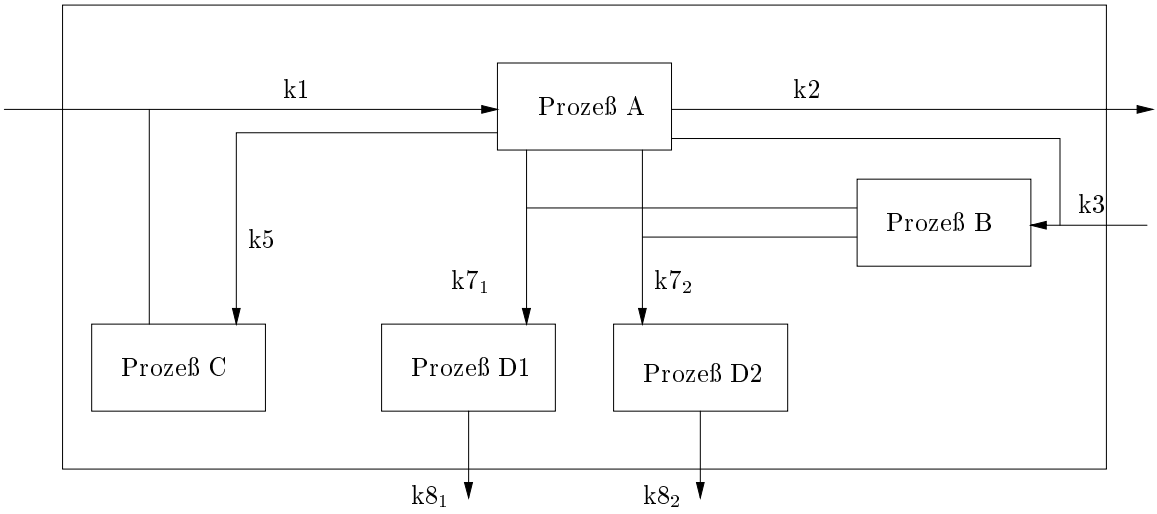


Abbildung 5.3: FOCUS-Netz für den SDL-Ausschnitt aus Abbildung 5.1

Beispiel die Eingabesignalwege `k3` und `k4` des SDL-Prozesses `B` durch den FOCUS-Kanal $k3$ modelliert.

5.2.3.1 Dynamische Erzeugung einer einzelnen Prozeßinstanz

Der SDL-Prozeß `A` erzeugt während des Zustandsübergangs von `Z1` nach `Z2` eine Instanz des SDL-Prozesses `C`. Danach sind die beiden SDL-Prozesse `A` und `C` durch die Signalwege `k5` und `k6` verbunden (siehe Abb. 5.1). Das dazugehörige FOCUS-Netzwerk (siehe Abb. 5.3) zeigt die beiden Komponenten *Prozeß A* und *Prozeß C* und ihre Kanalverbindung durch $k1$ und $k5$. Die Prozeßerzeugung wird in der Semantikdefinition für den SDL-Prozeß `A` durch folgende Funktionsgleichung erfaßt:

$$
\begin{array}{l}
\exists\ \mathit{fprozeß\_C} \in [\![\ \mathit{Prozeß\_C}\ ]\!] : \\
Z1\ [\sigma,\ c]\ (\{k1 \mapsto a1\}\ \&\ s)\ = \\
\mathsf{if}\ c = 0 \\
\quad \mathsf{then} \left(\underbrace{Z2\ [\sigma,\ 1]}_{Verhalten\ von\ A} \ \overline{\oplus}\ \underbrace{\mathit{fprozeß\_C}\ (\{?k5,\ !k1\},\ v1(\sigma),\ v2(\sigma))}_{Verhalten\ von\ C} \right)\ (s) \\
\quad \mathsf{else}\ Z2\ [\sigma,\ c]\ (s) \\
\mathsf{endif}
\end{array}
$$

- Die Existenz einer Funktion, die ein zulässiges Verhalten der zu erzeugenden Komponente *Prozeß_C* darstellt, wird in der Gleichung durch die Forderung $\exists\ \mathit{fprozeß\_C} \in [\![\ \mathit{Prozeß\_C}]\!]$ sichergestellt. $[\![\ \mathit{Prozeß\_C}\ ]\!]$ bezeichnet die Menge aller mobilen Funktionen, die das Verhalten von *Prozeß_C* beschreiben.

- Der Parameter c der Funktion $Z1$ dient als Zähler und wird mit 0 initialisiert. Er stellt sicher, daß maximal eine Instanz des Prozesses C erzeugt wird. Vor jeder Ausführung des *Create*-Konstrukts wird überprüft, ob aktuell bereits eine Prozeßinstanz des Prozesses C existiert und c somit den Wert 1 hat. Ist dies der Fall, so wird der Zustandsübergang ohne Prozeßerzeugung durchgeführt. Liefert der Zähler dagegen den Wert 0, so wird die Prozeßerzeugung durchgeführt; der Zähler wird auf 1 gesetzt.
- In der Komponente *Prozeß A* trifft auf Kanal $k1$ die Nachricht $a1$ ein, die den Zustandsübergang mit der Prozeßerzeugung von C initiiert.
- Das Erzeugen der Komponente *Prozeß C*, die in der Semantikdefinition der Prozeßinstanz `C` entspricht, wird durch den Aufruf der Funktion *fprozeß_C* modelliert.
- Die neu erzeugte Komponente *Prozeß C* wird über den Kompositionsoperator $\overline{\oplus}$ in die bestehende Systemstruktur eingebunden[2].
- Bei der Erzeugung von *Prozeß C* wird die Schnittstelle initialisiert. Dazu erhält die Funktion *fprozeß_C* die Ports $?k5$ und $!k1$ als Parameter, welche aus der Menge pp der erzeugenden Komponente *Prozeß A* stammen. Mit diesen Ports hat *Prozeß C* Lesezugriff auf die neu geschaffene Kanalverbindung $k5$ und Schreibzugriff auf die bereits vorhandene Kanalverbindung $k1$. Die komplementären Kanalrechte, also die Ports $!k5$ und $?k1$, besitzt die Komponente *Prozeß A*.
- Als weitere Parameter erhält die Funktion *fprozeß_C* die aktuellen Werte der Variablen $v1$ und $v2$ von *Prozeß A*, also $v1(\sigma)$ und $v2(\sigma)$.
- Die Funktionen $Z2$ und *fprozeß_C* arbeiten auf dem Strom s weiter.

5.2.3.2 Dynamische Erzeugung einer Menge von Prozeßinstanzen

Der SDL-Prozeß `A` erzeugt eine Menge von Instanzen des SDL-Prozesses `D`. Dabei können diese Instanzen Signale mit den Prozessen `A` und `B` und mit der Blockumgebung austauschen. Die maximale Anzahl der gleichzeitig existierenden Prozeßinstanzen von `D` ist durch max nach oben beschränkt, wobei $max > 1$ sei.

Im Gegensatz zu SDL ist der Nachrichtenaustausch zwischen Komponenten in FOCUS nur über Kanäle möglich. Bei der Umsetzung der SDL-Prozeßerzeugung nach FOCUS sind somit im Unterschied zu SDL für jede neu erzeugte Komponente die entsprechenden Kanalverbindungen zu den bereits vorhandenen Komponenten zur Verfügung zu stellen. Damit sind direkte Kommunikationsverbindungen geschaffen; die Verwendung von eindeutigen Prozeßidentifikatoren für die Adressierung erübrigt sich (siehe Abschnitt 5.2.1). Ferner erhalten die Komponenten, denen die SDL-Prozeßinstanzen der Prozeßmenge zugeordnet werden, in unserer Semantikdefinition unterschiedliche Bezeichner. Die verschiedenen Komponenten

[2]Bei dem Operator $\overline{\oplus}$ handelt es sich um eine spezielle Ausprägung des Kompositionsoperators $\oplus$, der auf funktionaler Ebene die Komposition von gezeiteten mit ungezeiteten Funktionen definiert. Eine Definition dieses Operators ist in [Gro96b] enthalten.

Prozeß D werden durchnumeriert, ebenso die jeweils zu generierenden Kanalverbindungen zwischen einer Komponente *Prozeß* D_i und den Komponenten A und B sowie der Systemumgebung.

$\forall\, i : Nat\ \ \exists\ fprozeß\_D_i \in [\![\ Prozeß\_D_i\ ]\!] :$

$Z1\ [\sigma,\ i,\ c]\ (\{k1 \mapsto a2\}\ \&\ s)\ =$

if $c < max$ then

$$\underbrace{\{k3 \mapsto !k7_i,\ k2 \mapsto ?k8_i\}}_{Ports\ an\ B\ und\ Env}\ \&\ \left(\underbrace{Z3\ [\sigma,\ i+1,\ c+1]}_{Verhalten\ von\ A}\ \overline{\oplus}\ \underbrace{fprozeß\_D_i\ \ (\{?k7_i, !k8_i\},\ v1(\sigma))}_{Verhalten\ von\ D_i}\right)(s)$$

else $Z3\ [\sigma,\ i,\ c]\ (s)$

endif

- Die Variable i dient dazu, die erzeugten Instanzen der Komponente *Prozeß_D* und die neu geschaffenen Kommunikationsverbindungen zu numerieren und damit mit eindeutigen Bezeichnern zu versehen.
- Die Existenz der Funktion *fprozeß_D*$_i$ wird in der Gleichung durch die Forderung $\exists\ fprozeß\_D_i \in [\![\ Prozeß\_D_i\ ]\!]$ sichergestellt. *fprozeß_D*$_i$ ist eine mobile Funktion, die ein zulässiges Verhalten der i-ten Instanz des SDL-Prozesses D darstellt.
- In der Komponente *Prozeß A* trifft auf Kanal $k1$ die Nachricht $a2$ ein, die den Zustandsübergang mit der Erzeugung einer Instanz von *Prozeß D* initiiert.
- Das Erzeugen der i-ten Komponente *Prozeß D*, welches der Prozeßgenerierung entspricht, wird durch den Aufruf der Funktion *fprozeß_D*$_i$ modelliert. Damit wird die i-te Instanz der Komponente erzeugt. Die neu erzeugte Komponente wird über den Kompositionsoperator $\overline{\oplus}$ in die bestehende Systemstruktur eingebunden.
- Bei der Erzeugung von *Prozeß* D_i wird die Schnittstelle initialisiert. Dazu erhält die Funktion *fprozeß_D*$_i$ eine Menge von Ports $\{?k7_i, !k8_i\}$. Mit diesen Ports kann *Prozeß* D_i Kanalverbindungen in der vorhandenen Systemstruktur aufbauen. Die zu den Ports gehörenden Kanäle bilden die initiale Schnittstelle von *Prozeß* D_i. Zudem schickt *Prozeß A* die Ports $!k7_i$ und $?k8_i$ an die Komponente *Prozeß B* bzw. die Blockumgebung, damit diese ebenfalls Zugriff auf die Kanäle erhalten.
- Als weiteren Parameter erhält die Funktion *fprozeß_D*$_i$ den aktuellen Wert der Variablen $v1$ von *Prozeß A*, also $v1(\sigma)$.
- Nach dem Erzeugen einer Komponente wird der Wert der Variablen i um 1 erhöht.
- Der Zähler c gibt die Zahl der aktuell existierenden Prozeßkomponenten an. Er wird nach einem erfolgreichen *Create*-Aufruf um 1 erhöht. Sind bereits *max* Prozeßkomponenten vorhanden, so kann keine weitere Komponentenerzeugung durchgeführt werden. Erhält der Prozeß die Nachricht, daß sich eine von ihm erzeugte Prozeßinstanz beendet hat, so wird der Zähler c um 1 verringert (siehe auch nachfolgende Anmerkungen).

- Die Funktionen $Z3$ und $fprozeß\_D_i$ arbeiten auf dem restlichen Strom s weiter.

Anmerkungen:

1. Um sicherzustellen, daß nicht mehr als max Prozeßinstanzen gleichzeitig existieren, wird der Zähler c verwendet. Da die Zahl der Prozeßinstanzen abnehmen kann, wenn eine Instanz endet, kann die Variable i nicht als Zähler verwendet werden, da sonst eine eindeutige Bezeichnung der Prozeßkomponenten nicht gewährleistet ist.

2. Schicken mehrere Instanzen eines SDL-Prozesses P gleiche Signale über einen Signalweg an einen Prozeß, so kann dieser über den Ausdruck *Sender* abfragen, von welcher Instanz das jeweilige Signal stammt. In der Semantikdefinition läßt sich der Absender eines Signals dadurch bestimmen, daß die Komponenten *Prozeß* P_i die Signale mit ihrem Index i versehen. Die empfangende Komponente kann somit anhand des Indexes feststellen, welche Komponente Absender des Signals ist.

3. Ist die Anzahl der gleichzeitig existierenden Instanzen eines SDL-Prozesses nach oben beschränkt, so ist es notwendig, daß der SDL-Prozeß, der für die Erzeugung zuständig ist, die aktuelle Zahl der existierenden Instanzen kennt. Eine neue Instanz darf nur erzeugt werden, wenn die maximale Anzahl noch nicht erreicht ist. Es ist deshalb wesentlich, daß die Elternprozesse benachrichtigt werden, wenn sich eine der von ihnen erzeugten Instanzen durch die Ausführung des Stopsymbols beendet hat. Für die Semantikdefinition bedeutet dies, daß die sich beendende Komponente vor ihrer Beendigung eine entsprechende Nachricht an die Erzeugerkomponente sendet, damit diese ihren Zählerstand c aktualisieren kann.

4. Ist die Anzahl der gleichzeitig existierenden Instanzen eines SDL-Prozesses unbeschränkt, so entfällt im Gleichungsschema von Seite 91 die Verwendung des Zählers c; die Komponentenerzeugung kann ohne Einschränkung ausgeführt werden. Eine Benachrichtigung über das Löschen von Prozeßinstanzen entfällt ebenfalls.

5.3 Abschließende Bemerkung

In den vorherigen Abschnitten haben wir die formale Semantik für die dynamische Prozeßerzeugung in SDL vorgestellt. Diese Semantik wurde in einer erweiterten Version von FOCUS für mobile, dynamische Systeme mit many-to-many Kommunikation definiert. Der Schwerpunkt lag dabei auf der Umsetzung der Prozeßerzeugung, die Semantikdefinition für die übrigen SDL-Sprachkonstrukte kann ohne Probleme aus der Semantikdefinition für statische SDL-Systeme übernommen werden.

Die semantische Fundierung des *Create*-Konstrukts ist relativ komplex. Neben dem Funktionsaufruf, der die eigentliche Erzeugung umsetzt, ist eine genaue Analyse der bestehenden und zu erzeugenden Kanalverbindungen bei der dynamischen Erzeugung einer neuen Komponente erforderlich. Über das Mobilitätskonzept von FOCUS lassen sich die neuen Kommunikationsverbindungen elegant in das vorhandene System einfügen. Ferner sind lokale

Zähler innerhalb der Prozeßkomponenten einzuführen, um die Anzahl der bereits erzeugten Prozeßkomponenten zu kontrollieren, falls deren Anzahl nach oben beschränkt ist.

Im Gegensatz zum klassischen FOCUS, bei dem jedem Kanal genau eine lesende und eine schreibende Komponente zugeordnet sind (point-to-point Kommunikation), bietet das semantische Modell für dynamische, mobile Systeme die Kommunikationsform many-to-many an, bei der mehrere Komponenten auf einen Kanal zugreifen können. Wir haben diese Kommunikationsform bei der Semantikdefinition für die Prozeßerzeugung verwendet, da damit die Modellierung von Misch- und Verteilkomponenten eines SDL-Prozesses nicht explizit erforderlich ist und wir uns somit auf die wesentlichen Aspekte der Prozeßerzeugung konzentrieren konnten. Jedoch ist das semantische Modell der FOCUS-Erweiterung komplexer, so daß für die Semantikdefinition der statischen SDL-Systeme das klassische FOCUS die geeignete Wahl ist. Zudem ist es unserer Meinung nach für das Verständnis, welches Verhalten ein SDL-Prozeß aufweist, hilfreich, auch die impliziten Prozeßteile wie Mischen und Verteilen von Signalen explizit zu modellieren und damit greifbar zu machen.

Wie bei der Semantikdefinition deutlich wurde, ist mit der Erzeugung eines SDL-Prozesses während des Systemablaufs ein gravierender Eingriff in die vorhandene Systemstruktur verbunden. Werden mehrere Instanzen eines Prozesses erzeugt, so erfolgt die Kommunikation in SDL nicht mehr über Signalwege, sondern über die explizite Adressierung mit Prozeßidentifikatoren. Damit wird das Kommunikationsprinzip über Kanäle und Signalwege durchbrochen: Ein SDL-Prozeß empfängt und sendet nun Signale, ohne daß er mit den jeweiligen Kommunikationspartnern direkt durch Signalwege verbunden ist. Darüber hinaus ist innerhalb eines SDL-Prozesses globales Wissen über die Anzahl der bereits vorhandenen Prozeßinstanzen erforderlich – dies verletzt das Prinzip der Kapselung von Komponenten eines verteilten Systems. Zudem wird in SDL offengelassen, wie dieses Wissen erfaßt wird.

Die gravierenden Eingriffe in den Aufbau und in die Kommunikationsbeziehungen eines Systems durch die Prozeßerzeugung werden durch die Beschreibungsmittel von SDL nur unzureichend erfaßt. Die SDL-Spezifikation modelliert mit den System- und Blockdiagrammen lediglich die Anfangskonfiguration des Systems, nicht jedoch die dynamischen Veränderungen der Systemkonfiguration während des Systemablaufs. Wünschenswert sind Beschreibungsmittel, die diese Veränderungen sichtbar machen und dadurch den Systemingenieur die komplexen Strukturveränderungen leichter nachvollziehen lassen. Für die Spezifikation dynamischer, mobiler Systeme in FOCUS haben wir in [HS97, HS96] eine Vorgehensweise entwickelt, die die dynamischen und mobilen Anteile eines Systems veranschaulicht und vollständig erfaßt. Dazu wurde der Begriff der Systemphasen eingeführt. Eine Phase beschreibt einen Systemzustand, in dem sich das System wie ein statisches System verhält. Eine Veränderung der Struktur oder der Kommunikationsverbindungen des Systems führt in eine neue Phase über. In Anhang A werden wir diese Vorgehensweise bei der Semantikdefinition für das *Daemon Game* vorstellen. Bei dieser Spezifikation handelt es sich um ein Standardbeispiel für die Verwendung von *Create*. An diesem Beispiel entwickeln wir eine systematische Vorgehensweise für die Erfassung der dynamischen Erzeugung von Prozessen und Kommunikationsverbindungen in der Semantikdefinition.

Kapitel 6

Verifikation von SDL-Spezifikationen

In den Kapiteln 4 und 5 haben wir eine formale Semantik für die wichtigsten Sprachkonstrukte von SDL definiert. Spezifikationen, die mit Basic SDL erstellt sind, können nun in FOCUS-Spezifikationen umgesetzt werden und erhalten damit eine formale Semantik – SDL kann jetzt zu Recht als formale Spezifikationssprache bezeichnet werden. Basierend auf der formalen Semantik werden wir in diesem Kapitel eine Methode für die Verifikation von SDL-Spezifikationen entwickeln.

FOCUS bietet aufgrund seiner mathematisch-logischen Fundierung Beweistechniken der funktionalen Logik und der Bereichstheorie, speziell für den Bereich der Ströme. Wir werden zeigen, wie sich diese Beweistechniken für unsere SDL-Semantik übernehmen lassen, und analysieren, inwieweit sich Sprachkonzepte von SDL auf die Beweisführung auswirken. Als Fallstudie hierfür dient uns ein einfaches Kommunikationsprotokoll – das Alternating Bit Protokoll. Wir werden eine SDL-Spezifikation des Protokolls erstellen, die dazugehörige SDL-Semantik angeben und eine wesentliche Eigenschaft des Protokolls, nämlich die korrekte Datenübertragung, verifizieren. Damit liefern wir den Nachweis, daß formale Verifikation basierend auf der von uns definierten SDL-Semantik möglich ist.

Aufbauend auf den Erfahrungen, die wir in der Fallstudie gewonnen haben, werden wir eine Methode für die Verifikation von SDL-Spezifikationen entwickeln. Wir stellen einen durchgängigen Ansatz vor, der in mehreren Phasen einen Weg von der informellen Systembeschreibung über die SDL-Spezifikation bis hin zur Beweisführung beschreibt.

Um die Effizienz und die Akzeptanz von formaler Verifikation zu steigern, ist es wesentlich, die Beweisführung durch ein Verifikationswerkzeug zu unterstützen. Für FOCUS besteht eine Anbindung an den interaktiven, generischen Theorembeweiser Isabelle. Beweise über FOCUS-Spezifikationen, die in der Objektlogik HOLCF von Isabelle formalisiert sind, können werkzeugunterstützt mit Isabelle durchgeführt werden. Um diese Unterstützung auch für SDL-Spezifikationen zu erhalten, werden wir ein Konzept für die Formalisierung der formalen SDL-Semantik in HOLCF entwickeln.

Das Kapitel ist wie folgt gegliedert: In Abschnitt 6.1 zeigen wir anhand des Alternating Bit Protokolls, wie sich Beweistechniken von FOCUS für die Verifikation der SDL-Spezifikation des Protokolls verwenden lassen. In Abschnitt 6.2 wird eine Methode für die Verifika-

tion von SDL-Spezifikationen entwickelt. Anschließend beschreiben wir in Abschnitt 6.3 ein Konzept für die Formalisierung der von uns definierten SDL-Semantik in der Logik HOLCF des Theorembeweisers Isabelle. Das Kapitel schließt mit einem Blick auf einen alternativen Einsatz unserer SDL-Semantik, nämlich die formale Systementwicklung von SDL-Spezifikationen in FOCUS, sowie auf zwei Verfahren, die in der Praxis für die Überprüfung von SDL-Spezifikationen verwendet werden: Model Checking und Simulation.

6.1 Das Alternating Bit Protokoll – eine Fallstudie

Das Alternating Bit Protokoll garantiert die korrekte Datenübertragung zwischen einem Sender und einem Empfänger, die mittels eines fehlerhaften Transportmediums Daten austauschen. Wir verwenden dieses Protokoll als Fallstudie für die Verifikation von SDL-Spezifikationen. Wir erstellen eine SDL-Spezifikation des Protokolls und weisen die Korrektheit von Eigenschaften der Spezifikation mit mathematisch-logischen Mitteln nach. Dies demonstriert, daß unsere SDL-Semantik eine geeignete Basis für die Verifikation von SDL-Spezifikationen bildet.

Korrektheitsbeweise des Alternating Bit Protokolls in verschiedenen Varianten wurden bereits in anderen Arbeiten durchgeführt. So wird beispielsweise in der Arbeit von P. Dybjer und H.P. Sander ([DS89]) eine funktionale Spezifikation des Protokolls im μ-Kalkül verifiziert, wobei die Spezifikation auf den Konzepten der Ströme und Kahn-Netzwerke basiert und somit Ähnlichkeiten zu den Konzepten unserer SDL-Semantik aufweist. Die dort verwendete Beweisidee ist intuitiv und läßt sich an unser semantisches Modell anpassen, so daß wir uns bei der nachfolgenden Verifikationsaufgabe daran orientieren werden. Dies bringt den Vorteil, daß wir den auf unserer SDL-Semantik basierenden Beweis mit dem Beweis in [DS89] vergleichen und Aufschluß darüber erhalten können, ob SDL-spezifische Sprachkonzepte Einfluß auf die Beweisführung haben.

Zunächst beschreiben wir in Abschnitt 6.1.1 das Verhalten des Alternating Bit Protokolls und geben die SDL-Spezifikation des Protokolls an. Anschließend erfolgt in Abschnitt 6.1.2 die Übertragung der SDL-Spezifikation nach FOCUS, womit die formale Semantik des Protokolls gegeben wird. Es folgen in Abschnitt 6.1.3 einige Vereinfachungen an der FOCUS-Spezifikation, bevor in Abschnitt 6.1.4 die Beweisverpflichtung für die korrekte Datenübertragung vorgestellt wird. In Abschnitt 6.1.5 wird der Korrektheitsbeweis durchgeführt und anschließend in Abschnitt 6.1.6 analysiert und mit dem Beweis in [DS89] verglichen.

6.1.1 Informelle Beschreibung und SDL-Spezifikation

Das Alternating Bit Protokoll ist ein einfaches Protokoll, das die sichere Datenübertragung zwischen einem Sender und einem Empfänger über ein fehlerhaftes, aber faires Übertragungsmedium gewährleistet. Jede Nachricht wird vom Medium entweder korrekt oder fehlerhaft übertragen, wobei sich das Medium fair verhält: Wird eine Nachricht mehrmals

hintereinander über das Medium gesendet, so wird das Medium die Nachricht nach endlich vielen Versuchen schließlich korrekt übertragen. Weiterhin gilt, daß das Medium weder Nachrichten verliert noch Nachrichten vertauscht oder verdoppelt.

Die Sendereinheit des Protokolls (kurz Sender) erhält aus der Umgebung Daten, die sie über das Medium an die Empfängereinheit (kurz Empfänger) sendet, die diese an die Umgebung ausgibt. Dabei sollen alle Daten korrekt und in der richtigen Reihenfolge beim Empfänger ankommen. Da das Medium Nachrichten verfälschen kann, muß der Sender möglicherweise eine Dateneinheit mehrmals an den Empfänger senden, bis dieser die Dateneinheit korrekt erhalten hat. Nachrichten, die das Medium verfälscht, sind gekennzeichnet, so daß Sender und Empfänger diese als fehlerhaft identifizieren können. Dies kann zum Beispiel über das Verfahren CRC[1] erfolgen. Um eine sichere Datenübertragung trotz des fehlerhaften Mediums zu gewährleisten, werden Sequenznummern in Form eines Bit verwendet.

Der Sender versieht die zu übertragende Dateneinheit mit einem Kontrollbit. Eine Nachricht an den Empfänger besteht somit aus der Dateneinheit und dem Kontrollbit. Der Empfänger sendet beim Empfang der Nachricht das erhaltene Kontrollbit als Empfangsbestätigung zurück an den Sender. Er leitet die Dateneinheit der erhaltenen Nachricht jedoch nur dann an die Umgebung weiter, wenn das empfangene Kontrollbit sich vom Kontrollbit der zuletzt ausgegebenen Dateneinheit unterscheidet. Nach der Ausgabe einer Dateneinheit an die Umgebung ändert der Empfänger den Wert seines Kontrollbits. Erhält der Empfänger eine fehlerhafte Nachricht vom Medium, so sendet er das Kontrollbit der Dateneinheit, die er zuletzt an die Umgebung ausgegeben hat, an den Sender zurück.

Der Sender erhält als Empfangsbestätigung vom Empfänger ein Kontrollbit. Stimmt der Wert dieses Bits mit dem des Kontrollbits überein, das er mit der Dateneinheit an den Empfänger gesendet hat, so handelt es sich um eine korrekte und erfolgreiche Übertragung. Der Sender aktualisiert sein Kontrollbit und überträgt die nächste Dateneinheit. Unterscheiden sich jedoch die Werte des erhaltenen und des versendeten Bit oder hat der Sender eine fehlerhafte Nachricht vom Medium empfangen, so sendet er die Nachricht, bestehend aus Dateneinheit und Kontrollbit, erneut über das Medium an den Empfänger und wartet auf dessen Bestätigung.

Basierend auf dieser informellen Beschreibung erstellen wir die **SDL-Spezifikation** für den Sender und den Empfänger. Das Verhalten des Mediums werden wir nicht mit SDL beschreiben. Wir gehen davon aus, daß es sich bei dem Medium um eine gegebene Hardware-Komponente handelt, deren Eigenschaften wir mit FOCUS spezifizieren. Die Spezifikation des Verhaltens des Mediums in SDL mittels der nichtdeterministischen Entscheidung *any* ist nicht möglich, da dieses Sprachkonstrukt keine Fairneß bei der Auswahl der Alternativen „korrekte Übertragung“ und „fehlerhafte Übertragung“ garantiert. Das von uns vorgeschlagene Timerkonzept würde die Möglichkeit bieten, durch das Empfangen von Timersignalen diese Auswahl fair zu modellieren. Allerdings ist diese Modellierung unter Einbeziehung des Timermechanismus verhältnismäßig aufwendig. Deshalb ziehen wir die direkte Modellierung der Medien in FOCUS vor (siehe Seite 105).

[1] CRC (Cyclic Redundancy Check) ordnet den zu übertragenden Nachrichten eine Prüfsequenz zu.

Die SDL-Spezifikation `ABP` des Alternating Bit Protokolls umfaßt die Spezifikation der Sendereinheit `ABP_Transmitter`[2] und der Empfängereinheit `ABP_Receiver` sowie die Kommunikationsverbindungen zwischen diesen beiden Blöcken und dem Medium. Um die Modellierung zu vereinfachen und zwei unidirektionale Übertragungswege zwischen `ABP_Transmitter` und `ABP_Receiver` zu erhalten, teilen wir das Medium in zwei Medien `Medium1` und `Medium2` auf. Somit ergibt sich die in Abbildung 6.1 dargestellte Struktur des SDL-Systems `ABP`. Der Block `ABP_Transmitter` enthält als einzigen Prozeß `Transmitter`, der Block `ABP_Receiver` den Prozeß `Receiver`, so daß wir auf die Abbildung der SDL-Blockebene verzichten. Wie zuvor erläutert, spezifizieren wir das Verhalten von `Medium1` bzw. `Medium2` nicht mit SDL. Neben der Struktur des Systems werden die Signale sowie der Datentyp `Bit` definiert.

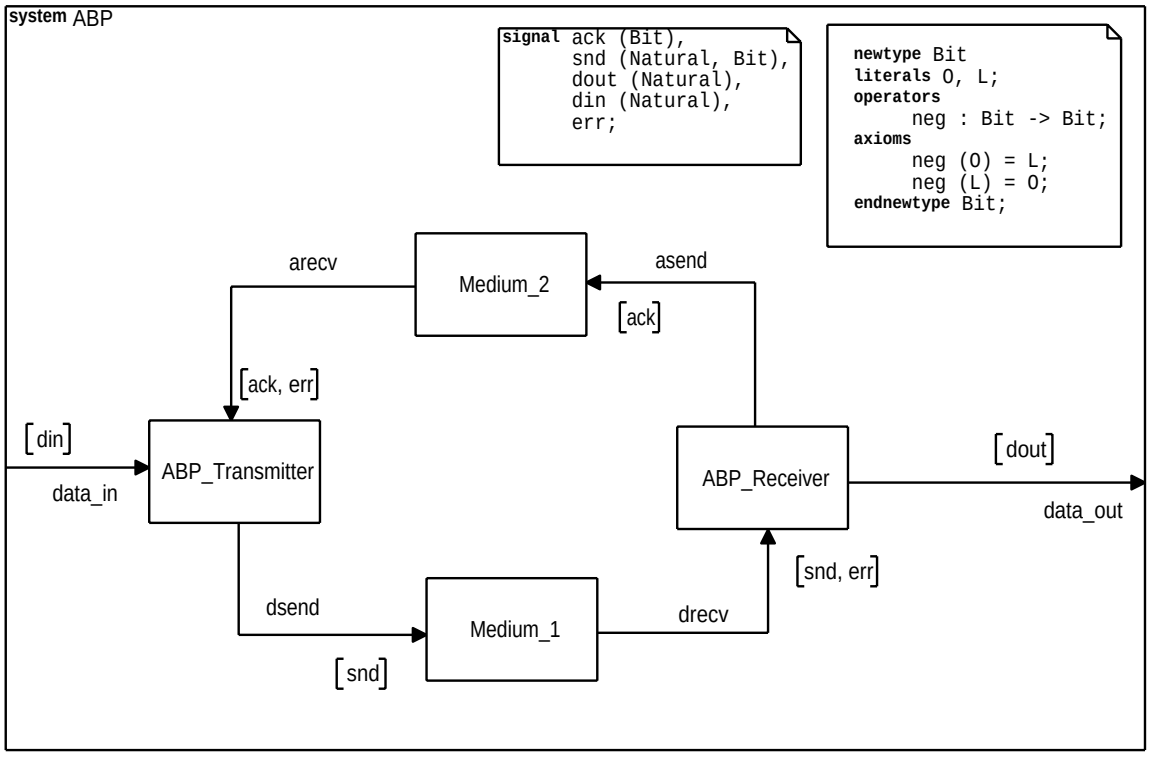


Abbildung 6.1: SDL-Spezifikation des Systemaufbaus des Alternating Bit Protokolls

Die SDL-Prozesse für den Sender, `Transmitter`, und den Empfänger, `Receiver`, sind in den Abbildungen 6.2 und 6.3 angegeben. Die zu übertragenden Dateneinheiten sind vom Typ `Natural`, die Kontrollbits vom Typ `Bit`. Die Systemumgebung sendet die zu übertragenden Dateneinheiten mittels der Signale `din` an das System und erhält die korrekt übertragenen Dateneinheiten über die Signale `dout`. Das Verfälschen einer Nachricht durch eines der Medien modellieren wir dadurch, daß die Medien statt der Nachricht das Signal `err` ausgeben. Der Prozeß `Transmitter` sendet die zu übertragende Dateneinheit `data` zusammen mit dem Kontrollbit `b` mittels des Signals `snd(data,b)` über `Medium1` an den

[2]Die naheliegende Bezeichnung des Blocks und des Prozesses mit „Sender“ ist nicht möglich, da es sich dabei um einen vorbelegten Bezeichner von SDL handelt.

Empfängerprozeß `Receiver`. Der Prozeß `Receiver` sendet das erhaltene Kontrollbit mittels des Signals `ack(sb)` über `Medium2` an den Senderprozeß. Anschließend überprüft er mittels der Entscheidung `sb = rb`, ob die Dateneinheit an die Umgebung auszugeben ist. Solange sich der Senderprozeß im Zustand `ackwait` befindet und auf das Kontrollbit des Empfängerprozesses wartet, werden weitere Signale `din` aus der Umgebung mittels des Savesymbols im Eingabepuffer gespeichert, bis der Senderprozeß wieder in den Zustand `idle` gelangt und bereit ist, die nächste Dateneinheit zu übertragen.

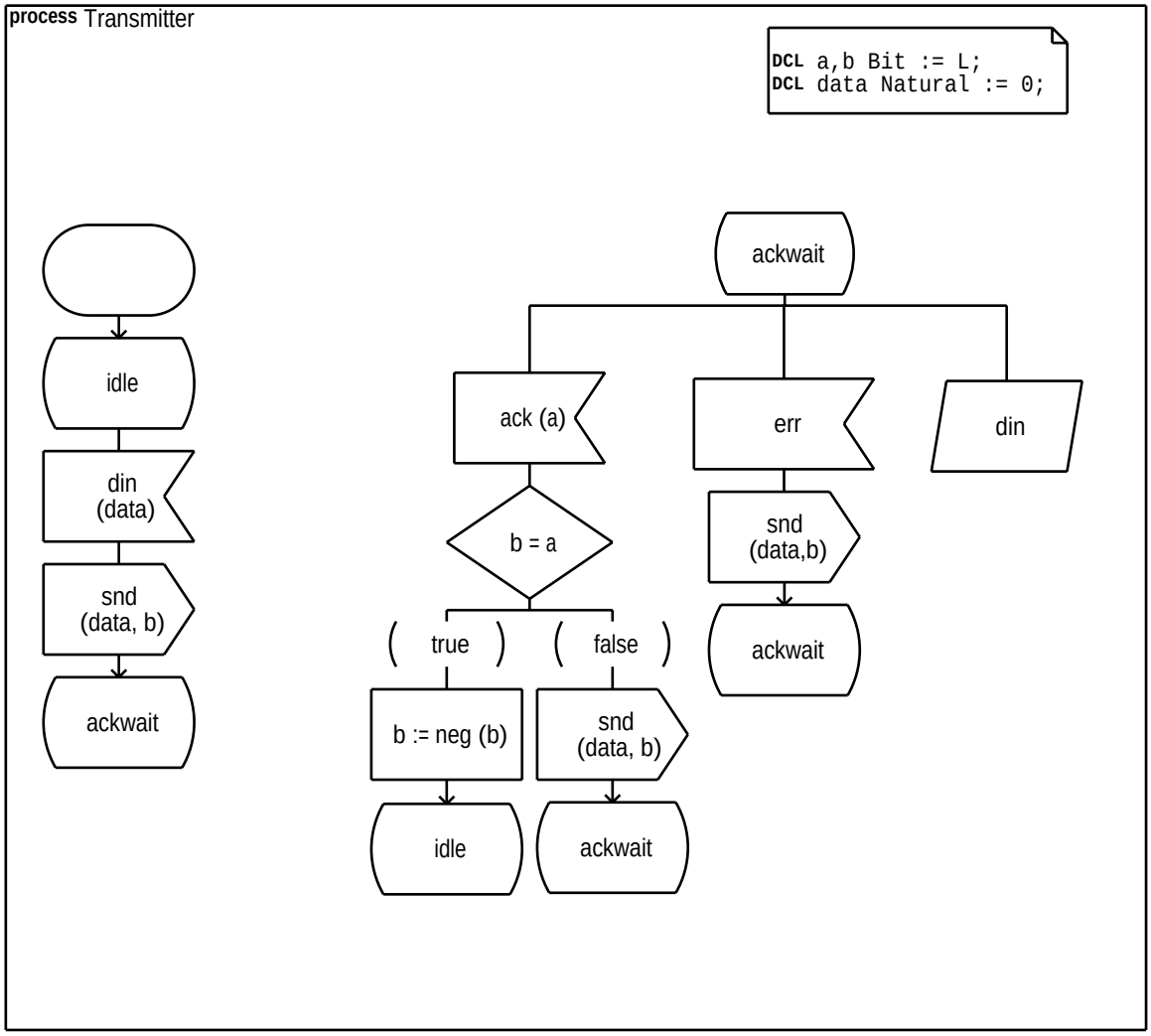


Abbildung 6.2: SDL-Spezifikation des Senders des Alternating Bit Protokolls

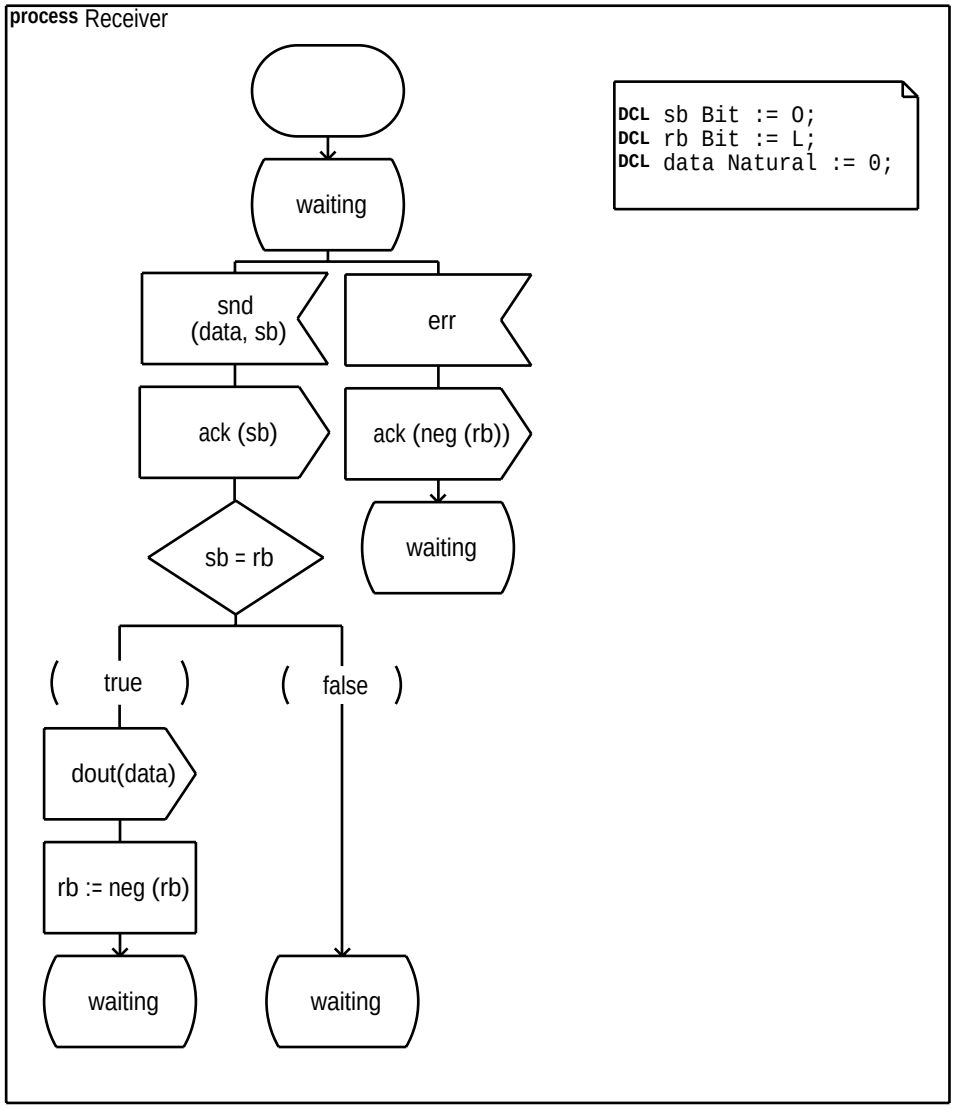


Abbildung 6.3: SDL-Spezifikation des Empfängers des Alternating Bit Protokolls

6.1.2 Überführung der SDL-Spezifikation in die Focus-Spezifikation

Im folgenden geben wir die formale Semantik für die SDL-Spezifikation des Alternating Bit Protokolls an. Wir gehen dabei wie in Kapitel 4 beschrieben vor und bilden die SDL-Spezifikation auf eine FOCUS-Spezifikation ab. Abbildung 6.4 zeigt den strukturellen Aufbau der FOCUS-Spezifikation, wobei die Kanäle mit Nachrichtenmengen versehen sind. Laut unseren Richtlinien für das schematische Erstellen der FOCUS-Spezifikation (siehe Seite 52) entfällt die Einführung von Komponenten der Art *Block* in der FOCUS-Spezifikation, da die SDL-Blöcke `ABP_Transmitter` und `ABP_Receiver` nur jeweils einen SDL-Prozeß enthalten.

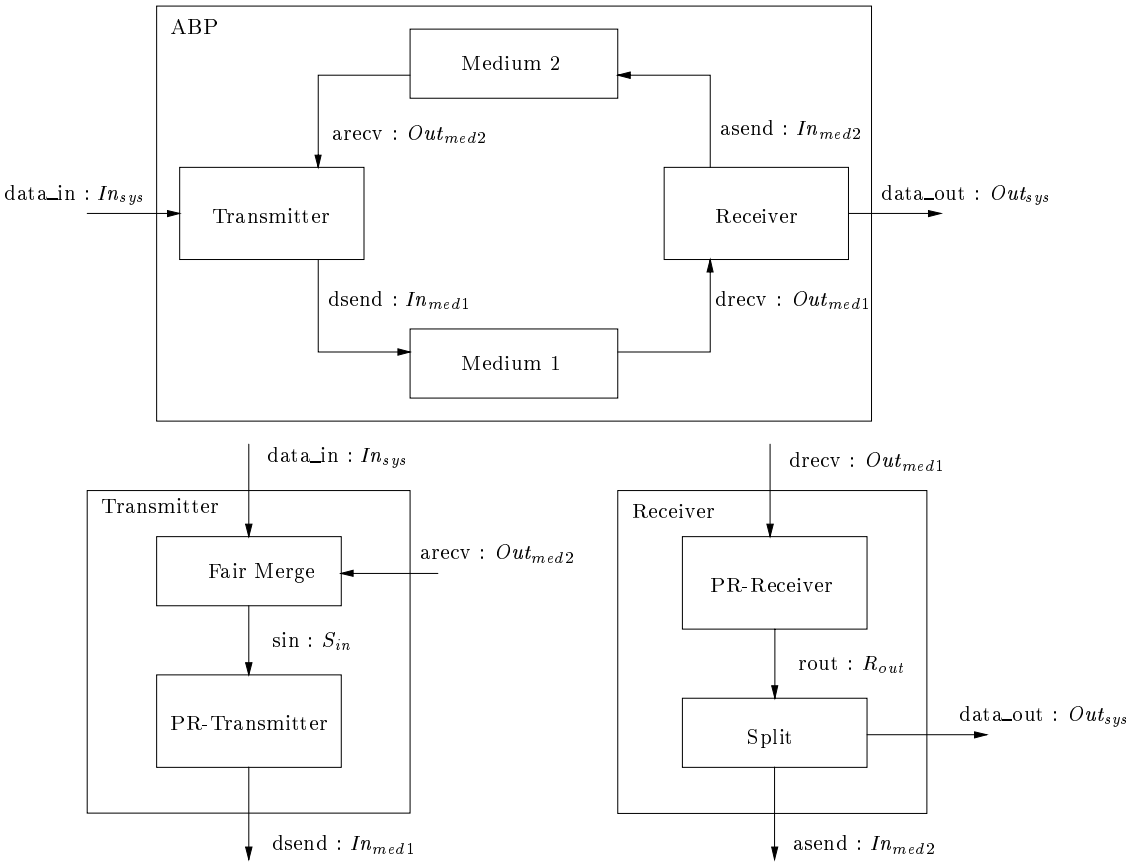


Abbildung 6.4: Struktur der FOCUS-Spezifikation des Alternating Bit Protokolls

Wir setzen folgende abstrakten **Datentypen** aus SPECTRUM als gegeben voraus (siehe [BFG+93a])

- natürliche Zahlen: *Nat*
- $Bit = \{O, L\}$ mit $\neg O = L$ und $\neg L = O$

und bilden die SDL-Datentypen `Bit` und `Natural` auf diese SPECTRUM-Typen ab. Die Operation $neg(b)$ aus SDL entspricht in der FOCUS-Spezifikation somit $\neg b$.

Aus der SDL-Spezifikation leiten wir folgende **Nachrichtenmengen** für die Kanäle in FOCUS ab:

$$
\begin{aligned}
In_{sys} &= \{din\ (d) \mid d : Nat\} \\
Out_{sys} &= \{dout\ (d) \mid d : Nat\} \\
In_{med1} &= \{snd\ (d,\ b) \mid d : Nat,\ b : Bit\} \\
Out_{med1} &= \{snd\ (d,\ b) \mid d : Nat,\ b : Bit\} \cup \{err\} \\
In_{med2} &= \{ack\ (b) \mid b : Bit\} \\
Out_{med2} &= \{ack\ (b) \mid b : Bit\} \cup \{err\} \\
S_{in} &= In_{sys} \cup Out_{med2} \\
R_{out} &= Out_{sys} \cup In_{med2}
\end{aligned}
$$

Gemäß unserer Semantikdefinition für SDL in Kapitel 4 erhalten wir folgende Spezifikation des **Focus-Systems** *ABP*:

agent *ABP*

input channel $data\_in : In_{sys}$
output channel $data\_out : Out_{sys}$

is network

$$
\begin{aligned}
< dsend > &= Transmitter < data\_in, arecv >; \\
< data\_out, asend > &= Receiver < drecv >; \\
< drecv > &= Medium1 < dsend >; \\
< arecv > &= Medium2 < asend >
\end{aligned}
$$

end *ABP*

agent *Transmitter*

input channel $data\_in : In_{sys},\ arecv : Out_{med2}$
output channel $dsend : In_{med1}$

is network

$$
\begin{aligned}
< sin > &= Fair\_Merge < data\_in, arecv >; \\
< dsend > &= PR\_Transmitter < sin >
\end{aligned}
$$

end *Transmitter*

agent *Fair Merge* – **time dependent** – **wp**

input channel $data\_in : In_{sys},\ arecv : Out_{med2}$
output channel $sin : S_{in}$

is basic

$$\begin{aligned} data\_in &= (In_{sys} \cup \{\surd\})\ ©\ sin \wedge \\ arecv &= (Out_{med2} \cup \{\surd\})\ ©\ sin \end{aligned}$$

end *Fair Merge*

agent *PR_Transmitter* – **time independent**

input channel $sin : S_{in}$
output channel $dsend : In_{med1}$

is basic

$$\exists\ idle : Nat \times Bit \times S^{\omega}_{in} \to In^{\omega}_{med1}.\ idle\ [0,\ L]\ (sin)\ =\ dsend$$

where

$$\forall\ s : S^{\omega}_{in}.\ \forall\ a,\ b : Bit.\ \forall\ data,\ d : Nat.$$

$$idle\ [d,\ b]\ (din\ (data)\ \&\ s) = snd\ (data,\ b)\ \&\ ackwait\ [data,\ b]\ (s) \wedge$$

$$idle\ [d,\ b]\ \ (ack\ (a)\ \&\ s) = idle\ [d,\ b]\ \ (s)\ \wedge$$

$$idle\ [d,\ b]\ \ (err\ \&\ s) =\ idle\ [d,\ b]\ \ (s)\ \wedge$$

$ackwait\ [d,\ b]\ (s) =$
case $search(\{ack\ (a),\ err\},\ s)$ of
  $ack\ (a)$: if $a = b$
    then $idle\ [d,\ \neg b]\ (del(ack(a), s))$
    else $snd\ (d,\ b)\ \&\ ackwait\ [d,\ b]\ (del(ack(a), s))$
    endif
  err : $snd\ (d,\ b)\ \&\ ackwait\ [d,\ b]\ (del(err, s))$
endcase

end *PR_Transmitter*

Gemäß der Anmerkung auf Seite 72 können wir darauf verzichten, die Variable `a:Bit` des SDL-Prozesses `Transmitter` in den Datenzustand der Funktion *idle* aufzunehmen. Gleiches gilt für die Variablen `data:Natural` und `sb:Bit` des SDL-Prozesses `Receiver`. Da die SDL-Prozesse `Transmitter` und `Receiver` über nur wenige lokale Variablen verfügen, verzichten wir auf die explizite Einführung eines Datenzustands $\sigma : D$ und verwenden direkt die Variablen als Parameter in den Funktionen *idle*, *ackwait* bzw. *waiting*.

agent *Receiver*

input channel $drecv : Out_{med1}$
output channel $data\_out : Out_{sys},\ asend : In_{med2}$

is network

$$< rout > \quad = \quad PR\_Receiver < drecv >;$$
$$< data\_out,\ asend > \quad = \quad Split < rout >$$

end *Receiver*

agent *PR_Receiver* – **time independent**

input channel $drecv : Out_{med1}$
output channel $rout : R_{out}$

is basic

$$\exists\ waiting :\ Bit \times Out^{\omega}_{med1} \to R^{\omega}_{out}.\ waiting\ [L]\ (drecv)\ =\ rout$$

where

$$\forall\, s : Out^{\omega}_{med1}.\ \forall\ data : Nat.\ \forall\ rb,\ sb : Bit.$$

$$waiting\ [rb]\ (snd\ (data,\ sb)\ \&\ s) =$$
if $rb = sb$
then $ack\ (sb)\ \&\ dout\ (data)\ \&\ waiting\ [\neg rb]\ (s)$
else $ack\ (sb)\ \&\ waiting\ [rb]\ (s)$
endif $\wedge$

$$waiting\ [rb]\ (err\ \&\ s) = ack\ (\neg rb)\ \&\ waiting\ [rb]\ (s)$$

end *PR_Receiver*

agent *Split* – **time dependent** – **wp**

input channel $rout : R_{out}$
output channel $data\_out : Out_{sys},\ asend : In_{med2}$

is basic

$$data\_out \quad = (Out_{sys} \cup \{\surd\})\ ©\ rout\ \wedge$$
$$asend \quad = (In_{med2} \cup \{\surd\})\ ©\ rout$$

end *Split*

Das Verhalten der beiden Blöcke `Medium1` und `Medium2` aus der SDL-Spezifikation `ABP` ist nicht mit SDL spezifiziert, sondern wird direkt in FOCUS beschrieben:
Jedes Medium verfügt über ein Orakel, das festlegt, ob eine Nachricht korrekt oder fehlerhaft übertragen wird. Das Orakel ist ein Strom von Nachrichten des Typs *Bit*. Liefert

es O, so überträgt das Medium die Nachricht fehlerhaft, liefert es L, erfolgt die Übertragung korrekt. Um die Fairneß der Übertragung zu garantieren, fordern wir, daß das Orakel unendlich viele L enthält.

agent *Medium1* – **time independent**

input channel $dsend : In_{med1}$
output channel $drecv : Out_{med1}$

is basic

$$\exists\ o : Bit^{\infty}.\ \exists\ med_1 : Bit^{\infty} \times In^{\omega}_{med1} \to Out^{\omega}_{med1}.$$

$$\#(\{L\} \copyright o) = \infty\ \wedge\ med_1\ (o,\ dsend) = drecv$$

where

$$\forall\, s : In^{\omega}_{med1}.\ \forall\ m : In_{med1}.\ \forall\ q : Bit^{\infty}.$$

$$med_1\ (O\ \&\ q,\ m\ \&\ s) = err\ \&\ med_1\ (q,\ s)\ \wedge$$
$$med_1\ (L\ \&\ q,\ m\ \&\ s) = m\ \&\ med_1\ (q,\ s)$$

end *Medium1*

agent *Medium2* – **time independent**

input channel $asend : In_{med2}$
output channel $arecv : Out_{med2}$

is basic

$$\exists\ o : Bit^{\infty}.\ \forall\ med_2 : Bit^{\infty} \times In^{\omega}_{med2} \to Out^{\omega}_{med2}.$$

$$\#(\{L\} \copyright o) = \infty\ \wedge\ med_2\ (o,\ asend) = arecv$$

where

$$\forall\, s : In^{\omega}_{med2}.\ \forall\ m : In_{med2}.\ \forall\ q : Bit^{\infty}.$$

$$med_2\ (O\ \&\ q,\ m\ \&\ s) = err\ \&\ med_2\ (q,\ s)\ \wedge$$
$$med_2\ (L\ \&\ q,\ m\ \&\ s) = m\ \&\ med_2\ (q,\ s)$$

end *Medium2*

6.1.3 Vereinfachung der FOCUS-Spezifikation

Im vorherigen Abschnitt haben wir die gegebene SDL-Spezifikation gemäß der von uns entwickelten Vorgehensweise für die Semantikdefinition in eine FOCUS-Spezifikation umgesetzt. Die semantische Umsetzung ist dabei bewußt so allgemein gehalten, daß sie auf alle SDL-Spezifikationen anwendbar ist. Für manche SDL-Spezifikationen läßt sich eine Vereinfachung der FOCUS-Spezifikation erzielen. Wir werden dies im folgenden an der Spezifikation *ABP* vorführen.

In der eben erstellten FOCUS-Systembeschreibung wird das Verhalten der Basiskomponenten *PR_Transmitter* und *PR_Receiver* durch die Funktionen *idle* und *ackwait* bzw. *waiting* auf ungezeiteten Strömen spezifiziert. Diese Funktionen resultieren aus der schematischen Umsetzung der Zustandsübergänge der SDL-Prozeßgraphen in Funktionsgleichungen. Bei einer genaueren Betrachtung ergeben sich Möglichkeiten, diese Funktionsdefinitionen und damit die anschließende Beweisführung zu vereinfachen.

- Aus einem SDL-Prozeßgraphen werden für die unterschiedlichen Zustände Funktionen abgeleitet, wobei jede Funktion über den gesamten Datenzustand des SDL-Prozesses verfügt. Da einige Variablen des Datenzustands nur für bestimmte Zustandsübergänge des Prozesses und damit in der Semantik nur für bestimmte Funktionen erforderlich sind, kann in manchen Funktionen auf diese Variablen verzichtet werden. In der FOCUS-Spezifikation *ABP* werden wir deshalb in der Spezifikation von *PR_Transmitter* bei der Funktion *idle* auf die Variable für den zu übertragenden Datenwert verzichten, da es genügt, wenn dieser Wert bis zu seiner erfolgreichen Übertragung von der Funktion *ackwait* in einer Variablen gespeichert wird (siehe nachfolgende Definition von $idle'$).

- Die Verwendung von Savesignalen in einem Zustand führt in der SDL-Semantik zu einer Funktion, die nicht durchgängig auf das jeweils erste Element des Eingabestroms zugreifen kann, sondern das erste Element im Strom verarbeitet, das nicht in der Menge der Savesignale liegt (siehe Abschnitt 4.3.2.4). Dies erschwert die Argumentation über das Verhalten der Funktion. Wir werden deshalb den Datenzustand der Funktion um eine Sequenz erweitern, die Savesignale zwischenspeichert und somit als Puffer dient. Savesignale können dadurch wie „normale“ Eingabesignale behandelt werden; es kann auf das jeweils erste Element statt auf ein Element innerhalb des Eingabestroms zugegriffen werden. Handelt es sich um ein Savesignal, so wird es an die Sequenz der Savesignale angehängt und aus dem Eingabepuffer gelöscht. Sobald ein Aufruf einer anderen Funktion erfolgt, wird die Sequenz der Savesignale an den Beginn des aktuellen Eingabestroms gehängt.
  In der Spezifikation von *ABP* wenden wir dieses Vorgehen auf die Funktion *ackwait* an. Anstelle mittels der Funktion *search* im Eingabestrom nach einem Signal zu suchen, das kein Savesignal ist, werden die Savesignale, die vor einem solchen Signal im Eingabestrom vorkommen, in einer Sequenz gespeichert. Diese Sequenz ist Teil des Datenzustands von *ackwait*. Durch dieses Vorgehen erübrigt sich auch die Anwendung der Funktion *del*, die das Signal, das die Funktion *search* gefunden hat, nach Durchführung des Zustandsübergangs aus dem Eingabestrom entfernt (siehe nachfolgende Definition von $ackwait'$).
  In der SDL-Semantikdefinition haben wir den Savemechanismus sehr nahe an dem durch SDL gegebenen Konzept behandelt und keine explizite Zerlegung des Eingabepuffers vorgenommen, sondern mittels der Funktionen *search* und *del* auf Elemente des Eingabepuffers zugegriffen. Die nun vorgestellte Zerlegung des Eingabepuffers in eine Sequenz von Savesignalen und den restlichen Eingabepuffer ist im Vergleich zur Semantikdefinition operationell geprägt, weist aber den Vorteil auf, daß durchgehend auf die erste Nachricht des Eingabestroms zugegriffen wird.

- Die Komponente *Receiver* ist ein Netzwerk bestehend aus den Komponenten *Split* und *PR_Receiver*. Die Komponente *PR_Receiver* erhält den Eingabestrom *drecv* und erzeugt einen Ausgabestrom, der anschließend von der Komponente *Split* in ein Tupel von Ausgabeströmen *data_out* und *asend* zerlegt wird.
  Da es sich bei *PR_Receiver* um eine Komponente mit einem einfachen Verhalten handelt, kann die Aufteilung in zwei Ausgabeströme direkt in *PR_Receiver* anstatt in *Split* vorgenommen werden. Wir ersetzen die Netzwerkkomponente *Receiver* durch die Basiskomponente *Receiver'*, die das gleiche Verhalten wie *Receiver* aufweist. Die Komponente erhält den Eingabestrom *drecv* und erzeugt direkt die beiden Ausgabeströme *data_out* und *asend* (siehe nachfolgende Definition von *Receiver'*).

Im folgenden führen wir die eben erläuterten Vereinfachungen an der FOCUS-Spezifikation durch, indem wir

- in der Basiskomponente *PR_Transmitter* die Funktionen *idle* und *ackwait* durch *idle'* und *ackwait'* ersetzen und
- das Netzwerk *Receiver* durch die Basiskomponente *Receiver'* ersetzen.

Die Korrektheit der vorgenommenen Vereinfachungen ist formal zu beweisen. Dazu ist zu zeigen, daß die Komponenten *Receiver* und *Receiver'* bzw. die Funktionen *idle* und *idle'* das gleiche Verhalten aufweisen. Die dafür erforderlichen Beweise sind einfach und werden mittels struktureller Induktion und Induktion über die Stromlänge geführt.

Definition von $idle'$ und $ackwait'$:

$idle' : Bit \times S^{\omega}_{in} \to In^{\omega}_{med1}$
$ackwait' : \{din(c) \mid c : Nat\}^{\omega} \times Nat \times Bit \times S^{\omega}_{in} \to In^{\omega}_{med1}$

where

$$\forall\, a,\ b : Bit.\ \forall\, d,\ data : Nat.\ \forall\, ds : \{din(c) \mid c : Nat\}^{\omega}.\ \forall\, s : S^{\omega}_{in}.$$

$$idle'\ [b]\ (din(d)\ \&\ s) \ = \ snd(d,\ b)\ \&\ ackwait'\ [\langle\rangle,\ d,\ b]\ (s)$$

$$idle'\ [b]\ (ack(a)\ \&\ s) \ = \ idle'\ [b]\ (s)$$

$$idle'\ [b]\ (err(a)\ \&\ s) \ = \ idle'\ [b]\ (s)$$

$$ackwait'\ [ds,\ d,\ b]\ (din(data)\ \&\ s) \ = \ ackwait'\ [ds \circ \langle din(data)\rangle,\ d,\ b]\ (s)$$

$$ackwait'\ [ds,\ d,\ b]\ (err\ \&\ s) \ = \ snd(d,\ b)\ \&\ ackwait'\ [ds,\ d,\ b]\ (s)$$

$$ackwait'\ [ds,\ d,\ b]\ (ack(a)\ \&\ s) \ = \ \text{if } b = a$$
$$\text{then } idle'\ [\neg b]\ (ds \circ s)$$
$$\text{else } snd(d,\ b)\ \&\ ackwait'\ [ds,\ d,\ b]\ (s)$$
$$\text{endif}$$

Definition von *Receiver'*:

agent *Receiver'* – **time independent**

input channel $drecv : Out_{med1}$
output channel $data\_out : Out_{sys},\ asend : In_{med2}$

is basic

$\exists\ rsend : Bit \times Out^{\omega}_{med1} \to Out^{\omega}_{sys}.\ \exists\ rack : Bit \times Out^{\omega}_{med1} \to In^{\omega}_{med2}.$
$rsend\ [L]\ (drecv) = data\_out \wedge rack\ [L]\ (drecv) = asend$

where

$\forall\ s : Out^{\omega}_{med1}.\ \forall\ d : Nat.\ \forall\ rb,\ sb : Bit.$

$rsend\ [rb]\ (snd(d,\ sb)\ \&\ s) =$
if $sb = rb$ then $dout(d)\ \&\ rsend\ [\neg rb]\ (s)$ else $rsend\ [rb]\ (s)$ endif $\wedge$
$rsend\ [rb]\ (err\ \&\ s) = rsend\ [rb]\ (s)\ \wedge$

$rack\ [rb]\ (snd(d,\ sb)\ \&\ s) =$
if $sb = rb$ then $ack(sb)\ \&\ rack\ [\neg rb]\ (s)$ else $ack(sb)\ \&\ rack\ [rb]\ (s)$ endif $\wedge$
$rack\ [rb]\ (err\ \&\ s) = ack(\neg rb)\ \&\ rack\ [rb]\ (s)$

end *Receiver'*

6.1.4 Beweisverpflichtung und Vorgehensweise

Ziel unserer Fallstudie ist es, formal mit mathematisch-logischen Mitteln nachzuweisen, daß die SDL-Spezifikation des Alternating Bit Protokolls eine zuverlässige Übertragung der Nachrichten über die fehlerhaften, aber fairen Medien realisiert. In diesem Abschnitt stellen wir die Beweisverpflichtung vor, d.h. wir formulieren die Korrektheitsaussage, die wir für die SDL-Spezifikation beweisen werden. Den Beweis werden wir mittels Induktion führen (siehe Abschnitt 6.1.5). Für die Beweisführung ist es erforderlich, auf das semantische Modell der eben vorgestellten FOCUS-Spezifikation zuzugreifen. Wir stellen deshalb den Bezug zwischen der FOCUS-Spezifikation und ihrem semantischen Modell vor und geben eine Vorgehensweise für die Behandlung zeitabhängiger und zeitunabhängiger Verhaltensanteile in der Beweisführung an.

Die Semantik der FOCUS-Spezifikation *ABP*, die wir in Abschnitt 6.1.2 der SDL-Spezifikation zugeordnet und in Abschnitt 6.1.3 vereinfacht haben, ist durch eine Menge stark pulsgetriebener Funktionen definiert. Aus der Spezifikation *ABP* wird ein Prädikat *is_abp* abgeleitet, das diejenigen stark pulsgetriebenen Funktionen auszeichnet, die ein zulässiges Verhalten von *ABP* beschreiben (siehe auch Abschnitt 3.2.2). In Anhang B.1 sind sämtliche Prädikate aufgeführt, die aus der ANDL-Spezifikation des Alternating Bit Protokolls resultieren. Die Semantik der SDL-Spezifikation des Alternating Bit Protokolls lautet im semantischen Modell von FOCUS:

$$[\![ABP]\!] \stackrel{def}{=} \{f \mid is\_abp(f)\}$$

Jede Funktion aus der Menge $[\![ABP]\!]$ repräsentiert ein mögliches Ein-/Ausgabeverhalten des Systems. Wir erhalten eine Menge von Funktionen, weil in der SDL-Spezifikation und in der Spezifikation der beiden Medien nichtdeterministische Verhaltensanteile enthalten sind (vgl. auch Abschnitt 4.4). Mittels eines formalen Beweises werden wir zeigen, daß jedes mögliche Verhalten des Systems die korrekte Übertragung von Nachrichten garantiert.

Wir formulieren die Korrektheitsbedingung für die SDL-Spezifikation zunächst textuell:

> Für alle Ströme, die zulässige Ein- bzw. Ausgabeströme des Systems sind, gilt: wird im Ein- und Ausgabestrom von der Zeit abstrahiert, so sind der Datenanteil des Eingabestroms und der Datenanteil des Ausgabestroms identisch. Jede Dateneinheit des Eingabestroms wird übertragen, die Reihenfolge der Daten bleibt erhalten; die Übertragung der Daten erfolgt mit beliebiger zeitlicher Verzögerung.

In der SDL-Spezifikation werden die zu übertragenden Daten mittels der Signale $din(d)$ bzw. $dout(d)$ an das System bzw. an die Umgebung gesendet, so daß die Ein- und Ausgabeströme des Systems syntaktisch verschieden sind, auch wenn ihr Datenanteil übereinstimmt. Um die Ein- und Ausgabeströme hinsichtlich ihres Datenanteils vergleichen zu können, führen wir eine Funktion ex ein, die aus den Ein- und Ausgabenachrichten des Protokolls den Datenanteil extrahiert:

$$ex : S \to Nat \qquad \text{mit } S = \{din(d), dout(d) \mid d : Nat\}$$
$$\forall\, d : Nat.\ ex(din(d)) = d \wedge ex(dout(d)) = d$$

Die Funktion ex^* definieren wir als die punktweise Erweiterung von ex auf einen Strom:

$$ex^* : S^\omega \to Nat^\omega$$
$$\text{mit } \forall\, m : S,\ s : S^\omega.\ ex^*(m \,\&\, s) = ex(m) \,\&\, ex^*(s) \ \wedge\ ex^*(\langle\rangle) = \langle\rangle$$

Die textuell formulierte Korrektheitsbedingung überführen wir nun in folgende Beweisverpflichtung auf logischer Ebene:

$$\forall f \in [\![ABP]\!].\ \forall in : In_{sys}^{\infty}.\ \forall out : Out_{sys}^{\infty}.\ f(in) = out \Longrightarrow ex^*(\overline{in}) = ex^*(\overline{out})$$

Die Idee für den Korrektheitsbeweis orientiert sich an [DS89]. Ausgehend vom Anfangszustand des Protokolls wird ein Zustand erreicht, bei dem die erste zu übertragende Dateneinheit vom Empfänger erhalten und an die Umgebung ausgegeben, das dazugehörige Kontrollbit jedoch noch nicht vom Sender empfangen worden ist. Ausgehend von diesem Zustand läßt sich ein nachfolgender Zustand erreichen, der sich vom Anfangszustand in folgenden Punkten unterscheidet: der Eingabestrom ist um das erste Element verringert, die Dateneinheit ist korrekt übertragen worden, und die Kontrollbits bei Sender und Empfänger haben ihren Wert geändert. Jetzt kann der Sender die nächste Dateneinheit übertragen.

Für die Umsetzung dieser Beweisidee ist es erforderlich, Aussagen über das Zusammenwirken der einzelnen Systemkomponenten zu treffen. Dazu betrachten wir die Struktur des Systems ABP: Jede Funktion $f \in [\![ABP]\!]$ wird durch das Zusammenspiel mehrerer Funktionen realisiert, die ihrerseits das Verhalten der Komponenten des Systems beschreiben. Dies wird im Prädikat *is_abp* definiert (siehe Seite 166). Die Komponenten des Systems können wiederum aus Komponenten bestehen, deren Verhalten ebenfalls durch Funktionen definiert ist. Betrachten wir den Aufbau des Systems auf seiner untersten Strukturierungsebene, so wird das Verhalten des Gesamtsystems durch eine Menge von Funktionen beschrieben, die jeweils das Verhalten einer Basiskomponente festlegen. In folgender Formel formulieren wir diesen Zusammenhang. Für jede Basiskomponente des Systems existiert eine pulsgetriebene Funktion, die die Beziehung zwischen dem Ein- und dem Ausgabestrom der Komponente festlegt. Daraus folgt das Gleichungssystem (1) über den Strömen *in* und *out* sowie über den internen Strömen *asend*, *arecv*, *sin*, *dsend* und *drecv*. Es ist zu beachten, daß wir von der vereinfachten FOCUS-Systembeschreibung in Abschnitt 6.1.3 ausgehen.

$$
\begin{aligned}
&\forall f \in [\![ABP]\!].\ \forall in : In_{sys}^{\infty}.\ \forall out : Out_{sys}^{\infty}.\ f(in) = out \implies \\
&\quad \exists pr\_sender \in [\![PR\_Sender]\!].\ \exists merge \in [\![Fair\ Merge]\!].\ \exists receiver \in [\![Receiver']\!]. \\
&\quad \exists medium_1 \in [\![Medium1]\!].\ \exists medium_2 \in [\![Medium2]\!]. \\
&\quad \exists asend : In_{med2}^{\infty}.\ \exists arecv : Out_{med2}^{\infty}.\ \exists sin : S_{in}^{\infty}.\ \exists dsend : In_{med1}^{\infty}.\ \exists drecv : Out_{med1}^{\infty}. \\
&\qquad \left.\begin{aligned}
sin &= merge\ (in,\ arecv) \wedge \\
dsend &= pr\_sender\ (sin) \wedge \\
drecv &= medium_1\ (dsend) \wedge \\
(out,\ asend) &= receiver\ (drecv) \wedge \\
arecv &= medium_2\ (asend)
\end{aligned}\right\} (1)
\end{aligned}
$$

Die Strombelegung der internen Kanäle wird durch das rekursive Gleichungssystem (1) definiert. In unserem semantischen Modell gilt, daß jede stark pulsgetriebene, rekursiv definierte Funktion einen eindeutigen unendlichen Fixpunkt besitzt (siehe Seite 42). Somit existiert für jede Funktion $f \in [\![ABP]\!]$, die durch die Komposition der Funktionen *merge*, *pr_sender*, *medium*1, *medium*2 und *receiver* rekursiv definiert ist, ein eindeutiger Fixpunkt, der das Gleichungssystem erfüllt. Dieser Fixpunkt setzt sich aus den fünf unendlichen gezeiteten Strömen *asend*, *arecv*, *sin*, *dsend* und *drecv* zusammen. Für jedes Strompaar (in, out), das ein gültiges Ein-/Ausgabeverhalten des Netzwerks beschreibt, gibt es eine eindeutige Strombelegung der internen Verbindungsstruktur, die das Gleichungssystem erfüllt.

Da zeitliche Betrachtungen für das korrekte Übertragungsverhalten des Alternating Bit Protokolls keine Rolle spielen, werden wir den Korrektheitsbeweis nicht unmittelbar im gezeiteten semantischen Modell von FOCUS, sondern auf der Spezifikationsebene von FOCUS durchführen, auf der bis auf *Fair Merge* alle Basiskomponenten zeitunabhängig spezifiziert sind. Für Basiskomponenten mit dem Schlüsselwort `time independent` ist das Ein-/Ausgabeverhalten als Relation zwischen ungezeiteten Ein- und Ausgabeströmen spezifiziert. Innerhalb der Basiskomponenten sind im Verhaltensanteil dafür Funktionen auf

ungezeiteten Strömen angegeben. Das Gleichungssystem (1) impliziert nun das Gleichungssystem (2), in dem von der Zeitinformation in den Nachrichtenströmen abstrahiert wird, sofern sie Ein- bzw. Ausgabeströme von zeitunabhängigen Basiskomponenten sind. Auf die Ströme des Tupels, das die Funktion *receiver* als Ausgabe erzeugt, wird mit den Projektionsfunktionen π_1 und π_2 zugegriffen.

$$\begin{aligned}
&\forall f \in [\![ABP]\!].\ \forall in : In^{\infty}_{sys}.\ \forall out : Out^{\infty}_{sys}.\ f(in) = out \implies \\
&\quad \exists pr_sender \in [\![PR_Sender]\!].\ \exists merge \in [\![Fair\ Merge]\!].\ \exists receiver \in [\![Receiver']\!]. \\
&\quad \exists medium_1 \in [\![Medium1]\!].\ \exists medium_2 \in [\![Medium2]\!]. \\
&\quad \exists asend : In^{\infty}_{med2}.\ \exists arecv : Out^{\infty}_{med2}.\ \exists sin : S^{\infty}_{in}.\ \exists dsend : In^{\infty}_{med1}.\ \exists drecv : Out^{\infty}_{med1}.
\end{aligned}$$

$$\left.\begin{aligned}
sin &= merge\ (in,\ arecv) \wedge \\
dsend &= pr_sender\ (sin) \wedge \\
drecv &= medium_1\ (dsend) \wedge \\
(out,\ asend) &= receiver\ (drecv) \wedge \\
arecv &= medium_2\ (asend)
\end{aligned}\right\} (1) \qquad \Longrightarrow$$

$$\begin{aligned}
&\exists idle' : Bit \times S^{\omega}_{in} \to In^{\omega}_{med1} && \text{with } \overline{pr_sender(sin)} = idle'\ [L]\ (\overline{sin}). \\
&\exists rsend :\ Bit \times Out^{\omega}_{med1} \to Out^{\omega}_{sys} && \text{with } \overline{\pi_1(receiver(arecv))} = rsend\ [L]\ (\overline{arecv}). \\
&\exists rack :\ Bit \times Out^{\omega}_{med1} \to In^{\omega}_{med2} && \text{with } \overline{\pi_2(receiver(arecv))} = rack\ [L]\ (\overline{arecv}). \\
&\exists p_1,\ p_2 : Bit^{\infty} && \text{with } \{L\}\copyright p_1 = \infty \wedge \{L\}\copyright p_2 = \infty. \\
&\exists med_1 : Bit^{\infty} \times In^{\omega}_{med1} \to Out^{\omega}_{med1} && \text{with } \overline{medium_1(dsend)} = med_1(p_1, \overline{dsend}). \\
&\exists med_2 : Bit^{\infty} \times In^{\omega}_{med2} \to Out^{\omega}_{med2} && \text{with } \overline{medium_2(asend)} = med_2(p_2, \overline{asend}).
\end{aligned}$$

$$\left.\begin{aligned}
sin &= merge\ (in,\ arecv) \wedge \\
\overline{dsend} &= idle'\ [L]\ (\overline{sin}) \wedge \\
\overline{drecv} &= med_1\ (p_1,\ \overline{dsend}) \wedge \\
\overline{out} &= rsend\ [L]\ (\overline{drecv}) \wedge \\
\overline{asend} &= rack\ [L]\ (\overline{drecv}) \wedge \\
\overline{arecv} &= med_2\ (p_2,\ \overline{asend})
\end{aligned}\right\} (2)$$

Im Gleichungssystem (2) liegen nun, bedingt durch das Auftreten von zeitabhängigen und zeitunabhängigen Basiskomponenten Gleichungen über gezeiteten und ungezeiteten Strömen vor. Aufgrund der Fixpunkteigenschaft der internen Ströme des Gleichungssystems existiert für jeden dieser internen Ströme für die Zeitabstraktion eine eindeutige Umkehrabbildung, die dem Strom ohne Zeitinformation wieder die ursprünglich vorhandene Zeitinformation zuordnet. Treffen wir beispielsweise eine Aussage über die Zeitabstraktion von *arecv* mit $\overline{arecv} = err\ \&\ rt(\overline{arecv})$, so basiert diese Aussage auf einer Zerlegung des gezeiteten Stroms *arecv* mit $\exists s_1, s_2 : arecv = s1 \circ s2\ \wedge\ \overline{s_1} = \langle err \rangle \wedge \overline{s_2} = rt(\overline{arecv})$. Dieser Zusammenhang ist für unsere nachfolgende Beweisführung von grundsätzlicher Bedeutung.

6.1.5 Beweisführung

Das zu beweisende Theorem über die Korrektheit der SDL-Spezifikation `ABP` bei Einbeziehung der beiden unzuverlässigen, aber fairen Medien haben wir bereits im vorangegangenen Abschnitt formuliert. Es lautet

Theorem:

$$\forall f \in [\![ABP]\!].\ \forall in : In_{sys}^{\infty}.\ \forall out : Out_{sys}^{\infty}.\ f(in) = out \Longrightarrow ex^*(\overline{in}) = ex^*(\overline{out})$$

Beweis:

Auf den unendlichen gezeiteten Strömen existiert kein eigenständiges Induktionsprinzip. Deshalb gehen wir wie folgt vor: Wir bilden jeden unendlichen gezeiteten Eingabestrom mit endlichem Nachrichtenanteil durch ein Induktionsmaß – die Zeitabstraktion – auf einen Strom des wohldefinierten Bereichs der endlichen Ströme ab. In diesem Bereich führen wir einen strukturellen Induktionsbeweis durch und zeigen damit, daß die Aussage für alle unendlichen gezeiteten Eingabeströme mit endlichem Nachrichtenanteil gilt. Auf Grund der Zulässigkeit der zu beweisenden Aussage und der Stetigkeit des Induktionsmaßes folgt, daß die Aussage auch für alle unendlichen gezeiteten Eingabeströme mit unendlichem Nachrichtenanteil gilt. Damit haben wir die Korrektheit des Theorems für beliebige unendliche Eingabeströme bewiesen.

Der Beweis des Theorems stützt sich auf zwei Lemmata ab, deren Beweis in Anhang B.2 zu finden ist. Die Lemmata formalisieren die Übergänge zwischen den Zuständen, die das Alternating Bit System während der Übertragung einer Nachricht einnimmt (siehe Beweisidee auf Seite 109). Um die Induktionsannahme im Induktionsschritt anwenden zu können, verallgemeinern wir das Theorem, so daß die Korrektheit des Protokolls für beliebige Anfangswerte der Kontrollbits von Sender und Empfänger gilt. Im Gleichungssystem (2) auf Seite 111 wird anstelle des Wertes L die Variable $b : Bit$ als Parameter der Funktionen $idle'$, $rack$ und $rsend$ verwendet. Diese Variable ist im äußersten Bindungsbereich der Formel universell zu quantifizieren. Das Theorem gilt dann insbesondere für die in der SDL-Spezifikation gewählte Anfangsbelegung L.

Mit den Überlegungen aus Abschnitt 6.1.4 folgern wir aus den Prämissen des Theorems die Existenz eines rekursiven Gleichungssystems $(*)$, das die Beziehung zwischen den Systemkomponenten und ihren Ein- und Ausgabeströmen charakterisiert. Auf Basis dieses Gleichungssystems werden wir die Beweisschritte durchführen.

$$\left.\begin{array}{rcl} sin & = & merge\ (in,\ arecv) \wedge \\ \overline{dsend} & = & idle'\ [b]\ (\overline{sin}) \wedge \\ \overline{drecv} & = & med_1\ (p_1,\ \overline{dsend}) \wedge \\ \overline{out} & = & rsend\ [b]\ (\overline{drecv}) \wedge \\ \overline{asend} & = & rack\ [b]\ (\overline{drecv}) \wedge \\ \overline{arecv} & = & med_2\ (p_2,\ \overline{asend}) \end{array}\right\} (*)$$

Sei $f \in [\![ABP]\!]$ im folgenden beliebig, aber fest gewählt.

Induktionsanfang:

Sei $in \in In^{\infty}_{sys}$ beliebig, aber fest mit $\overline{in} = \langle\rangle$.

$\overline{in} = \langle\rangle \Longrightarrow in = \surd^{\infty}$ und damit $merge(\surd^{\infty}, arecv) = arecv$.

Daraus folgt mit der Definition von $idle'$: $idle'\ [b]\ (\overline{arecv}) = \langle\rangle$.
es folgt mit $(*)$: $\overline{dsend} = \langle\rangle \wedge \overline{drecv} = \langle\rangle \wedge \overline{out} = \langle\rangle \wedge \overline{asend} = \langle\rangle \wedge \overline{arecv} = \langle\rangle$.

Somit: $ex^*(\overline{in}) = ex^*(\overline{out}) = \langle\rangle$. Der Induktionsanfang gilt.

Der Fixpunkt des Gleichungssystems für den leeren Eingabestrom $\overline{in}$ lautet:

$$(\surd^{\infty},\ \surd^{\infty},\ \surd^{\infty},\ \surd^{\infty},\ \surd^{\infty})$$

D.h. erhält das System keine Eingabenachrichten, so tauschen die Systemkomponenten keine Nachrichten aus und das System erzeugt keine Ausgabenachrichten.

Induktionsschritt:

Sei im folgenden $in \in In^{\infty}_{sys}$ beliebig, aber fest mit $\overline{in} = din(m)\ \&\ \overline{in'}$ und sei für in' die Induktionsannahme erfüllt.

Mit Lemma 1 existieren out', sin', $dsend'$, $drecv'$, $asend'$, $arecv'$, $p_1{}'$, $p_2{}'$ mit

$$\begin{aligned}
\overline{sin'} &= merge\ (\overline{in'},\ \overline{arecv'})\ \wedge \\
\overline{dsend'} &= ackwait'\ [\langle\rangle,\ m,\ b]\ (\overline{sin'})\ \wedge \\
\overline{drecv'} &= med_1\ (p_1{}',\ \overline{dsend'})\ \wedge \\
\overline{asend'} &= rack\ [\neg b]\ (\overline{drecv'})\ \wedge \\
\overline{out'} &= rsend\ [\neg b]\ (\overline{drecv'})\ \wedge \\
\overline{arecv'} &= med_2\ (p_2{}',\ ack(b)\ \&\ \overline{asend'}) \\
\text{und es gilt:}\quad \overline{out} &= dout(m)\ \&\ \overline{out'}
\end{aligned}$$

Daraus folgt: $ex(ft(\overline{in})) = ex(din(m)) = m = ex(dout(m)) = ex(ft(\overline{out}))$

Mit Lemma 2 existieren sin'', $dsend''$, $drecv''$, $asend''$, $arecv''$, $p_1{}''$, $p_2{}''$ mit

$$\begin{aligned}
\overline{sin''} &= merge\ (\overline{in'},\ \overline{arecv''})\ \wedge \\
\overline{dsend''} &= idle'\ [\neg b]\ (\overline{sin''})\ \wedge \\
\overline{drecv''} &= med_1\ (p_1{}'',\ \overline{dsend''})\ \wedge \\
\overline{asend''} &= rack\ [\neg b]\ (\overline{drecv''})\ \wedge \\
\overline{out'} &= rsend\ [\neg b]\ (\overline{drecv''})\ \wedge \\
\overline{arecv''} &= med_2\ (p_2{}'',\ \overline{asend''})
\end{aligned}$$

Auf dieses Gleichungssystem wenden wir nun die Induktionsannahme über in' an und erhalten $ex^*(\overline{in'}) = ex^*(\overline{out'})$.

Wir folgern:

$$ex^*(\overline{in}) = ex(din(m)) \;\&\; ex^*(\overline{in'}) = ex(dout(m)) \;\&\; ex^*(\overline{out'}) = ex^*(\overline{out})$$

Damit ist die Korrektheit des Theorems für alle $in : In_{sys}^{\infty}$ mit $\overline{in} : In_{sys}^{*}$ gezeigt. Auf Grund der Zulässigkeit der zu beweisenden Aussage (diese folgt unmittelbar aus der Stetigkeit der Funktion f) und der Stetigkeit des Induktionsmaßes folgt die Korrektheit des Theorems für alle $in : In_{sys}^{\infty}$.

6.1.6 Analyse des Beweises

Die vorangegangene Beweisführung war erfolgreich und hat den Nachweis erbracht, daß mittels unserer SDL-Semantik die Verifikation von Eigenschaften von SDL-Spezifikationen möglich ist. Wir haben für die SDL-Spezifikation des Protokolls die korrekte Datenübertragung in Verbindung mit fehlerhaften, aber fairen Übertragungsmedien nachgewiesen.

- Unsere SDL-Semantik, die in Form von FOCUS-Spezifikationen definiert ist, ist handhabbar für die Beweisführung. Die Beweisprinzipien von FOCUS konnten in unserem Fall problemlos angewendet werden. Für den Bereich der unendlichen gezeiteten Ströme konnten wir durch die Verwendung der Zeitabstraktion als Induktionsmaß einen strukturellen Induktionsbeweis auf dem reinen Nachrichtenanteil der Ströme durchführen. Wesentlich für die Beweisführung war die Verwendung der Fixpunkteigenschaft für die internen Ströme des rekursiven Gleichungssystems. Fixpunktinduktion, die in der Regel zu komplexen Beweisen führt, war nicht erforderlich.
- FOCUS bietet verschiedene semantische Modelle für verteilte Systeme. Wir haben uns bei der Definition der formalen Semantik für SDL ohne dynamische Prozeßgenerierung für ein gezeitetes Strommodell entschieden, in dem das Verhalten von Komponenten durch stark pulsgetriebene Funktionen definiert wird. Diese Wahl hat sich nun als sehr positiv für die Beweisführung erwiesen. Dank des Modells ist ein eleganter Umgang mit rekursiven Gleichungssystemen möglich, da für rekursiv definierte stark pulsgetriebene Funktionen ein eindeutiger Fixpunkt existiert. Im Vergleich zur ursprünglichen Version von ANDL mit ungezeitetem semantischen Modell entfällt damit die Forderung nach der minimalen Belegung der internen Kanäle eines Netzwerks (vgl. hierzu Abschnitt 7.2 in [SS95] und [BDD+93]).
- Aus der Strukturbeschreibung des Protokolls, die mit den ANDL-Netzwerkkomponenten gegeben ist, läßt sich schematisch ein Gleichungssystem über gezeiteten Strömen ableiten. Sind im System zeitunabhängig spezifizierte Komponenten vorhanden, so kann ein Gleichungssystem abgeleitet werden, in dem Beziehungen zwischen ungezeiteten Strömen definiert sind. Damit werden in dem Gleichungssystem Aussagen über gezeitete Ströme und ihre Zeitabstraktion, also ungezeitete Ströme, getroffen. Aufgrund der im letzten Punkt erläuterten Fixpunkteigenschaften kann im Beweis

jedoch über die Zeitabstraktion eines Stroms argumentiert werden, ohne daß dabei die eindeutige Zuordnung zum gezeiteten Strom verloren geht. Diese Vorgehensweise kann allgemein für ANDL-Spezifikationen verwendet werden.

- Die von uns verwendete Beweisidee ist intuitiv und basiert auf den unterschiedlichen Zuständen, die das Alternating Bit System während der Nachrichtenübertragungen einnimmt. Diese Systemzustände spiegeln sich sowohl in der Zusammensetzung der Ströme als auch in den Funktionen, die die einzelnen Komponenten repräsentieren, und ihren Datenzuständen wider. Die Komponenten, die aus SDL-Prozessen abgeleitet sind, werden durch wechselseitig rekursive Funktionen modelliert, wobei jede Funktion einen Zustand des Prozesses repräsentiert. Anhand dieser Zuordnung läßt sich im Gleichungssystem der Zustand der entsprechenden Prozesse ablesen – der Beweis ist damit intuitiv mit der zugrundeliegenden SDL-Spezifikation verbunden.
- Die Semantikdefinition für SDL ist konstruktiv; dies ist bedingt durch die Funktionsgleichungen, die aus den SDL-Prozessen abgeleitet werden. Dieser Spezifikationsstil hat sich bei unserer Beweisführung bewährt, da er die Anwendung der Funktionsgleichungen ähnlich wie beim Pattern Matching in funktionalen Programmiersprachen unterstützt ([BW88]). Dies ist besonders bei Fallunterscheidungen über Nachrichtenmengen günstig.

Bei unserer Beweisführung haben wir uns an der Beweisidee in [DS89] orientiert. Wenn wir die dort gegebene funktionale Spezifikation des Protokolls und die aus unserer SDL-Spezifikation resultierende FOCUS-Spezifikation (ohne die in Abschnitt 6.1.3 vorgenommenen Vereinfachungen) betrachten, so unterscheiden sich diese im wesentlichen dadurch, daß in der Spezifikation in [DS89] keine Misch- und Verteilkomponenten vorliegen. Im folgenden werden wir den Einfluß dieser Komponenten auf unsere Beweisführung untersuchen. Es wird deutlich, daß es Stellen in der Beweisführung gibt, die aufgrund SDL-spezifischer Konzepte umständlich sind und zu vermehrtem Beweisaufwand führen.

Mischen von Eingabeströmen:

Das Mischen der Eingabeströme eines SDL-Prozesses zu einem einzigen Strom in einem Eingabepuffer ist ein typisches SDL-Konzept, bei dem die Zuordnung der Eingabesignale zu den einzelnen Eingabekanälen verloren geht. Signale, für die der Prozeß in einem Zustand keinen Zustandsübergang vorsieht, werden gelöscht, sofern sie nicht mittels des Savemechanismus gerettet werden. Das Konzept des Eingabepuffers führt in der SDL-Semantikdefinition zur Einführung von *Fair Merge*-Komponenten und bei Vorliegen von Savesymbolen zu komplexen Funktionsgleichungen (siehe Seite 74). In unserer Fallstudie waren aufgrund dieses Konzepts für den Beweis der beiden Lemmata (siehe Anhang B.2) zahlreiche Fallunterscheidungen erforderlich. In den Beweisen war mehrfach zu unterscheiden, ob die erste Nachricht des Eingabestroms $\overline{sin}$ der Funktionen $idle'$ und $ackwait'$ ein Savesignal oder eine zu verarbeitende Nachricht ist.

In der in [DS89] gegebenen Spezifikation arbeiten die Komponenten explizit mit mehreren Eingabeströmen. So wird beispielsweise in der Spezifikation des Senders für das Warten auf das Bestätigungsbit eine Funktion aufgerufen, die nur den Eingabestrom liest, der vom

Empfänger kommt, und den Eingabestrom aus der Umgebung ignoriert. Damit werden Nachrichten erst gelesen, wenn sie auch verarbeitet werden können.

Das Konzept der Eingabepuffer ist eine grundlegende Eigenschaft von SDL, die nicht durch die Betrachtung getrennter Eingabeströme ersetzt werden kann. In bezug auf die Savesignale konnten wir die SDL-Semantik in der Fallstudie jedoch so vereinfachen, daß trotz der Savesymbole ein direkter Zugriff auf die jeweils erste Eingabenachricht des Stroms möglich ist (siehe die Funktion *ackwait'* in der Komponente *PR_Transmitter*).

Verteilen von Ausgabenachrichten:

In einem SDL-Prozeß können während eines Zustandsübergangs Signale an unterschiedliche Empfängerprozesse gesendet werden. In der SDL-Semantik verteilt die Komponente *Split* die Ausgabenachrichten einer Komponente *PR* auf die verschiedenen Ausgabekanäle (siehe Spezifikation von *Receiver*).

In [DS89] erfolgt die Ausgabe von Nachrichten für die verschiedenen Ausgabekanäle durch eigene Funktionen – es findet eine Entkoppelung der Ausgabeströme statt. So existieren in der Spezifikation des Empfängers zwei Funktionen, die getrennt für die beiden Ausgabekanäle die Nachrichten aus dem Eingabestrom berechnen.

Bei einfachen SDL-Prozeßgraphen kann versucht werden, die dazugehörige FOCUS-Spezifikation so umzuformen, daß getrennte Funktionen für die Ausgabekanäle vorliegen. In unserer Fallstudie ist dies bei der Komponente *Receiver* mit der Definition der Funktionen *rsend* und *rack* gelungen. Bei komplexeren Prozeßgraphen wird dies jedoch nur schwer möglich sein, so daß die Komponente *Split* nicht durch Umformungen aus der Spezifikation entfernt werden kann. In diesen Fällen muß auf der ungezeiteten Spezifikationsebene durch Projektionsfunktionen auf die einzelnen Ausgabeströme von Split zugegriffen werden. Wie bei der Komponente *FairMerge* ist dann im Gleichungssystem ein Wechsel zwischen gezeiteten und ungezeiteten Strömen erforderlich.

6.2 Verifikationsmethode für SDL-Spezifikationen

Mit der Fallstudie Alternating Bit Protokoll haben wir demonstriert, daß Verifikationsaufgaben für SDL-Spezifikationen auf Basis der von uns definierten SDL-Semantik durchführbar sind. Aus den dabei gesammelten Erfahrungen werden wir nun eine Methode für die Verifikation von SDL-Spezifikationen entwickeln.

Für die Entwicklung von nachweisbar korrekten SDL-Systemspezifikationen geben wir eine systematische Vorgehensweise an, die in mehreren Phasen den Weg von der Erstellung der SDL-Spezifikation bis zur Beweisführung beschreibt. Wir zählen diese Phasen im folgenden kurz auf und gehen anschließend auf das Vorgehen in den einzelnen Phasen ausführlich ein. Am Schluß dieses Abschnitts erläutern wir, wie sich unsere Methode durch einen rechnerbasierten Werkzeugeinsatz unterstützen läßt.

Phase 1: Entwicklung der SDL-Spezifikation

Phase 2: Angabe der formalen Semantik der SDL-Spezifikation als FOCUS-Systemspezifikation

Phase 3: Analyse und Vereinfachung der FOCUS-Spezifikation

Phase 4: Formulierung der Beweisverpflichtung

Phase 5: Beweisführung

Phase 1: Entwicklung der SDL-Spezifikation

Ausgangspunkt für die Erstellung der SDL-Spezifikation ist eine informelle Beschreibung des zu modellierenden Systems. Der Übergang von der informellen Beschreibung zur SDL-Spezifikation betrifft die Rolle von SDL als Softwareentwicklungsmethode und ist nicht Gegenstand dieser Arbeit. In der Literatur finden sich zahlreiche Arbeiten, die die frühen Phasen der Systementwicklung und den Übergang zur SDL-Spezifikation behandeln, zum Beispiel [BH93] und [SIN97].
Es ist grundsätzlich möglich, bei der Systembeschreibung das Verhalten einiger Systemkomponenten direkt in FOCUS zu definieren. Dabei ist es sinnvoll, den strukturellen Aufbau des Systems vollständig mit SDL zu modellieren, um die Kommunikationsbeziehungen zwischen den einzelnen Systemkomponenten zu erfassen. Das Verhalten der Systemkomponenten kann dann entweder als SDL-Block bzw. SDL-Prozeß oder direkt in FOCUS festgelegt werden. In der SDL-Semantik erfolgt anschließend die Integration der unterschiedlichen Verhaltensspezifikationen (siehe Phase 2). Dieses Vorgehen bietet sich an, um Hardwarekomponenten, die nicht mit SDL spezifiziert werden, in die Systemmodellierung mit SDL einzubeziehen oder um Verhaltensaspekte zu definieren, für die in SDL keine adäquaten Sprachmittel zur Verfügung stehen (zum Beispiel Fairneßbedingungen).

Phase 2: Angabe der formalen Semantik der SDL-Spezifikation als Focus-Systemspezifikation

Für die in Phase 1 erstellte SDL-Spezifikation wird die formale Semantik angegeben. Dazu wird aus der SDL-Spezifikation eine FOCUS-Spezifikation abgeleitet, die das Verhalten der SDL-Spezifikation eindeutig definiert. Das Vorgehen für die Umsetzung von SDL nach FOCUS haben wir in Kapitel 4 ausführlich vorgestellt. Es läßt sich grob durch folgende Schritte beschreiben:

- Die Struktur der SDL-Spezifikation wird durch ein ANDL-Netzwerk beschrieben. Für die Systemebene sowie für jede Blockebene werden ANDL-Komponenten angegeben. Die Nachrichtentypen für die Kommunikationsverbindungen werden aus den Kanaltypen der SDL-Spezifikation abgeleitet. Dieses Vorgehen erfolgt schematisch, wie in Abschnitt 4.1.5 vorgestellt.
- Für die Fundierung des Datenanteils der SDL-Signale und der lokalen Datenräume der SDL-Prozesse werden die SDL-Datentypen auf abstrakte Datentypen aus SPEC-

TRUM abgebildet. Hierfür steht in SPECTRUM eine umfangreiche Bibliothek von Standarddatentypen zur Verfügung.

- Auf der untersten Strukturierungsebene besteht eine SDL-Spezifikation aus SDL-Prozessen. Diese werden in der Semantikdefinition in ANDL-Netzwerke umgesetzt. Dabei stehen für die in sich abgeschlossenen Verhaltensanteile der Prozesse, wie etwa das Mischen von Eingabenachrichten oder die Timerüberwachung, Semantikbausteine zur Verfügung, so daß sich die ANDL-Netzwerke schematisch zusammensetzen lassen. Das eigentliche Ein-/Ausgabeverhalten eines Prozesses, das durch den Prozeßgraphen als Zustandsübergangsautomat definiert ist, ist dagegen für jeden Prozeß individuell anzugeben. Dazu werden für die Prozeßzustände Funktionen definiert und aus den Zustandsübergängen des Prozesses Funktionsgleichungen abgeleitet.
- Liegen in der SDL-Spezifikation Komponenten vor, deren Verhalten mit FOCUS definiert ist, so werden diese Verhaltensdefinitionen anstelle der Block- bzw. Prozeßkomponenten in die SDL-Semantik eingebunden.

Phase 3: Analyse und Vereinfachung der Focus-Spezifikation

Als Resultat von Phase 2 liegt eine FOCUS-Spezifikation vor, die das Verhalten der SDL-Spezifikation auf formale und damit eindeutige Weise beschreibt. Eine Analyse dieser FOCUS-Spezifikation kann Spezifikationsteile identifizieren, die vereinfacht werden können. Dies trägt zu einer einfacheren Beweisführung bei.
Möglichkeiten für Vereinfachungen sind beispielsweise das Entfernen überflüssiger Parameter in Funktionen oder die Entkopplung der Ausgabenachrichten von Prozeßkomponenten ohne die Verwendung von Verteilkomponenten.
Es ist sicherzustellen, daß die vereinfachte Spezifikation das gleiche Verhalten wie die ursprüngliche Spezifikation aufweist. Dies ist formal zu beweisen oder durch die Anwendung korrektheitserhaltender Transformationsschritte zu gewährleisten.

Phase 4: Formulierung der Beweisverpflichtung

Ausgehend von der informellen Beschreibung des zu modellierenden Systems werden Eigenschaften formuliert, die das korrekte Verhalten des Systems charakterisieren. Die Eigenschaften können in einem ersten Schritt informell beschrieben werden, bevor sie in eine prädikatenlogische Formel umgesetzt werden.

Eigenschaften, die das mit SDL spezifizierte System S erfüllen soll, werden in FOCUS als Prädikat über der SDL-Semantik formuliert. Es wird gefordert, daß jede Funktion f, die ein zulässiges Ein-/Ausgabeverhalten der SDL-Spezifikation darstellt, die im Prädikat P festgelegten Eigenschaften erfüllt. Diese Forderung wird in eine Implikation umgesetzt und stellt die Beweisverpflichtung für die in Phase 5 folgende Beweisführung dar:

$$\forall f \in [\![S]\!].\ \forall in, out.\ f(in) = out \Longrightarrow P(in, out)$$

Die Formulierung einer gewünschten Eigenschaft mit SDL erscheint wenig sinnvoll, da SDL keine Möglichkeit für die Spezifikation von abstraktem Ein-/Ausgabeverhalten vorsieht.

Es bietet sich an, für die Formulierung der Eigenschaften graphische Beschreibungsmittel wie etwa Message Sequence Charts ([IT96]) zu verwenden und aus der graphischen Beschreibung die formale Beweisverpflichtung abzuleiten. Dieses Vorgehen stellt eine mögliche Fortführung unserer Arbeit dar (siehe auch Abschnitt 7.2) und setzt eine eindeutige formale Semantik für Message Sequence Charts voraus.

Phase 5: Beweisführung

Die Beweisführung hat zum Ziel, mit mathematischen Mitteln zu überprüfen, ob die SDL-Spezifikation die in Phase 4 formulierte Beweisverpflichtung erfüllt. Der wesentliche und kreative Aspekt bei der Beweisführung ist das Finden der Beweisidee mit der Wahl der dazu passenden Beweistechnik. Hier spielt die Erfahrung mit der Verifikation von verteilten Systemen eine wichtige Rolle; ein genaues Verständnis des Verhaltens und der Kooperation der Systemkomponenten sind Voraussetzung für das Finden der Beweisidee und die anschließende Beweisführung.

Für die Beweisführung stehen in FOCUS die Beweistechniken der klassischen Prädikatenlogik höherer Stufe, der funktionalen Logik und der Bereichstheorie zur Verfügung. Durch den funktionalen Stil der SDL-Semantik lassen sich diese ohne spezielle Anpassungen an SDL-spezifische Gegebenheiten anwenden. Somit kann die Mächtigkeit von FOCUS voll ausgeschöpft werden. Sind mehrere Verifikationsaufgaben für SDL durchgeführt worden, so können diese hinsichtlich der verwendeten Beweisideen und Beweistechniken analysiert werden. Daraus lassen sich möglicherweise Richtlinien für die Verifikation von bestimmten Klassen von SDL-Spezifikationen (zum Beispiel für Kommunikationsprotokolle) angeben.

Mit SDL werden überwiegend verteilte Systeme mit einem hohen Anteil interner Kommunikation spezifiziert. Um über die Kooperation der Systemkomponenten, die mittels Nachrichtenaustausch erfolgt, argumentieren zu können, wird, wie in Abschnitt 6.1.4 erläutert, aus der SDL-Semantikdefinition ein Gleichungssystem über den internen Strömen und den Ein- und Ausgabeströmen des Systems abgeleitet. Sofern es sich dabei um ein rekursives Gleichungssystem handelt, besitzt dieses einen eindeutigen Fixpunkt, der sich aus den Strömen der internen Kanäle zusammensetzt. Bei der Beweisführung ist dadurch der Bezug auf Fixpunkteigenschaften möglich.

Aus dem Gleichungssystem läßt sich ein globaler Zustand des Systems ableiten. Dieser setzt sich zum einen aus den lokalen Datenzuständen der einzelnen Komponenten, die im Gleichungssystem durch stromverarbeitende Funktionen repräsentiert werden, und zum anderen aus den aktuellen Strombelegungen der Kanäle zusammen. Die Komponenten, die das Verhalten von SDL-Prozeßgraphen modellieren, sind durch wechselseitig rekursive Funktionen definiert. Da die Bezeichner für diese Funktionen den Zustandsnamen aus den SDL-Prozessen entsprechen, läßt sich aus dem Gleichungssystem anhand der Funktionsnamen ablesen, in welchem Kontrollzustand sich die SDL-Prozesse befinden. Dieser Bezug zur SDL-Spezifikation ist hilfreich, um den Systemzustand zu bestimmen. Wie unsere Fallstudie gezeigt hat, ist für die Durchführung von strukturellen Induktionsbeweisen über den Eingabeströmen eines Systems das Verwenden eines Systemzustands hilfreich.

Ist die Beweisführung erfolgreich, so ist sichergestellt, daß die in Phase 1 entwickelte SDL-Spezifikation die mittels der Beweisverpflichtung geforderten Eigenschaften erfüllt. Schlägt die Beweisführung dagegen fehl, so ist die SDL-Spezifikation aus Phase 1 zu verbessern, die SDL-Semantik neu zu bestimmen und erneut Phase 5 durchzuführen.

Werkzeugunterstützung für die Methode

Für die Akzeptanz und Praktikabilität der von uns entwickelten Verifikationsmethode ist ein werkzeuggestütztes Vorgehen unerläßlich. Im folgenden erläutern wir, für welche Phasen rechnerbasierte Werkzeugunterstützung zur Verfügung steht.

In Phase 1 kann die SDL-Spezifikation mit den graphischen Editoren kommerzieller SDL-Case-Werkzeuge erstellt werden. Darüber hinaus bieten diese Case-Werkzeuge die Möglichkeit, die syntaktische Korrektheit der erstellten SDL-Spezifikation gemäß den Richtlinien der Z.100 automatisch überprüfen zu lassen.

Die in Phase 2 erforderliche Abbildung der SDL-Spezifikation nach FOCUS kann automatisch erfolgen. Dazu wird aus der textuellen Darstellung der SDL-Spezifikation, die sich mittels der Case-Werkzeuge generieren läßt, durch einen Umsetzer die FOCUS-Spezifikation erzeugt. Eine prototypische Implementierung des Umsetzers liegt vor und ist in [Hin96b] beschrieben (siehe auch Abschnitt 4.4).

Die Beweise, die in Phase 5 zu führen sind, sind erfahrungsgemäß komplex und umfangreich, so daß der Einsatz eines Theorembeweisers wünschenswert ist. Ziel eines solchen Einsatzes ist es, den Beweisführenden von langwierigen und mühsamen, routineartigen Beweisschritten zu befreien, so daß er sich auf den kreativen Anteil der Beweisführung konzentrieren kann. Für FOCUS steht eine Beweisunterstützung durch den interaktiven Theorembeweiser Isabelle zur Verfügung. Voraussetzung ist, daß die FOCUS-Spezifikationen in HOLCF, einer Objektlogik von Isabelle, formalisiert werden. Für FOCUS-Spezifikationen, die mit ANDL erstellt sind, wird im Rahmen des Projektes SFB 342/A6 ein Prototyp für die automatische Umsetzung nach HOLCF erstellt. Diese Implementierung läßt sich so erweitern, daß auch die Umsetzung der von uns definierten SDL-Semantik nach HOLCF automatisch erfolgen kann. Allerdings reicht die bisher durchgeführte Formalisierung von FOCUS in HOLCF nicht aus, um die SDL-Semantik in HOLCF zu formalisieren. Im nächsten Abschnitt werden wir deshalb ein Konzept für die Formalisierung der SDL-Semantik in HOLCF vorstellen und damit die Voraussetzungen für die werkzeugunterstützte Beweisführung für SDL-Spezifikationen schaffen.

Nach erfolgreicher Durchführung von Phase 5 kann unter Verwendung von SDL-Case-Werkzeugen der Zielcode aus der SDL-Spezifikation automatisch generiert werden.

Zusammenfassend halten wir fest, daß sich die von uns entwickelte Methode im hohen Maße durch Werkzeugunterstützung automatisieren läßt.

6.3 Formalisierung der SDL-Semantik in HOLCF

In diesem Abschnitt stellen wir ein Konzept für die Formalisierung der SDL-Semantik in der Logik HOLCF des interaktiven Theorembeweisers Isabelle vor. HOLCF ist eine Logik höherer Stufe, die um die Konzepte der Bereichstheorie erweitert ist. Eine Beschreibung von dieser Logik findet sich in [Reg95a, NMSvO98]. Dieser Abschnitt wendet sich an Leser, die an der Verifikation von SDL-Spezifikationen mit Isabelle interessiert sind und dazu eine Übertragung der SDL-Semantik nach HOLCF vornehmen wollen. Wir setzen voraus, daß diese Leser mit den Verifikationstechniken und den Objektlogiken HOL und HOLCF von Isabelle vertraut sind, da eine Einführung in diese Thematik nicht Zielsetzung dieser Arbeit ist.

Wir haben die SDL-Semantik mit ANDL, der Kernsprache von FOCUS, definiert. Für ANDL-Spezifikationen mit ungezeitetem semantischen Modell existiert bereits eine Anbindung an HOLCF, die in [SS95] beschrieben ist. Für die Umsetzung der von uns um Zeitaspekte erweiterten Sprache ANDL ist es erforderlich, in HOLCF ein gezeitetes semantisches Strommodell zu definieren. Dazu zählen in erster Linie gezeitete unendliche Ströme (sog. vollständige Kommunikationsgeschichten) sowie pulsgetriebene Funktionen.

Zunächst beschreiben wir die Einführung des gezeiteten Strommodells in HOLCF. Anschließend stellen wir vor, wie sich die Semantik von SDL-Spezifikationen auf dieser Basis in HOLCF definieren läßt. Dabei gehen wir auf folgende Aspekte genauer ein:

- Definition gezeiteter und unendlicher Ströme
- Definition von pulsgetriebenen Funktionen
- Definition von Nachrichtenmengen
- Definition von Netzwerkkomponenten
- Definition von gezeiteten und ungezeiteten Basiskomponenten

Diese Aspekte werden wir am Beispiel der Semantik der SDL-Spezifikation des Alternating Bit Protokolls verdeutlichen.

6.3.1 Einführung des gezeiteten Strommodells in HOLCF

Für die Einführung des gezeiteten Strommodells in HOLCF definieren wir in Abschnitt 6.3.1.1 unendliche gezeitete Ströme (vollständige Kommunikationsgeschichten) und in Abschnitt 6.3.1.2 pulsgetriebene Funktionen. In Abschnitt 6.3.1.3 stellen wir die Definition von Funktionen auf gezeiteten und unendlichen Strömen in diesem Modell vor.

6.3.1.1 Unendliche gezeitete Ströme

Die semantische Basis der um Zeitaspekte erweiterten Sprache ANDL basiert auf stark pulsgetriebenen Funktionen, die unendliche gezeitete Ströme auf unendliche gezeitete Ströme abbilden (siehe dazu auch Abschnitt 3.2.2).

In HOLCF ist bereits der polymorphe Datentyp `'a stream` enthalten, der partielle oder unendliche Ströme über einem beliebigen Typ der Klasse `pcpo` modelliert:

```
domain 'a stream = && (ft::'a) (lazy rt::'a stream)
```

`'a` ist eine Typvariable, die einen beliebigen Typ aus der Klasse `pcpo` bezeichnet[3]. So bildet `dnat stream` zum Beispiel Ströme über den natürlichen Zahlen. Ein Strom besteht aus dem ersten Element, auf das mit `ft` zugegriffen wird, und dem restlichen Strom, der über `rt` angesprochen wird. Die Funktion `&&` (& in FOCUS) beschreibt das Voranstellen eines Elements vor einen Strom und bildet aus einem Element `d` der Sorte `'a` und einem Strom `s` der Sorte `'a stream` den Strom `d && s`. Der leere Strom wird mit $\langle\rangle$ bezeichnet und intern auf $\bot$ abgebildet.

Zusätzlich stehen zur Verfügung: die Funktion `slen` für die Länge eines Stroms, die Filterfunktion `slfilter` und das Prädikat `stream_finite`, das prüft, ob ein Strom endlich oder unendlich ist.

Für Ströme, deren Elemente geliftete HOL-Objekte und somit flach geordnet sind[4], ist der Typ `'b fstream` definiert:

```
types 'b::term fstream = ('b lift) stream
```

Das Voranstellen eines Elements `d::term` vor einen Strom über gelifteten HOL-Elementen wird durch die Funktion $\rightsquigarrow$ beschrieben.

Um in der Menge aller Ströme die Menge der unendlichen Ströme auszuzeichnen, führen wir mittels des Konstruktors `subdom` aus der ADT-Library von HOLCF den Typ `'a istream` für unendliche Ströme ein (siehe [Slo97] für eine detaillierte Beschreibung von `subdom` und der ADT-Library).

Zunächst erfolgt nach der Methodik von [Slo97] eine Einbettung des Typs `'a stream` in den domain `IS`.

```
domain 'a IS = ISabs (ISrep::'a stream)
```

Das Prädikat `adm_pred'`, das die Menge aller unendlichen Ströme charakterisiert, wird mittels Negation der Funktion `stream_finite` definiert und stellt ein zulässiges Prädikat dar:

```
adm_ISadm_pred
(adm_pred':: 'a IS ⇒ bool) ≡ (λ s. ¬ stream_finite (ISrep‘s))
```

[3]Soweit nicht anders angegeben, sind im folgenden alle Typvariablen aus der Klasse `pcpo`.
[4]HOL ist eine Objektlogik höherer Stufe in Isabelle.

Schließlich werden die Instanz von `IS` als zulässiger `pcpo`-Typ bewiesen und daraus der Typ der unendlichen Ströme `'a istream` definiert (siehe [Wen97] für Beweise von Instanzen):

```
instance IS :: (pcpo)adm {| (rtac adm_ISadm_pred 1) |}
types 'a istream = 'a IS subdom
```

Für die Modellierung des diskreten, globalen Zeitbegriffs führen wir den Typ der gezeiteten Ströme `'a tstream` als Ströme über endlichen Listen ein, wobei jede Liste die Nachrichten eines Zeitintervalls enthält. Die Modellierung von gezeiteten Strömen als Ströme über Listen ist isomorph zu der in Abschnitt 3.1 gewählten Verwendung von Zeitticks $\surd$ zur Abgrenzung der Zeitintervalle. Leere Listen stellen somit Zeitintervalle dar, in denen keine Nachricht gesendet wird. Für die endlichen Listen verwenden wir den polymorphen HOLCF-Typ `'a dlist`:

```
types 'a tstream = ('a dlist) stream
```

mit

```
domain 'a dlist = dnil | ## (dhd::'a) (dtl::'a dlist)
```

Somit können nun für die unterschiedlichen Stromtypen aus der FOCUS-Theorie folgende Typen der Klasse `pcpo` in HOLCF definiert werden:

ungezeitete Ströme (N^{ω}):	`'a stream`
gezeitete Ströme ($N^{\underline{\omega}}$):	`'a tstream = ('a dlist) stream`
unendliche ungezeitete Ströme (N^{∞}):	`'a istream`
unendliche gezeitete Ströme ($N^{\underline{\infty}}$):	`'a itstream = ('a dlist) istream`

Die Menge der endlichen Ströme kann nicht als eigener `pcpo`-Typ definiert werden, da das entsprechende Prädikat nicht zulässig ist und somit die Voraussetzungen für die Anwendung des `subdom`-Konstruktors nicht erfüllt sind.
Die eben aufgeführten Stromtypen lassen sich auch für Ströme mit gelifteten HOL-Elementen definieren. Sei `'b` eine Typvariable der Klasse `term`; `'a list` ist der Datentyp für endliche Listen in HOL:

flache ungezeitete Ströme:	`'b fstream`
flache gezeitete Ströme:	`'b ftstream = ('b list lift) stream`
unendliche flache ungezeitete Ströme:	`'b fistream = ('b lift) istream`
unendliche flache gezeitete Ströme:	`'b fitstream = ('b list lift) istream`

Ob eine Modellierung mit allgemeinen Strömen oder mit Strömen über gelifteten HOL-Elementen gewählt wird, hängt von den Strom- und Datentypen der jeweiligen FOCUS-Spezifikation ab, die in HOLCF zu formalisieren ist.

6.3.1.2 Pulsgetriebene Funktionen

Für die Definition pulsgetriebener Funktionen ist die Einführung einer Präfixfunktion `prefix` auf unendlichen Strömen notwendig. Diese liefert für eine endliche natürliche Zahl

n und einen unendlichen Strom s das Präfix des Stroms bis einschließlich des n-ten Stromelementes[5].

```
domain dnat = dzero | dsucc (dpred::dnat)
prefix :: dnat → 'a istream → 'a stream        (↓)
↓ ≡ fix‘(Λ h n s.
         If is_dzero‘n then ⊥ else Ift‘s && h‘(dpred‘n)‘(Irt‘s) fi)[6]
```

Mit dieser Definition läßt sich nun die Eigenschaft der starken Pulsgetriebenheit als Prädikat `is_spd` über stromverarbeitenden Funktionen definieren. Um das Prädikat sowohl auf Funktionen des Typs `'a itstream → 'b itstream` als auch auf Funktionen des Typs `'c::term fitstream → 'd::term fitstream` anwenden zu können, wird es nicht auf vollständigen Kommunikationsgeschichten, sondern allgemeiner auf unendlichen Strömen definiert. Dies erleichtert die nachfolgende Einführung von Subdomains für pulsgetriebene Funktionen. Analog dazu läßt sich auch das Prädikat für die schwache Pulsgetriebenheit definieren, auf das wir hier nicht näher eingehen.

```
is_spd :: ('a istream → 'b istream) ⇒ bool
is_spd f ≡ (! s t.(! j. s↓j = t↓j --> (f‘s)↓(dsucc‘j) = (f‘t)↓(dsucc‘j)))
```

Für die Menge aller stark pulsgetriebenen Funktionen führen wir eigenständige Typen mittels des Konstruktors `subdom` ein, wobei als charakterisierendes Prädikat das eben eingeführte Prädikat `is_spd` dient.

- Der Typ `'a` $\rightarrowtail$ `'b` bezeichnet alle stark pulsgetriebenen Funktionen auf allgemeinen Strömen: `'a itstream → 'b itstream`.
- Der Typ `'c::term` $\rightsquigarrow$ `'d::term` bezeichnet alle stark pulsgetriebenen Funktionen auf Strömen mit gelifteten HOL-Elementen: `'c::term fitstream → 'd fitstream`.

Damit ist eine sehr kompakte Darstellungsweise für stark pulsgetriebene Funktionen gegeben, bei der nur die Typen der Stromelemente anzugeben sind. Die Typdefinitionen sind noch für Funktionen auf Tupeln von Strömen zu erweitern. Der Einfachheit halber verwenden wir die Typsymbole $\rightarrowtail$ und $\rightsquigarrow$ im folgenden auch für Funktionen mit Tupeln von Ein- bzw. Ausgabeströmen.

Um die Verwendung von stark pulsgetriebenen Funktionen zu vereinfachen, definieren wir eigene Funktionsapplikationen für stark pulsgetriebene Funktionen. Damit verbergen wir die Anwendung der Repräsentationsfunktionen `Frep` und `Rep_Sd`. Als Beispiel folgt die Applikation ‘‘ für stark pulsgetriebene Funktionen über Strömen mit gelifteten HOL-Elementen:

```
pdfapp :: ('a ⇝ 'b) ⇒ ('a fitstream ⇒ 'b fitstream)    _‘‘_
f‘‘e ≡ Frep‘(Rep_Sd f)‘e
```

[5] Im Gegensatz zur Funktion `stream_take` handelt es sich bei `prefix` um eine stetige Funktion, was die nachfolgende Einführung des Subdomains für stark pulsgetriebene Funktionen erleichtert.

[6] Die Funktionen `Ift` und `Irt` liefern – angewandt auf einen unendlichen Strom – das erste Element bzw. den restlichen Strom (siehe Abschnitt 6.3.1.3).

6.3.1.3 Funktionen auf gezeiteten und unendlichen Strömen

Die Funktionen auf den Strömen des Typs `'a stream` können nicht auf Ströme des Typs `'a istream` angewandt werden. Für die neu eingeführten Typen von Strömen ist deshalb die Definition einer Reihe von Funktionen erforderlich, wobei auf vorhandenen Stromfunktionen mittels Liftfunktionen aufgesetzt wird. Einige Funktionen auf dem Typ der unendlichen Ströme finden sich bereits in [Slo97]:

```
Ift         :: 'a istream → 'a
Irt         :: 'a istream → 'a istream
Icons       :: 'a → 'a istream → 'a istream       (&&&)
ith         :: dnat → 'a istream → 'a
imap        :: ('a → 'b) → 'a istream  'b istream

Ift         ≡ Λ x.ft‘(ISrep‘(Rep_Sd x))
Irt         ≡ sd_lift(IS_lift‘rt)
&&&         ≡ Λx.sd_lift(IS_lift‘(s.Scons‘x‘s))
ith         ≡ fix‘(Λith.Λn.Λs.
                 If is_dzero‘n then Ift‘s else ith‘(dpred‘n)‘(Irt‘s) fi)
imap        ≡ Λf. sd_lift(IS_lift‘(smap‘f))
```

Die Funktion `Ift` liefert das erste Element eines unendlichen Stroms, `Irt` den restlichen Strom. Die Funktion `Icons` fügt ein Element an den Anfang eines unendlichen Stroms an, `ith‘n` liefert das `n`-te Element eines unendlichen Stroms, `imap‘f` wendet die Funktion `f` auf alle Elemente eines unendlichen Stroms an. Diese Funktionen lassen sich für alle mit `istream` konstruierten Typen (d.h. auf `'a::pcpo istream`, `'b::term fistream`, `'a::pcpo itstream` und `'b::term fitstream`) verwenden.

Stromverarbeitende Funktionen, die bei der Definition der SDL-Semantik eine wesentliche Rolle spielen, sind die Zeitabstraktion und Filterfunktionen. Bei diesen Funktionen sind unterschiedliche Definitionen für allgemeine Ströme und für Ströme mit gelifteten HOL-Elementen erforderlich, da bei diesen Funktionen auf den Inhalt der einzelnen Listenelemente zugegriffen wird und dabei zwischen HOL- und HOLCF-Listen zu unterscheiden ist.

Operatoren für die Zeitabstraktion sind `irmtck` für allgemeine, unendliche gezeitete Ströme und `firmtck` für unendliche gezeitete Ströme mit gelifteten HOL-Elementen. Beide entfernen die Zeitinformation aus den Strömen, indem sie die Listenstruktur der Ströme auflösen und aus den einzelnen Elementen der Listen einen Strom bilden.

```
irmtck      :: 'a itstream → 'a stream

irmtck ≡ fix‘(Λh s. (case Ift‘s of dnil ⇒ h‘(Irt‘s)
                          | d##dl ⇒ (d && h‘(dl &&& Irt‘s))))


firmtck     :: 'b::term fitstream → 'b fstream
```

```
firmtck ≡
fix‘(Λh s. (case Ift‘s of Undef ⇒ ⊥
                        | Def l ⇒ (case l of [] ⇒ h‘(Irt‘s)
                                           | e#rl ⇒ (Def e&&
                                                 h‘((Def rl)&&&Irt‘s)))))
```

Die Filterfunktion auf allgemeinen gezeiteten Strömen verwendet als erstes Argument eine berechenbare, wahrheitswertige Funktion, um die Menge der zu filternden Elemente zu bestimmen; die Filterfunktion auf gezeiteten Strömen mit gelifteten HOL-Elementen verwendet hierfür direkt die Mengendarstellung über `'b::term set`.

```
dfilter     :: ('a → tr) → 'a dlist → 'a dlist
ifilter     :: ('a → tr) → 'a itstream → 'a itstream
ifsfilter   :: 'b::term set ⇒ 'b fitstream → 'b fitstream

dfilter ≡
fix‘(Λh f l. case l of dnil ⇒ dnil
                     | d##dl ⇒ If f‘d then d##(h‘f‘dl) else h‘f‘dl fi)

ifilter ≡ Λ h.imap‘(dfilter‘h)

ifsfilter A ≡
imap‘(Λ se. (case se of Undef ⇒ ⊥
                        | Def l ⇒ Def (filter (λ x. x ∈ A) l)))
```

Für Ströme des Typs `'a istream` gilt, daß keines der Stromelemente gleich $\bot$ ist. Andernfalls würde sich ein partieller Strom ergeben, was der Typisierung widerspricht. Bei den Definitionen der Funktionen `firmtck` und `fisfilter` erfolgt jedoch beim Zugriff auf die einzelnen Stromelemente eine Fallunterscheidung, ob es sich um ein undefiniertes (`Undef`) oder ein definiertes Element (`Def l`) handelt. Diese Fallunterscheidung ist aufgrund des Liftens der HOL-Elemente erforderlich. Ersterer Fall läßt sich in den Beweisen durch Widerspruch lösen, so daß das in den Definitionen vorgegebene Verhalten für das weitere Vorgehen ohne Belang ist.

6.3.2 Definition von Nachrichtenmengen

Für jeden Kanal in einer ANDL-Spezifikation muß die Menge von Nachrichten angegeben werden, die über diesen Kanal gesendet werden dürfen (Typisierung der Kanäle). Dabei können die Nachrichten mit Parametern versehen werden.

Beispiel einer Nachrichtenmenge in ANDL:

$$Message = \{m1,\ m2,\ m3\ (d,\ e) \mid d,\ e : Nat\}$$

In HOLCF werden Typen für Mengen von Nachrichten über das `datatype`-Konstrukt definiert. Das oben gegebene Beispiel einer Nachrichtenmenge entspricht in HOLCF folgender Typvereinbarung:

```
datatype Message = m1 | m2 | m3 (d::dnat) (e::dnat)
```

Dabei sind die Typen der Parameter d und e aus der Klasse `pcpo`, da es sich um Datentypen der Bereichstheorie handelt. Die Bezeichner der Nachrichten $m1$, $m2$, $m3$ sind aus der Klasse `term`, da es sich dabei um rein syntaktische Information ohne Berechnungsanteil handelt. Die Bezeichner der einzelnen Nachrichten übernehmen die Rolle von Konstruktoren für den Typ `Message`. Auf die Parameter der Nachricht `m3` wird über die Selektoren `d` bzw. `e` zugegriffen.

Signalparameter in SDL brauchen nicht zwingend belegt zu sein. In der Semantikdefinition wurden deshalb erweiterte Datentypen für Nachrichtenmengen eingeführt, bei denen nicht belegte Parameter mit dem Symbol ϕ bezeichnet sind (siehe Seite 70). Somit lautet die erweiterte Typvereinbarung für Parameter von Signalen in HOLCF:

```
datatype 'a EXT = par 'a | ϕ
```

Liegt eine Spezifikation mit Signalen vor, deren Parameter nicht durchgängig belegt sind, so ist anstelle des einfachen Typs für die Parameter der erweiterte Typ zu verwenden. Obige Mengenvereinbarung lautet für diesen Fall:

```
datatype Message = m1 | m2 | m3 (d::dnat EXT) (e::dnat EXT)
```

Die Nachrichtenmengen des **Alternating Bit Protokolls** werden nun wie folgt umgesetzt:

```
domain bit = L | H      mit   neg :: bit → bit      (strict)
neg‘L = H   und   neg‘H = L

types 'a sig_par = 'a * bit

types data = dnat

datatype IN     = din (data)
         OUT    = dout (data)
         DACK   = snd (data sig_par)
         EDACK  = esnd (data sig_par) | err1
         ACK    = ack (bit)
         EACK   = eack (bit) | err2
```

Eine direkte Umsetzung der SPECTRUM-Datentypen aus der SDL-Semantikdefinition nach HOLCF ist nicht möglich. Dies liegt daran, daß in HOLCF die Bezeichner der Konstruktor- und Selektorfunktionen eindeutig sein müssen. Die Bezeichner aus den SPECTRUM-Datentypen für die Nachrichtenmengen sind deshalb teilweise umzubenennen. In unserem Beispiel ist dies bei den Ein- und Ausgabenachrichten der Medien der Fall. So ist beispielsweise die Nachrichtenmenge `EDACK` gleich der Nachrichtenmenge `DACK` vereinigt mit der Nachricht `err1`.
Für den Typ `bit` der alternierenden Bits ist die Definition als `pcpo`-Typ erforderlich, da die alternierenden Bits auch Bestandteil des Datenzustands der SDL-Prozesse sind und dafür partielle Datentypen erforderlich sind.

6.3.3 Definition von Netzwerkkomponenten

Anhand der Netzwerkkomponente *ABP* aus Abschnitt 6.1.2 erläutern wir, wie die Spezifikation eines Netzwerks in HOLCF formalisiert wird. Bei der Formalisierung wird von der Darstellung der Netzwerkkomponente als Prädikat über stark pulsgetriebenen Funktionen ausgegangen, wie wir es in Anhang B.1 aufgeführt haben. Die Funktionalität des Prädikats für die Netzwerkkomponente *ABP* in HOLCF lautet wie folgt:

```
is_abp:: (IN ⇝ OUT ) ⇒ bool
```

Mit diesem Prädikat wird eine Menge von stark pulsgetriebenen Funktionen definiert, die einen Eingabestrom des Nachrichtentyps `IN` erhalten und einen Ausgabestrom des Nachrichtentyps `OUT` liefern. Aufgrund des von uns definierten eigenständigen Typs für stark pulsgetriebene Funktionen (vgl. Abschnitt 6.3.1.2) ergibt sich eine kompakte Darstellung der Funktionalität des Prädikats, bei der – ähnlich zur Darstellung in ANDL – die Betonung auf den Nachrichtentypen der Ein- und Ausgabekanäle liegt. Die in der ungezeiteten Version der Sprache ANDL geforderte Eigenschaft, daß die Belegung der internen Kanäle des Komponentennetzwerks minimal ist, entfällt in der gezeiteten Version von ANDL, da auf der semantischen Ebene nur über vollständige Kommunikationsgeschichten argumentiert wird.

Die Teilkomponenten werden ebenfalls über Prädikate charakterisiert, deren Funktionalitäten wie folgt lauten:

```
is_transmitter :: ((IN * EACK) ⇝ DACK) ⇒ bool
is_med1 ::        (DACK ⇝ EDACK ) ⇒ bool
is_receiver ::    (EDACK ⇝ (OUT * ACK)) ⇒ bool
is_med2 ::        (ACK ⇝ EACK) ⇒ bool
```

Dabei bezeichnet der HOLCF-Konstruktor $*$ den kartesischen Produkttyp, mit dem sich Tupel von Ein- bzw. Ausgabeströmen als Paare von Strömen modellieren lassen. Auf die einzelnen Komponenten eines Stromtupels <`s1, s2`> wird mit den HOLCF-Operationen `cfst` und `csnd` zugegriffen.

Die nachfolgende Definition des Prädikats `is_abp` setzt sich aus zwei Teilen zusammen:

1. Für die Beschreibung der Struktur des Netzwerkes wird angegeben, aus welchen Teilkomponenten sich das Netzwerk zusammensetzt. Es wird gefordert, daß für jede Teilkomponente eine Funktion existiert, die ein gültiges Verhalten der jeweiligen Teilkomponente definiert.

2. Anschließend erfolgt die Angabe des Netzwerkgleichungssystems. Es wird gefordert, daß für jede beliebige Belegung der Ein- und Ausgabekanäle eine Belegung der internen Kanäle existiert, die das gesamte Gleichungssystem erfüllt.

```
is_abp ABP ≡ ∃ S R M1 M2.(
1     is_transmitter S ∧ is_receiver R ∧ is_med1 M1 ∧ is_med2 M2 ∧
2     ∀ din dout. ABP‘‘din = dout →
        ∃ ds dr as ar.  S‘‘<din,ar>  =  ds ∧
                        M1‘‘ds  =  dr ∧
                         R‘‘dr  =  <dout,as> ∧
                        M2‘‘as  =  ar )
```

Die Darstellung des Prädikats in HOLCF unterscheidet sich nur gering von der Darstellung in der Sprache ANDL. Dazu trägt wesentlich die Funktionsapplikation ‘‘ für stark pulsgetriebene Funktionen bei, die wir in Abschnitt 6.3.1.2 eingeführt haben.

6.3.4 Definition von Basiskomponenten

Am Beispiel der Komponente *PR_Receiver* aus Abschnitt 6.1.2 erläutern wir, wie eine in ANDL zeitunabhängig definierte Basiskomponente nach HOLCF umgesetzt wird.

Wir definieren das charakterisierende Prädikat

```
is_pr_receiver :: (EDACK ⇝ (OUT * ACK)) ⇒ bool
```

Die Definition des Prädikats `is_pr_receiver` stützt sich auf das Prädikat `receiver` ab:

```
is_pr_receiver f ≡ ∀ in out. f‘‘in = out → receiver in out
```

Jede Funktion `f`, die ein Verhalten der Komponente *PR_Receiver* beschreibt, muß die im Prädikat `receiver` festgelegte Eigenschaftsbeschreibung erfüllen.

```
receiver :: [EDACK fitstream, (OUT fitstream * ACK fitstream)] ⇒ bool
```

Das Prädikat `receiver` enthält die Funktionsgleichungen, die wir aus dem SDL-Prozeßgraphen für die einzelnen Zustandsübergänge abgeleitet haben. Da die Komponente *PR_Receiver* zeitunabhängig spezifiziert ist, definieren wir die Funktion `waiting` auf ungezeiteten Strömen. Die Zeitinformation in den Ein- und Ausgabeströmen wird mittels Anwendung der Funktion `firmtck` entfernt.

```
receiver in out ≡
(∃ waiting:: (bit → EDACK fstream → OUT fstream * ACK fstream).
waiting‘H‘(firmtck‘in) = <firmtck‘(cfst‘out), firmtck‘(csnd‘out)> ∧
(∀ s  d::data  rb,sb::bit.
(waiting‘rb‘(esnd (|d,sb|)~> s) =
If rb ≐≐ sb
then <(dout d)⇝cfst‘(waiting‘(neg rb)‘(rt‘s)),
          (ack sb)⇝csnd‘(waiting‘(neg rb)‘(rt‘s))>
else <cfst‘(waiting‘rb‘(rt‘s)), (ack sb)⇝csnd‘(waiting‘rb‘(rt‘s))>
fi) ∧
```

```
(waiting`rb`(err1~> s) =
<cfst`(waiting`rb`(rt`s)), (ack (neg rb))~>csnd`(waiting`rb`(rt`s))>)))
```

Definition von Misch- und Verteilkomponenten:

Beispiele für Basiskomponenten, deren Spezifikation zeitabhängig erfolgt, sind in der SDL-Semantikdefinition die Misch- und Verteilkomponenten.

In FOCUS wird im allgemeinen auf der Grundlage von Nachrichtenmengen ohne explizite Typisierung spezifiziert. Isabelle fordert hingegen strenge Typisierung. Diese Diskrepanz wird besonders bei der Definition von Misch- und Verteilkomponenten in HOLCF deutlich.

Für Typen, die mittels der Konstruktoren `datatype` oder `domain` vereinbart werden, gilt, daß die Konstruktor- und Selektorfunktionen eindeutig sein müssen; d. h. daß ein Bezeichner nur in einer `datatype`- bzw. `domain`-Deklaration als Funktionsname gewählt werden darf. Damit ist die Mengendefinition von M der folgenden Art in Isabelle nicht möglich:

```
datatype M1 = a1 | a2 | a3
datatype M2 = b1 | b2
datatype M  = a1 | a2 | b1
```

Außerdem existiert kein Typkonstruktor für die Typvereinigung, so daß auch die Vereinigung zweier Typen nicht direkt möglich ist. Gerade um die Ein- und Ausgabenachrichtenmengen für Misch- und Verteilkomponenten anzugeben, wäre dies aber sehr vorteilhaft.

Folgende Typvereinbarung setzt die Vorstellung von Mengenvereinigung $M_1 \cup M_2$ um:

```
datatype S = K1 M1 | K2 M2
```

Ein Element `s` des Typs `S` liegt somit in der Menge `M1` (`s = K1 e` mit `e::M1`) oder in der Menge `M2` (`s = K2 e` mit `e::M2`). Diese Typdefinition entspricht der Summe von zwei Typen. Eine Vereinigung von lediglich zwei Mengen läßt sich auch über den HOL-Typkonstruktor für Summen modellieren; bei mehr als zwei Mengen ergeben sich jedoch geschachtelte Summentypen, so daß die eben vorgestellte Modellierung mittels `datatype` zweckmäßiger ist.

Als Beispiel spezifizieren wir eine Mischkomponente `FM`, die einen Strom mit Elementen der Menge `IN` und einen Strom mit Elementen der Menge `ACK` zu einem Strom mischt. Wie eben beschrieben, definieren wir einen Typ `INACK` als

```
datatype INACK = K1 IN | K2 ACK
```

Die Funktionalität des charakterisierenden Prädikats `is_fm` lautet somit:

```
is_fm :: ((IN fitstream * ACK fitstream) ~> INACK fitstream) => bool
```

Zusätzlich ist ähnlich zu Summentypen die Einführung von Injektionsfunktionen `INACK_to_IN` bzw. `INACK_to_ACK` notwendig, die Elemente des Typs `INACK` wieder auf die zugehörigen Elemente der Teiltypen `IN` bzw. `ACK` abbilden. Mit Hilfe dieser Funktionen lassen sich

Ströme mit Elementen des Typs `INACK` in Ströme mit Elementen des Typs `IN` bzw. `ACK` überführen, so daß ein Vergleich der Ein- und Ausgabeströme von `FM` möglich ist.

```
INACK_to_IN_stream :: INACK fitstream → IN fitstream
INACK_to_ACK_stream :: INACK fitstream → ACK fitstream

INACK_to_IN_stream ≡ imap‘(se. case se of Undef ⇒ Def []
                    | Def l ⇒ Def (map INACK_to_IN l))
```

Somit wird das Prädikat `is_fm` wie folgt definiert:

```
is_fm FM ≡ (!i1 i2 out. FM ‘‘<i1,i2> = out →
(INACK_to_IN_stream ‘(ifsfilter (Collect INACK_K1)‘ out) = i1 ∧
INACK_to_ACK_stream ‘(ifsfilter (Collect INACK_K2)‘ out) = i2))
```

Für das Herausfiltern der Elemente aus dem Gesamtstrom werden die Prädikate

```
INACK_K1 s ≡ (EX d.(s = K1 din d))
INACK_K2 s ≡ (EX b.(s = K2 ack b))
```

als Parameter der Funktion `Collect` verwendet. Damit werden alle Nachrichten `din (d)` bzw. `ack (b)` mit beliebigen Parameterwerten `d` bzw. `b` aus dem Strom gefiltert.

Es zeigt sich, daß die Umsetzung von Misch- und Verteilkomponenten in HOLCF zu technischen Details führt, die bei FOCUS-Spezifikationen auf Papier außer acht gelassen werden. Bei der Spezifikation in HOLCF können in Zusammenhang mit Misch- und Verteilkomponenten Nachrichtennamen nicht durchgängig verwendet werden, was auf die strenge Typisierung von Isabelle zurückzuführen ist.
Erheblich aufwendiger werden die eben vorgestellten Definitionen, wenn mehr als zwei Ströme gemischt bzw. ein Strom in mehr als zwei Ströme aufgeteilt werden soll oder die Komplexität der zugrundeliegenden Nachrichtenmengen zunimmt.

6.3.5 Diskussion der Umsetzung

Für die Definition der SDL-Semantik haben wir die logische Kernsprache ANDL ([SS95]) so erweitert, daß die Spezifikation von zeitabhängigen Verhaltensanteilen möglich ist. Dazu haben wir für ANDL ein gezeitetes Strommodell als semantische Basis eingeführt und die ANDL-Syntax erweitert (siehe Kapitel 3.2). In den vorangegangenen Abschnitten wurden die semantischen Konzepte für die um Zeitaspekte erweiterte Sprache ANDL in HOLCF definiert, so daß nun ein einheitliches gezeitetes Strommodell für FOCUS-Spezifikationen mit und ohne Zeitanteile in HOLCF vorliegt. Damit ist eine Anbindung des von uns erweiterten ANDL an den Theorembeweiser Isabelle gegeben.

Am Beispiel des Alternating Bit Protokolls haben wir demonstriert, wie auf Basis dieser Formalisierung die formale SDL-Semantik nach HOLCF übertragen wird. Damit ist die Voraussetzung für eine zukünftige rechnergestützte Verifikation von SDL-Spezifikationen mit dem interaktiven Theorembeweiser Isabelle geschaffen. Für den praktikablen Einsatz

von Isabelle ist es jedoch notwendig, daß ein umfangreiches Sortiment an Beweisprinzipien, Taktiken, Theoremen und Hilfssätzen für die Beweisführung vorliegt. Mit Hilfe eines solchen Sortiments ist eine Infrastruktur zu definieren, so daß von der unteren Ebene des semantischen Modells abstrahiert werden kann und die technischen Details der Beweisführung verborgen werden. Nur so ist es möglich, die Interaktionen während des Beweisvorgangs zu reduzieren und sich auf die Umsetzung der Beweisideen zu konzentrieren. Die Entwicklung einer solchen Infrastruktur ist Teil der laufenden Arbeiten des Projekts SFB 342/A6 (siehe [BBSS97]) und nicht Gegenstand dieser Arbeit.

6.4 Diskussion

Die in dieser Arbeit entwickelte SDL-Semantik unterstützt zwei alternative Vorgehensweisen, um zu einer korrekten SDL-Spezifikation zu gelangen: Eigenschaftsverifikation, für die wir in den vorangegangenen Abschnitten eine Methode entwickelt haben, sowie formale Systementwicklung, auf die wir im folgenden kurz eingehen. Daran anschließend stellen wir der von uns entwickelten Verifikationsmethode für SDL zwei Verfahren gegenüber, die in der Praxis für die Überprüfung von SDL-Spezifikationen eingesetzt werden.

Formale Systementwicklung von SDL-Spezifikationen in FOCUS

In dieser Arbeit haben wir eine Verifikationsmethode entwickelt, mit der Eigenschaften einer SDL-Spezifikation verifiziert werden können. Daneben ist mit der formalen Semantik von SDL auch die Voraussetzung gegeben, um eine korrekte SDL-Spezifikation als Resultat einer formalen Systementwicklung in FOCUS zu erhalten. Dabei wird in FOCUS eine abstrakte Anforderungsspezifikation erstellt, die schrittweise verfeinert wird, bis die Übertragung in eine SDL-Spezifikation erfolgt. SDL übernimmt somit die Rolle einer Zielsprache für Systementwicklungen in FOCUS. Auf Basis der von uns definierten formalen SDL-Semantik können für jeden Entwicklungsschritt geeignete Transformationsregeln mit Verifikationsbedingungen definiert werden. Werden sämtliche Beweisverpflichtungen, die sich aus den Übergängen zwischen den Spezifikationen ergeben, verifiziert, so resultiert aus der abstrakten FOCUS-Anforderungsspezifikation eine SDL-Spezifikation, die die Eigenschaften der Anforderungsspezifikation erfüllt. Entwicklung und Verifikation gehen bei diesem Vorgehen Hand in Hand, Fehler werden bereits frühzeitig während der Entwicklung der Spezifikation entdeckt. In [Stø95] wird dieser Ansatz für einen eingeschränkten Sprachumfang von Basic SDL vorgestellt; basierend auf unserer Semantik kann der Ansatz nun für einen mächtigeren Sprachumfang von SDL erweitert werden.

Es gibt in der Literatur eine Vielzahl von Beispielen für formale Systementwicklungen in unterschiedlichen formalen Ansätzen. So beschreibt [FKO97] die transformationelle Entwicklung korrekter Telekommunikationssysteme, die in SDL-Spezifikationen resultiert, [FvGS98] stellt die Entwicklung einer Spezifikation des Alternating Bit Protokolls in der Theorie von Owicki und Gries ([OG76]) für parallele Programme vor.

Formale Systementwicklung wird in der Industrie für nicht machbar gehalten, da der Aufwand für die Verifikation der Beweisverpflichtungen aus den Verfeinerungsschritten als zu hoch und kostspielig gilt. Ein anderer Grund für die ablehnende Haltung mag darin liegen, daß SDL für viele industrielle Systementwickler bereits das höchste Maß an Abstraktion darstellt. Es besteht keine Bereitschaft, sich in die abstrakten Formalismen einzuarbeiten, die in den frühen Phasen der Systementwicklung erforderlich sind. Wir haben uns aus diesen Gründen für die Entwicklung der Methode für die Eigenschaftsverifikation von SDL-Spezifikationen entschieden, da uns diese Vorgehensweise praktikabler erscheint als eine formale Systementwicklung.

Simulation von SDL-Spezifikationen

Kommerzielle SDL-Case-Werkzeuge bieten neben graphischen Editoren und automatischer Codeerzeugung Simulatoren an, mit denen sich laut Aussage der Hersteller SDL-Spezifikationen „validieren und verifizieren“ lassen. Für die Simulation wird aus der SDL-Spezifikation ausführbarer C-Code erzeugt und um Funktionen einer Simulationsbibliothek ergänzt. Dieser Code beschreibt ein SDL-Modell der Spezifikation, welches der Simulation zugrunde liegt. Dem Modell wird ein Zustandsgraph (Erreichbarkeitsgraph) zugeordnet: jeder Knoten stellt einen erreichbaren Zustand, jede Kante eine ausführbare Transition dar. Bei der Ausführung wird die Nebenläufigkeit zwischen den Prozessen durch Interleaving modelliert. Ein Systemzustand setzt sich aus den Zuständen aller SDL-Prozesse zusammen, wobei ein Zustand eines SDL-Prozesses aus der aktuellen Belegung seiner Variablen, seiner Timerwerte und dem Inhalt seines Eingabepuffers besteht.

Bei der Simulation des Systems wird von einem vereinfachten Zeitmodell ausgegangen, das im wesentlichen der Zeitinterpretation I entspricht, die wir in Abschnitt 2.3.3 vorgestellt haben. Somit läßt sich aus den Ergebnissen der Simulation keine Aussage über das Systemverhalten unter der Einbeziehung von Zeitaspekten treffen.

Die Simulation ist sehr speicherplatzintensiv. Deshalb kann nur bei Systemen mit einem begrenzten Zustandsraum eine vollständige Überprüfung des gesamten Suchraums durchgeführt werden. Bei größeren Systemen sind vom Anwender die Suchtiefe des Zustandsgraphen und die Anzahl der zu überprüfenden Systemzustände geeignet einzuschränken; eine vollständige Überprüfung des Zustandsraums ist nicht durchführbar. Das bedeutet, daß nur Teile des möglichen Systemverhaltens überprüfbar sind, es lassen sich keine Aussagen bezüglich der Korrektheit des Systems treffen. Bei der Simulation durch Case-Werkzeuge handelt es sich damit um eine Form des Testens, die wegen ihres hohen Speicherplatzbedarfs keine zufriedenstellenden Ergebnisse liefert. Dies bestätigt eine Analyse des Werkzeugs SDT von Telelogic, die in [Hei94] durchgeführt wurde.

Model Checking

Model Checking ist eine formale Technik, um ein System auf die garantierte Erfüllung von Systemeigenschaften zu überprüfen. Diese Technik erlaubt die vollständige Überprüfung von begrenzten, zustandsendlichen Systemen und wird bereits industriell eingesetzt.

Eine Anwendung von Model Checking für SDL-Spezifikationen im Bereich der Protokollentwicklung für mobile Telephone beschreibt [RB98]. Für SDL-Systeme wird ein Verfahren für die automatische Korrektheitsüberprüfung mit dem Werkzeug SVE (Systems Verification Environment, [Büt97]) der Siemens AG vorgestellt. SVE überprüft die Korrektheit von zustandsendlichen, diskreten Systemen durch symbolisches Model Checking. Die Ausgabe von SVE ist entweder der Korrektheitsnachweis für die Eigenschaften oder eine Widerlegungssequenz, die anzeigt, welcher Ablauf des Systems die Eigenschaften verletzt. Die Anbindung von SDL an SVE ist auf den speziellen Entwicklungsprozeß bei der Siemens AG zugeschnitten und basiert auf einer darauf ausgerichteten Interpretation der SDL-Semantik. Dadurch ergeben sich eine Reihe von Einschränkungen hinsichtlich des semantischen Modells und des Sprachumfangs von SDL:

Semantisches Modell: Das zugrundeliegende semantische Modell von SDL entspricht nicht den Vorgaben der Z.100 und ist stark eingeschränkt. Die Abläufe der einzelnen SDL-Prozesse werden nicht nebenläufig ausgeführt, zu jedem Zeitpunkt ist nur ein Prozeß aktiv, alle anderen Prozesse sind inaktiv. Ein aktiver Prozeß führt einen Zustandsübergang durch und wird daraufhin wieder inaktiv. Durch ein einfaches Schedulingverfahren, das die Prozesse mit Prioritäten versieht, wird die Abarbeitung aller Prozesse organisiert. Es wird ein vereinfachtes Zeitmodell gewählt, mit der Annahme, daß alle Aktionen eines SDL-Prozesses innerhalb eines Zustandsübergangs gleichzeitig stattfinden. Die Länge der Eingabepuffer der SDL-Prozesse ist beschränkt; darüber hinaus werden statt eines Eingabepuffers je Prozeß mehrere priorisierte Puffer verwendet.

Abstraktion vom Datenanteil: SVE ist für die Überprüfung zustandsendlicher Systeme entwickelt, d.h. vor allem für diskrete Schaltungen, Steuerungen oder den Kontrollfluß in Kommunikationssystemen. Dies sind Systeme, bei denen der Datenanteil eine eher untergeordnete Rolle in Hinblick auf das Systemverhalten spielt. Bei diesen Systemen ist es möglich, den Datenanteil eines Systems bei der Überprüfung mit SVE nicht auszuwerten und somit den Zustandsraum gering zu halten.
In SDL ist der Datenanteil wesentlich für das Verhalten des Systems. Der Kontrollfluß der SDL-Prozesse ist eng mit dem Datenanteil gekoppelt (z.B. Entscheidungen über den Werten von lokalen Prozeßvariablen). Bisher unterstützt die SDL-Erweiterung von SVE jedoch nur sehr einfache Datentypen, nämlich Wahrheitswerte sowie Aufzählungstypen mit maximal zehn verschiedenen Werten. Damit ist der Softwareentwickler bei der Spezifikation des Datenanteils stark eingeschränkt. Eine genaue Betrachtung von SDL-Spezifikationen, die zu den Standardbeispielen für SDL zählen, hat ergeben, daß gerade im Datenanteil von SDL-Spezifikationen viele Spezifikationsfehler enthalten sind. Beispiele für Fehler finden sich in den SDL-Spezifikationen des Abracadabra-Protokolls ([Hin96a, Kle95]), des Sliding Window Protokolls ([FHH97]) und eines unzuverlässigen Mediums ([FHH97]).

Model Checking eignet sich, um in den frühen Entwicklungsphasen den Kontrollfluß von SDL-Spezifikationen zu überprüfen, und stellt damit, wie in [Büt97] motiviert, eine sinnvolle Ergänzung der konventionellen Qualitätsverfahren wie Simulation oder Testen dar. Model Checking eignet sich jedoch nicht, um die Korrektheit einer SDL-Spezifikation zu zeigen.

Abschließende Bemerkung

Wie unsere Analysen gezeigt haben, sind Model Checking und Simulation nicht geeignet, um die Korrektheit einer SDL-Spezifikation vollständig zu überprüfen. Die von uns entwickelte Verifikationsmethode hingegen ermöglicht den Korrektheitsnachweis von SDL-Spezifikationen. Dabei werden sämtliche Bestandteile einer Spezifikation miteinbezogen, nämlich strukturelle Beschreibung, Datenanteil und die Verhaltensbeschreibung durch die SDL-Prozesse. Die Anwendung unserer Methode ist zwar mit Aufwand verbunden, jedoch läßt sich dieser durch rechnerbasierte Werkzeugunterstützung verringern. Dazu zählen insbesondere die automatische Generierung der SDL-Semantik sowie eine Unterstützung durch den Theorembeweiser Isabelle bei der Beweisführung. Zudem rechtfertigt die garantierte Fehlerfreiheit einer verifizierten SDL-Spezifikation (hinsichtlich der gegebenen Anforderungsspezifikation) den Aufwand, der mit unserer Methode verbunden ist. Gerade in sicherheitskritischen Anwendungsbereichen ist die garantierte Fehlerfreiheit von Softwaresystemen unerläßlich – die Kosten, die in diesen Bereichen durch Softwarefehler verursacht werden, rechtfertigen den Kostenaufwand für eine formale Verifikation ([Rei97]).

Kapitel 7

Zusammenfassung und Ausblick

Mit den Ergebnissen der vorliegenden Arbeit ist für SDL der Wandel von einer informellen zu einer formalen Spezifikationssprache vollzogen. Unsere Arbeit demonstriert, daß pragmatische, in der Industrie eingesetzte Beschreibungstechniken mit den formalen Ansätzen aus dem akademischen Bereich kombinierbar sind und sich die Stärken beider Ausprägungen verbinden lassen. Dieses Kapitel faßt die Ergebnisse der Arbeit zusammen und zeigt eine Auswahl möglicher zukünftiger Arbeiten.

7.1 Zusammenfassung der Ergebnisse

Die Zielsetzung dieser Arbeit war die Erstellung einer formalen Semantikdefinition für die Spezifikations- und Beschreibungssprache SDL und darauf aufbauend die Entwicklung einer Verifikationsmethode für SDL-Spezifikationen. Nachfolgend geben wir einen Überblick über die wesentlichen Ergebnisse dieser Arbeit.

Defizite von SDL Unsere kritische Analyse der Konzepte und Sprachkonstrukte von SDL hat eine Vielzahl von Defiziten und Widersprüchlichkeiten aufgedeckt. Dazu zählen das mangelhafte Zeitkonzept, eine ungenügende Behandlung von partiellen Datentypen sowie das Fehlen von Fairneßbedingungen. Als Folge davon ist einer SDL-Spezifikation keine klare Bedeutung zugeordnet; das Verhalten eines mit SDL modellierten Systems hängt von der individuellen Interpretation des Softwareingenieurs ab. Diese Tatsache unterstreicht, daß eine formale Semantikdefinition für SDL, basierend auf mathematisch-logischen Konzepten, dringend erforderlich ist. Die von der ITU-T als Anhang der Empfehlung Z.100 veröffentlichte SDL-Semantik stellt keine Semantik im formalen, mathematischen Sinn dar. Unsere Analyse von SDL erbringt eindeutig den Nachweis, daß SDL keine formale Sprache ist, auch wenn dies von zahlreichen Anwendern, Anbietern von SDL-Case-Tools sowie Mitgliedern der Standardisierungsgruppe behauptet wird. Hier wird deutlich, daß die Eigenschaft „formal“ in der Industrie mit „eindeutige Syntax“ gleichgesetzt wird.

Festlegung des Zeitkonzepts Ein besonders gravierendes Defizit von SDL stellt das ungenügende und in sich widersprüchliche Zeitkonzept dar. Ein wesentlicher Beitrag der

vorliegenden Arbeit liegt in der Festlegung eines wohldefinierten Zeitbegriffs für SDL. Der grundlegende Zusammenhang zwischen Zeit und Verhalten ist für SDL-Systeme bisher in keiner Arbeit untersucht worden. Die in einigen Arbeiten geäußerte Kritik am Zeitkonzept in SDL bezieht sich auf die mangelnde Ausdrucksstärke von SDL für harte Echtzeitanforderungen.
Wir haben die verschiedenen Interpretationen der SDL-Zeit, die unter SDL-Anwendern vertreten sind, diskutiert sowie ihre jeweilige Auswirkung auf das Systemverhalten an einem Beispiel präsentiert. Abschließend haben wir ein Zeitkonzept definiert, das eine intuitive Vorstellung von Zeit in einem verteilten System auf abstrakter Ebene widerspiegelt: über die Ausführungsgeschwindigkeit eines SDL-Prozesses lassen sich keine Aussagen treffen. Somit verhält sich ein SDL-Prozeß nichtdeterministisch hinsichtlich des Zeitbedarfs. Das Zeitkonzept nimmt insbesondere keinen Bezug auf die tatsächlichen Ausführungszeiten, die sich bei der späteren Umsetzung der Spezifikation in ein lauffähiges Programm ergeben. Damit werden quantitative Angaben für den Ablauf von gesetzten Timern überflüssig, so daß wir das Timerkonzept dahingehend verändert haben. Dieses Zeitkonzept liegt unserer formalen Semantikdefinition zugrunde.

Formale Semantikdefinition Die von uns entwickelte SDL-Semantik ist intuitiv und nachvollziehbar und gibt SDL-Spezifikationen eine klare und mathematisch präzise Bedeutung. Die Semantik wird definiert, indem SDL-Spezifikationen in FOCUS-Spezifikationen übertragen werden. Da FOCUS-Spezifikationen eine formale Semantik auf Basis von stromverarbeitenden Funktionen besitzen, ist durch unser Vorgehen eine denotationelle Semantik für SDL gewonnen. Der Vorteil unseres Vorgehens liegt darin, daß nicht direkt auf der semantischen Ebene, sondern auf der Spezifikationsebene von FOCUS aufgesetzt wird. Dabei haben wir bewußt Systemstrukturdiagramme und die logische Kernsprache ANDL eingesetzt, um die Semantik auch Anwendern zugänglich zu machen, die mit mathematischen Notationen weniger vertraut sind. Unsere Semantikdefinition beseitigt die von uns aufgedeckten Defizite von SDL. Typische Charakteristika von verteilten Systemen, wie z.B. Kapselung von Prozessen oder Modularität, die durch die Defizite von SDL verletzt werden, sind für ein mit SDL spezifiziertes System mit der von uns gegebenen formalen Semantik nun erfüllt.
Für statische und dynamische Systeme haben wir das jeweils adäquate semantische Modell von FOCUS gewählt. Für beide Arten von Systemen konnte die Semantik so definiert werden, daß eine schematische Umsetzung von SDL nach FOCUS unterstützt wird.
Besonders hervorzuheben ist, daß mit der Wahl von FOCUS und der Integration von SPECTRUM, das wie FOCUS auf bereichstheoretischen Konzepten basiert, als Grundlage der Semantikdefinition ein einheitlicher Rahmen für die unterschiedlichen Aspekte einer SDL-Spezifikation gegeben ist. Wir haben durch unsere Semantikdefinition den strukturellen Aufbau, den Signalfluß, das Ein-/Ausgabeverhalten der einzelnen Prozesse, die dynamische Prozeßgenerierung sowie den Datenanteil in den Prozessen und Signalen in einem einheitlichen semantischen Modell integriert. Dieser Aspekt unserer Semantik wurde in [SK98], das mehrere SDL-Semantiken untersucht, hervorgehoben.

Verifikationsmethode Mit dieser Arbeit liegt eine durchgängige Methode zur Verifikation von SDL-Spezifikationen vor, die in mehreren Phasen von der Erstellung der SDL-Spezifikation über die Angabe der dazugehörigen SDL-Semantik bis zur Beweisführung

reicht. Damit ist nun erstmals ein durchgängig verifizierbarer Systementwicklungsprozeß mit SDL möglich. Die systematische, formale Verifikation von SDL-Spezifikation ist bisher in keiner Arbeit behandelt worden. Mit unserer Verifikationsmethode sind Aussagen über alle Systemaspekte einer SDL-Spezifikation möglich, nicht nur über Teilaspekte, wie dies bei der Verwendung von Model Checking oder Simulationen der Fall ist. Da unsere SDL-Semantik funktional ausgerichtet ist, lassen sich die Beweistechniken aus FOCUS ohne Einschränkung bei der Beweisführung einsetzen. Allerdings wirken sich Eigenheiten von SDL, wie beispielsweise das Konzept des Eingabepuffers in Verbindung mit Savesignalen, ungünstig auf die Beweisführung aus.

Praktikabilität Während der Arbeit haben wir die praktische Anwendbarkeit unseres Vorgehens berücksichtigt, da diese für die Akzeptanz und den Einsatz unerläßlich ist.

- Anhand zweier Fallstudien, dem Alternating Bit Protokoll und dem Daemon Game, haben wir die praktische Anwendung der von uns entwickelten Semantik sowie der Verifikationsmethode gezeigt.
- Die zu einer SDL-Spezifikation gehörende Semantik kann automatisch generiert werden - dies haben wir mittels eines prototypischen Umsetzers demonstriert. Darauf aufbauend kann eine Umsetzung der SDL-Semantik nach HOLCF erfolgen. Für die Formalisierung von SDL in HOLCF liegt mit dieser Arbeit ein umfassendes Konzept vor, so daß die Beweisführung für SDL-Spezifikationen zukünftig mit Unterstützung des Theorembeweisers Isabelle durchgeführt werden kann.

Ergebnisse für Focus Neben der zentralen Zielsetzung, die in der formalen Fundierung und Verifikationsmethode für SDL liegt, stellt diese Arbeit auch eine umfangreiche Anwendung von FOCUS hinsichtlich der Aspekte Spezifikation und Verifikation dar und hat einige Erweiterungen von FOCUS erbracht.

- Für die semantische Fundierung der SDL-Prozeßerzeugung war zunächst die Erweiterung des semantischen Modells von FOCUS erforderlich. Dazu haben wir in [HS96, HS97] basierend auf [GS97] eine methodische Vorgehensweise für die Spezifikation dynamischer Systeme in FOCUS entwickelt. Unsere SDL-Semantik für dynamische SDL-Systeme stellt neben [Spi98] eine der ersten Anwendungen dieser Erweiterung dar.
- ANDL, die in [SS95] definierte logische Kernsprache von FOCUS, wurde um die Spezifikation zeitabhängiger Basiskomponenten erweitert. Dazu wurde anstelle des bisher ungezeiteten ein gezeitetes semantisches Strommodell eingeführt. Für die Formalisierung dieser ANDL-Erweiterung in der Logik HOLCF des Theorembeweisers Isabelle wurde in HOLCF ein gezeitetes Strommodell mit pulsgetriebenen Funktionen definiert. Damit ist für die Verifikation von FOCUS-Spezifikationen mit gezeiteten und ungezeiteten Anteilen eine Anbindung an den Theorembeweiser Isabelle gegeben.
- Die von uns entwickelte Vorgehensweise für die Verifikation von SDL-Spezifikationen läßt sich allgemein für ANDL-Spezifikationen verwenden, so daß hiermit eine methodische Vorgehensweise für Verifikationsaufgaben im funktionalen Stil in FOCUS vorliegt. Dabei sind insbesondere das Ableiten von Gleichungssystemen aus ANDL-

Spezifikationen, die Ausnutzung von Fixpunkteigenschaften pulsgetriebener Funktionen sowie die Behandlung zeitabhängiger und zeitunabhängiger Spezifikationsanteile enthalten.

7.2 Ausblick

Zum Abschluß stellen wir einige zukünftige Arbeiten vor, die sich als Weiterführung der in dieser Arbeit erzielten Ergebnisse anbieten.

Anwendung der Methode Eine sinnvolle Fortführung unserer Arbeit besteht darin, die von uns entwickelte Verifikationsmethode an weiteren SDL-Spezifikationen zu erproben und Erfahrungen hinsichtlich der Beweisführung zu sammeln. Wünschenswert ist eine Verfeinerung der Methode für spezielle Anwendungsgebiete, wie etwa Kommunikationsprotokolle. Hierfür ist zu untersuchen, ob sich aufgrund der speziellen Architektur von Protokollen, die in der Regel aus Sender, Empfänger und Übertragungsmedien aufgebaut sind, ähnliche Beweisideen und Beweistechniken finden lassen, die als allgemeine Richtlinie formuliert werden können. Damit würde sich der Aufwand für die Verifikation erheblich verringern.
Sobald mit den Arbeiten des SFB 342/A6 eine Infrastruktur für FOCUS in Isabelle geschaffen ist, kann die Verifikation von SDL-Spezifikationen mit Isabelle interaktiv durchgeführt werden. Hier sind geeignete Theoreme und Lemmata für SDL-Spezifikationen zu formulieren und zu beweisen. Mit diesen können dann die automatischen Beweiswerkzeuge von Isabelle erweitert werden, so daß ein hoher Automatisierungsgrad der Beweisführung erreicht wird.

Integration von Message Sequence Charts Ein wesentlicher Punkt unserer Verifikationsmethode ist die Formulierung der Beweisverpflichtung für die SDL-Spezifikation. Dabei ist ein fundiertes Verständnis der Anforderungsdefinition und der SDL-Spezifikation erforderlich, damit die Beweisverpflichtung tatsächlich diejenigen Eigenschaften der SDL-Spezifikation beschreibt, die die Korrektheit der Spezifikation sicherstellen. Aus diesem Grund ist es sinnvoll, daß dieser Schritt vom Entwickler der SDL-Spezifikation selbst ausgeführt wird. Damit dieser nicht direkt auf die prädikatenlogische Ebene zugreifen muß, ist dafür zu sorgen, daß für die Formulierung der Beweisverpflichtung intuitive Beschreibungstechniken eingesetzt werden können. Hier bietet sich die Verwendung von Message Sequence Charts (MSCs, [IT96]) an, die SDL-Anwendern eine vertraute Beschreibungstechnik sind. Sie werden in Verbindung mit SDL für die Spezifikation von Nachrichtenaustausch und für Beispielabläufe von Systemen verwendet. Damit MSCs für die Formulierung von Beweisverpflichtungen verwendet werden können, sind zwei Voraussetzungen zu schaffen. Zum einen ist eine formale Semantik für MSC zu definieren, damit die Umsetzung von MSC in prädikatenlogische Ausdrücke erfolgen kann. Zum anderen ist zu untersuchen, für welche Klasse von Eigenschaften einer SDL-Spezifikation sich MSCs einsetzen lassen und welcher Sprachumfang von MSC dabei benötigt wird. Sind diese Voraussetzungen gegeben, so kann ein Anwender durch die Spezifikation von MSCs die Korrektheitsbedingung für eine SDL-Spezifikation graphisch angeben.

Implementierung der Werkzeugunterstützung Eine naheliegende Fortsetzung dieser Arbeit liegt in der vollständigen Implementierung der durchgängigen Werkzeugunterstützung, die wir in der Arbeit skizziert und teilweise prototypisch implementiert haben. Zusammen mit der oben angesprochenen Automatisierung der Beweise für SDL-Spezifikationen und der erläuterten Integration von Message Sequence Charts läßt sich die Vorstellung einer idealen Werkzeugunterstützung für Softwareentwickler entwerfen: Der Entwickler erstellt die SDL-Spezifikation und gibt die zu überprüfende Eigenschaft des SDL-Systems an. Dabei kann er aus einer Liste vorgegebener, textuell beschriebener Eigenschaften wählen oder Message Sequence Charts angeben. Beide Darstellungen werden im Werkzeug in prädikatenlogische Formeln umgesetzt. Ohne weitere Interaktion seitens des Anwenders wird der Korrektheitsbeweis durchgeführt. Wird die Beweisführung erfolgreich beendet, so erhält der Entwickler damit die Bestätigung für die Korrektheit seiner SDL-Spezifikation hinsichtlich der vorgegebenen Eigenschaft. Falls die Beweisführung scheitert, erfüllt die SDL-Spezifikation nicht die vorgegebene Eigenschaft. Der Entwickler erhält die entsprechende Beweisstelle, an der die Beweisführung gescheitert ist, und Hinweise, auf welchen Teil der SDL-Spezifikation an dieser Beweisstelle Bezug genommen wird.

Erweiterung der SDL-Semantik In dieser Arbeit erfolgte die formale Fundierung von SDL für die meist verwendeten Sprachkonstrukte, die unter dem Begriff Basic SDL zusammengefaßt sind. Weiterführende Arbeit sollten sich damit befassen, die SDL-Semantik für weitere SDL-Sprachkonstrukte, vor allem die objektorientierte Ausprägung von SDL, zu erweitern. Dem sollte allerdings eine eingehende Analyse der Sprachkonstrukte vorausgehen, die diese auf Konsistenz und ihre Verträglichkeit mit den Prinzipien verteilter Systeme überprüft. Interessant ist in diesem Zusammenhang die Frage, inwieweit die objektorientierte Ausprägung von SDL objektorientierten Konzepten gerecht wird.

SDL als Zielsprache von Focus Nachdem mit dieser Arbeit eine formale Semantik für SDL, formuliert in FOCUS, vorliegt, können sich weitere Arbeiten mit der Rolle von SDL als Implementierungssprache von FOCUS befassen. So läßt sich beispielsweise SDL als Zielsprache einer formalen Systementwicklung in FOCUS einsetzen, die von einer abstrakten Spezifikation zu einer SDL-Spezifikation führt. Dies stellt eine Erweiterung von [HS94] dar, in der dies für einen sehr eingeschränkten Sprachumfang von SDL durchgeführt wird. Ferner ist zu überlegen, ob sich SDL als pragmatische Spezifikationstechnik in FOCUS integrieren läßt. Gerade die SDL-Prozesse könnten ähnlich wie die Erweiterten Ereignisdiagramme ([HSS96]), die eine Einbettung von Message Sequence Charts darstellen, als pragmatische Automatendarstellung für Ein-/Ausgabeverhalten dienen.

Rolle von SDL als Softwareentwicklungsmethode Ein Aspekt, der in dieser Arbeit bewußt nicht behandelt wurde, ist die Rolle von SDL als Softwareentwicklungsmethode. Es ist jedoch durchaus von Interesse zu untersuchen, inwieweit die in der SDL-Literatur vorgeschlagenen Methoden einen durchgängigen Systementwicklungsprozeß unterstützen, insbesondere ob die in den Werkzeugen vorgeschlagene Anforderungsspezifikation mit der objektorientierten Methode OMT ([Rum91]) und Message Sequence Charts systematisch in eine SDL-Spezifikation überführbar ist.

Literaturverzeichnis

[BB91] Falko Bause und Peter Buchholz. *Protocol analysis using a timed version of SDL.* in J. Quemada, J. Mãnas und E. Vazquez (Hrsg.), *Formal Description Techniques.* North Holland 1991.

[BBSS97] Manfred Broy, Max Breitling, Bernhard Schätz und Katharina Spies. *SFB342, Teilprojekt A6 „Methodik des Entwurfs verteilter Systeme", Fortsetzungsantrag für die Jahre 1998-2000*, 1997.

[BDD+93] Manfred Broy, Frank Dederichs, Claus Dendorfer, Max Fuchs, Thomas F. Gritzner und Rainer Weber. *The Design of Distributed Systems — An Introduction to* FOCUS. SFB-Bericht 342/2/92 A, Technische Universität München, 1993.

[BFG+93a] Manfred Broy, Christian Facchi, Radu Grosu, Rudi Hettler, Heinrich Hußmann, Dieter Nazareth, Franz Regensburger, Oscar Slotosch und Ketil Stølen. *The Requirement and Design Specification Language SPECTRUM- Part I.* Technischer Bericht TUM-I9311, Technische Universität München, 1993.

[BFG+93b] Manfred Broy, Christian Facchi, Radu Grosu, Rudi Hettler, Heinrich Hußmann, Dieter Nazareth, Franz Regensburger, Oscar Slotosch und Ketil Stølen. *The Requirement and Design Specification Language SPECTRUM - Part II.* Technischer Bericht TUM-I9312, Technische Universität München, 1993.

[BGH+97] Ruth Breu, Radu Grosu, Franz Huber, Bernhard Rumpe und Wolfgang Schwerin. *Towards a Precise Semantics for Object-Oriented Modeling Techniques.* in Haim Kilov und Bernhard Rumpe (Hrsg.), *Proceedings of the ECOOP'97 Workshop on Precise Semantics for Object-Oriented Modeling Techniques.* Technische Universität München 1997.

[BH93] Rolv Bræk und Øystein Haugen. *Engineering Real Time Systems.* Prentice Hall, 1993.

[BHH+97] Ruth Breu, Ursula Hinkel, Christoph Hofmann, Cornel Klein, Barbara Paech, Bernhard Rumpe und Veronika Thurner. *Towards a Formalization of the Unified Modeling Language.* in *Proceedings of ECOOP'97, LNCS 1241.* Springer Verlag 1997.

[BJ78] Dines Bjørner und Cliff B. Jones. *The Vienna Development Method: The Meta-Language.* LNCS 61. Springer Verlag, 1978.

[BJ82] Dines Bjørner und Cliff B. Jones. *Formal Specification and Software Development.* Prentice Hall, 1982.

[BM95] J.A. Bergstra und C.A. Middelburg. *Process algebra semantics of phiSDL.* in *ACP '95, Report 95-14.* Eindhoven University of Technology, Department of Mathematics and Computing Science 1995, S. 309–346. Eingeladener Vortrag.

[BMŞ97a] J.A. Bergstra, C.A. Middelburg und R. Şoricuţ. *Discrete Time Network Algebra for a Semantic Foundation of SDL.* UNU/IIST Report 98, The United Nations University, International Institute for Software Technology, 1997.

[BMŞ97b] J.A. Bergstra, C.A. Middelburg und Gh. Ştefănescu. *Network Algebra for Asynchronous Dataflow.* International Journal of Computer Mathematics 65 (1997), S. 57 – 88.

[BMU97] J.A. Bergstra, C.A. Middelburg und Y.S. Usenko. *Discrete-time Process Algebra and the Semantics of SDL.* UNU/IIST Report 99, The United Nations University, International Institute for Software Technology, 1997.

[Bro91] Manfred Broy. *Towards a Formal Foundation of the Specification and Description Language SDL.* Formal Aspects of Computing 3 (1991), S. 21–57.

[Bro92] Manfred Broy. *Graphical and functional specification and verification of the behaviour of process and block diagrams in SDL.* Interner Bericht, Technische Universität Müchen, 1992.

[BS93] Jonathan Bowen und Victoria Stavridou. *The Industrial Take-up of Formal Methods in Safety-Critical and Other Areas: A Perspective.* in J.C.P. Woodcock und P.G. Larsen (Hrsg.), *FME'93: Industrial-Strength Formal Methods, First International Symposium of Formal Methods Europe, LNCS 670.* Springer Verlag 1993.

[BS98] Manfred Broy und Ketil Stølen. FOCUS *on System Development*, 1998. In Vorbereitung.

[Büt97] Wolfram Büttner. *Formale Spezifikation, Verifikation und Synthese zustandsendlicher Systeme.* it+ti 39 (1997) 3, S. 15 – 21.

[BW88] Richard Bird und Philip Wadler. *Introduction to Functional Programming* . Prentice Hall, 1988.

[Cha93] Zhou Chaochen. *Duration calculii: An Overview.* in D. Bjørner, M. Broy und I.V.Pottosin (Hrsg.), *Proceedings of Formal Methods in Programming and Their Applications.* Springer 1993, S. 256 – 266.

[DKRS91] Roger Duke, Paul King, Gordon Rose und Graeme Smith. *The Object-Z specification language (version 1).* Technischer Bericht, SVRC, The University of Queensland, 1991.

[DS89] Peter Dybjer und Herbert P. Sander. *A Functional Programming Approach to the Specification and Verification of Concurrent Systems.* Formal Aspects of Computing 1 (1989), S. 303–319.

[EHS97] Jan Ellsberger, Dieter Hogrefe und Armadeo Sarma. *Object-oriented Language for Communicating Systems.* Prentice Hall, 1997.

[FHH97] Christian Facchi, Markus Haubner und Ursula Hinkel. *The SDL Specification of the Sliding Window Protocol Revisited.* in Ana Cavalli und Amardeo Sarma (Hrsg.), *SDL '97: Time for Testing – SDL, MSC and Trends.* North Holland 1997, S. 507 – 519.

[FKO97] Clemens Fischer, Stephan Kleuker und Ernst-Rüdiger Olderog. *Beweisbar korrekte Telekommunikationssysteme.* it+ti 39 (1997) 3, S. 22 – 28.

[FLP95] Joachim Fischer, Stefanie Lau und Andreas Prinz. *A Short Note about BSDL.* SDL Newsletter 18 (Januar 1995), S. 15–21.

[FT97] Hans Fleischhack und Josef Tapken. *An M-Net Semantics for a Real-Time Extension of phi-SDL.* in John Fitzgerald, Cliff B. Jones und Peter Lucas (Hrsg.), *FME '97: 4th International Symposium of Formal Methods Europe, LNCS 1313.* Springer Verlag 1997.

[Fuc94] Maximilian Fuchs. *Technologieabhängigkeit von Spezifikationen digitaler Hardware.* Dissertation, Technische Universität München, 1994.

[FvGS98] W.H.J. Feijen, A.J.M. van Gasteren und Birgit Schieder. *An elementary derivation of the Alternating Bit Protocol.* in *MPC'98 (Fourth International Conference on Mathematics of Program Construction)*, 1998. erscheint in LNCS.

[God91a] Jens C. Godskesen. *An Operational Semantic Model for Basic SDL.* Technischer Bericht, TFL, 1991.

[God91b] Jens C. Godskesen. *An Operational Semantic Model for Basic SDL (Extended Abstract).* in Ove Færgemand und Rick Reed (Hrsg.), *SDL '91 - Evolving Methods. Proceedings of the Fifth SDL Forum.* North-Holland 1991.

[Gra90] Jens Grabowski. *Statische und dynamische Analysen für SDL-Spezifikationen auf der Basis von Petri-Netzen und Sequence-Charts.* Diplomarbeit, Universität Hamburg, 1990.

[Gro96a] Radu Grosu. *About Recursive Mobile Networks.* Interner Bericht, Technische Universität München, 1996.

[Gro96b] Radu Grosu. *Recursive Networks and Time Abstraction.* Interner Bericht, Technische Universität München, 1996.

[GS90] Carl A. Gunter und Dana S. Scott. *Semantic Domains.* in J. van Leeuwen (Hrsg.), *Handbook of Theoretical Computer Science.* North-Holland, 1990, S. 633–674.

[GS97] Radu Grosu und Ketil Stølen. *Compositional Specification of Mobile Systems.* Technischer Bericht TUM-I9748, Technische Universität München, 1997.

[Gut75] John V. Guttag. *The Specification and Application to Programming of Abstract Data Types.* Dissertation, University of Toronto, 1975.

[Har87] David Harel. *Statecharts: a visual formalism for complex systems.* Science of Computer Programming 8 (1987), S. 231–274.

[Hei94] Stefan Heinkel. *Verifikation in SDL.* Diplomarbeit, Universität Karlsruhe, 1994.

[Hin96a] Ursula Hinkel. *Eine Analyse der SDL-Spezifikation des Abracadabra Protokolls.* Interner Bericht, Technische Universität München, 1996.

[Hin96b] Ursula Hinkel. *SDL-Projekt: Entwurf eines Prototypen für die Übertragung von SDL-Spezifikationen nach* FOCUS. Interner Bericht, Technische Universität München, 1996.

[HL92] Dieter Hogrefe und Stefan Leue. *Specifying Real-Time Requirements for Communication Protocols.* Technischer Bericht IAM92-015, University of Berne, 1992.

[Hoa85] Charles A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[HS94] Eckhardt Holz und Ketil Stølen. *An Attempt to Embed a Restricted Version of SDL as a Target Language in* FOCUS. in Dieter Hogrefe und Stefan Leue (Hrsg.), *Proc. Forte'94.* Chapman & Hall 1994, S. 324–339.

[HS96] Ursula Hinkel und Katharina Spies. *Anleitung zur Spezifikation von mobilen, dynamischen* FOCUS*-Netzen.* Technischer Bericht TUM-I 9639, Technische Universität München, 1996.

[HS97] Ursula Hinkel und Katharina Spies. *Spezifikationsmethodik für mobile, dynamische* FOCUS*-Netze.* in A. Wolisz, I. Schieferdecker und A. Rennoch (Hrsg.), *Formale Beschreibungstechniken für verteilte Systeme, GI/ITG-Fachgespräch 1997.* GMD Verlag (St.Augustin) 1997.

[HSS96] Franz Huber, Bernhard Schätz und Katharina Spies. *AutoFocus - Ein Werkzeugkonzept zur Beschreibung verteilter Systeme.* in Ulrich Herzog (Hrsg.), *Formale Beschreibungstechniken für verteilte Systeme.* Universität Erlangen-Nürnberg 1996.

[Huß97] Heinrich Hußmann. *Formal Foundations for Software Engineering. Lecture Notes in Computer Science*, Band 1322. Springer Verlag, 1997.

[ISO91] ISO/IEC. *Information Technology - Open System Interconnection - Guidelines for the Application of Estelle, LOTOS and SDL.* Technischer Bericht ISO/IEC/TR 10167, 1991.

[IT93a] ITU-T. *Annex F to Recommendation Z.100 (Formal Definition of SDL92).* ITU, 1993.

[IT93b] ITU-T. *Recommendation Z.100, Specification and Description Language (SDL).* ITU, 1993.

[IT95] ITU-T. *Recommendation Z.105. SDL92 Combined with ASN.1 (SDL/ASN.1).* ITU, 1995.

[IT96] ITU-T. *Z.120 – Message Sequence Chart (MSC).* ITU-T, Geneva 1996.

[ITU94] ITU-T Study Group 10. *BSDL The Language - Version 0.2 (preliminary).* Technischer Bericht, ITU-T, 1994.

[Kle95] Christine Klein. *Spezifikation eines Dienstes und Protokolls in* FOCUS *- Die Abracadabra-Fallstudie.* Diplomarbeit, Technische Universität München, 1995.

[Leu95] Stefan Leue. *Specifying Real-Time Requirements for SDL Specifications - A Temporal Logic-Based Approach.* in *Protocol Specification, Testing, and Verification PSTV'95.* Chapman & Hall 1995.

[Lev95] Nancy G. Leveson. *Safeware: system safety and computers.* Addison-Wesley, 1995.

[MGHS96] Simon Mørk, Jens C. Godskesen, Michael R. Hansen und Robin Sharp. *A Timed Semantics for SDL.* in R. Gotzhein und J. Bredereke (Hrsg.), *Forte/PSTV'96.* Chapman & Hall 1996.

[Mil80] Robin Milner. *A Calculus of Communicating Systems.* LNCS, Band 92. Springer Verlag, 1980.

[NMSvO98] Tobias Nipkow, Olaf Müller, Oscar Slotosch und David von Oheimb. *HOLCF.* (1998). Eingereicht für Journal of Functional Programming.

[OFMP+94] Anders Olsen, Ove Færgemand, Birger Møller-Pedersen, Rick Reed und J. R. W. Smith. *Systems Engineering Using SDL-92.* Elsevier Science, 1994.

[OG76] S. Owicki und D. Gries. *An Axiomatic Proof Technique for Parallel Programs I.* Acta Informatica 6 (1976), S. 319 – 340.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover.* LNCS, Band 828. Springer Verlag, 1994.

[PBB+82] P. Pepper, M. Broy, F.L. Bauer, H. Partsch, W. Dosch und M. Wirsing. *Abstrakte Datentypen: Die algebraische Definition von Rechenstrukturen.* Informatik Spektrum 5 (1982), S. 107 – 119.

[RB98] Franz Regensburger und Aenne Barnard. *Formal Verification of SDL Systems at the Siemens Mobile Phone Department.* in *TACAS 98 (Tools and Algorithms for the Construction and Analysis of Systems)*, 1998. erscheint in LNCS.

[Reg94] Franz Regensburger. *HOLCF: Eine konservative Erweiterung von HOL um LCF.* Dissertation, Technische Universität München, 1994.

[Reg95a] Franz Regensburger. *HOLCF: Higher Order Logic of Computable Functions.* in Thomas E. Schubert, Phillip J. Windley und James Alves-Foss (Hrsg.), *Higher Order Logic Theorem Proving and Its Application (HOL95)*, 1995, S. 293–307.

[Reg95b] Franz Regensburger. *Predefined Specifications.* Interner Bericht, Technische Universität München, 1995.

[Rei97] Wolfgang Reif. *Software-Verifikation und ihre Anwendungen.* it+ti 39 (1997) 3, S. 34 – 40.

[Rum91] James Rumbaugh. *Object-Oriented Modelling and Design.* Prentice Hall, 1991.

[Rum96] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme.* Dissertation, Technische Universität München, 1996.

[SHB96] Bernhard Schätz, Heinrich Hußmann und Manfred Broy. *Graphical Development of Consistent System Specifications* . in Marie-Claude Gaudel und James Woodcock (Hrsg.), *FME'96: Industrial Benefit and Advances in Formal Methods, LNCS 1051.* Springer Verlag 1996, S. 248–267.

[SIN97] SINTEF. *TIMe: The Integrated Method*, 1997.

[SK98] Jason Steggles und Piotr Kosiuczenko. *A Formal Model for SDL Specifications based on Timed Rewriting Logic.* in *Second Workshop on Precise Semantics for Software Modeling Techniques*, 1998. Technischer Bericht, Technische Universität München.

[Slo97] Oscar Slotosch. *Refinements in HOLCF: Implementation of Interactive Systems.* Dissertation, Technische Universität München, 1997.

[Spi92] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International Series in Computer Science, 1992.

[Spi98] Katharina Spies. *Eine Methode zur formalen Modellierung von Betriebssystemkonzepten.* Dissertation, Technische Universität München, 1998.

[SS95] Bernhard Schätz und Katharina Spies. *Formale Syntax zur logischen Kernsprache der* FOCUS*-Entwicklungsmethodik.* SFB-Bericht 342/16/95 A, Technische Universität München, Institut für Informatik, 1995.

[SSR89] R. Saracco, J. R. W. Smith und R. Reed. *Telecommunications system engineering using SDL.* North-Holland, 1989.

[Stø95] Ketil Stølen. *Development of SDL Specifications in* FOCUS. in Amadeo Sarma und Rolv Bræk (Hrsg.), *Proceedings SDL Forum 95*, 1995, S. 269–278.

[TBYS96] Jeffrey J.P. Tsai, Yaodong Bi, Stefe J.H. Yang und Ross A.W. Smith. *Distributed Real-Time Systems: Monitoring, Visualization, Debugging, and Analysis.* John Wiley & Sons, 1996.

[Tur93] Kenneth J. Turner (Hrsg.). *Using Formal Description Techniques - An Introduction to Estelle, Lotos and SDL.* John Wiley & Sons, 1993.

[Ver97] Verilog. *Reference Manual - ObjectGEODE Simulator*, 1997.

[Wen97] Markus Wenzel. *Type classes and Overloading in Higher-Order Logic.* in *Theorem Proving in Higher Order Logics – TPHOL'97. LNCS 1275.* Springer Verlag 1997.

Anhang A

Fallstudie Daemon Game

Anhand der SDL-Spezifikation des Daemon Game zeigen wir die Anwendung der in Kapitel 5 erfolgten formalen Semantikdefinition für SDL-Systeme mit dynamischer Prozeßerzeugung. Dabei stellen wir eine Vorgehensweise vor, die die Aspekte Dynamik und Mobilität innerhalb des Daemon Game sichtbar macht und das systematische Erstellen der Semantikdefinition unterstützt. Die SDL-Spezifikation des Daemon Game ist in einem Technischen Bericht der ISO/IEC ([ISO91]) und in leicht abgewandelter Form in einigen SDL-Büchern enthalten und gilt als Standardbeispiel für die dynamische Prozeßerzeugung. Im folgenden stellen wir kurz die für die Prozeßerzeugung wesentlichen Teile der SDL-Spezifikation vor und beschreiben anschließend die Umsetzung in die Semantikdefinition in FOCUS. Dabei stützen wir uns auf die Vorgehensweise für die Spezifikation dynamischer, mobiler Systeme ab, die wir in [HS96, HS97] entwickelt haben, und wenden die in Kapitel 5 definierten Schemata für die dynamische SDL-Prozeßerzeugung an.

A.1 SDL-Spezifikation des Daemon Game

Das Daemon Game ist ein Spiel für mehrere Spieler. Ein sogenannter Daemon in der Systemumgebung des Daemon Games erzeugt fortlaufend *Bump*-Signale und sendet diese an das System. Ein Spieler kann raten, ob die Zahl der bisher erzeugten *Bump*-Signale gerade oder ungerade ist. Dazu sendet er ein *Probe*-Signal an das System. Ist die Zahl der *Bump*-Signale gerade, so erhält er als Systemreaktion das Signal *Win* zurück und sein Punktestand wird um eine Einheit erhöht. Ist dagegen die Zahl der *Bump*-Signale ungerade, so erhält er das Signal *Lose*; sein Punktestand wird um eine Einheit verringert.

Das System besteht aus einem SDL-Prozeß *Monitor* und beliebig vielen SDL-Prozessen des Typs *Game*. Damit mehrere Spieler gleichzeitig spielen können, erzeugt der Prozeß *Monitor* für jeden Spieler eine Instanz des SDL-Prozesses *Game*, der mit dem Spieler in direktem Kontakt steht. Der *Monitor*-Prozeß übernimmt die Rolle einer zentralen Einheit, die die *Bump*-Signale des Daemon an alle aktiven *Game*-Prozesse weiterleitet und darüber hinaus für die Verteilung der Spieler auf die einzelnen *Game*-Prozesse verantwortlich ist.

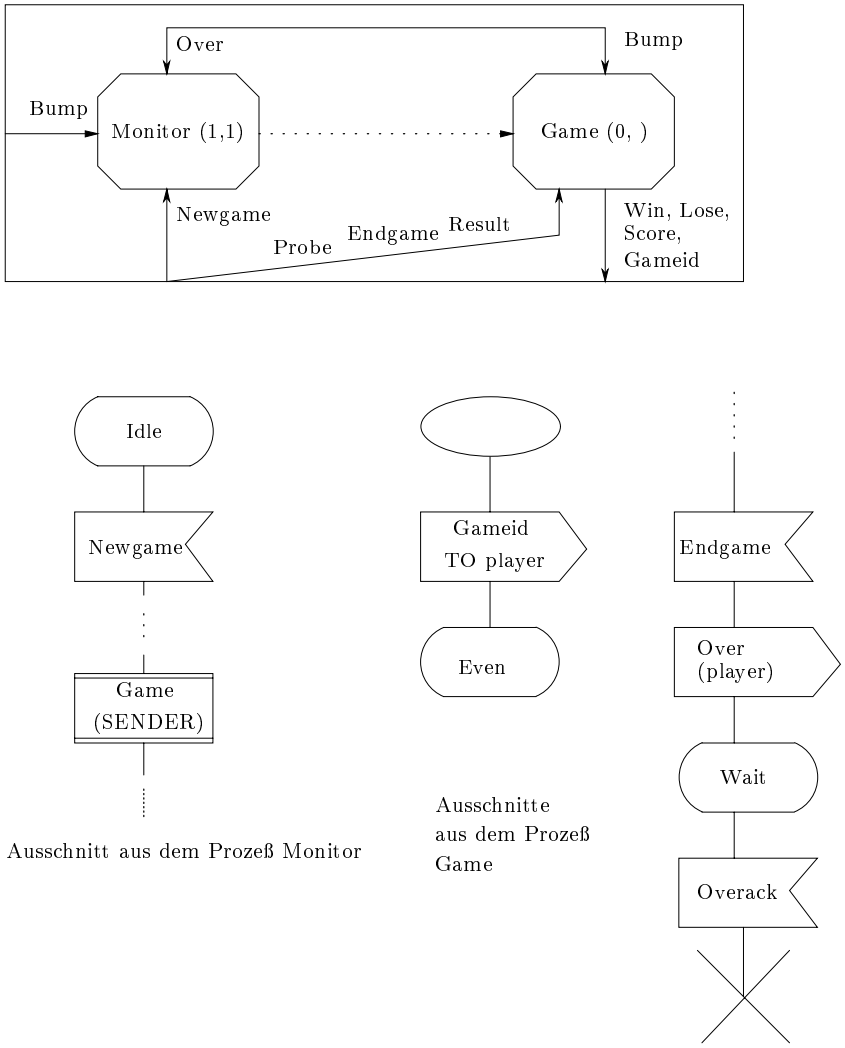


Abbildung A.1: Ausschnitte aus der SDL-Spezifikation des Daemon Game

Mit dem Signal *Newgame* meldet sich ein neuer Spieler beim System an. Mit dem Signal *Endgame* beendet er seine Teilnahme am Spiel.

Die Abbildungen A.2 bis A.5 zeigen die vollständige SDL-Spezifikation des Daemon Game. Abbildung A.1 zeigt diejenigen Ausschnitte aus der Spezifikation, die für die dynamische Prozeßerzeugung relevant sind. Wie aus der Spezifikation ersichtlich ist, kann der Prozeß *Monitor* unbegrenzt viele Prozesse des Typs *Game* erzeugen. Für jeden neuen Spieler wird eine Instanz des SDL-Prozesses *Game* erzeugt. Diese erhält als Parameter die Adresse des Spielers und nimmt darüber Kontakt mit diesem auf. Jede Prozeßinstanz *Game* ist über einen bidirektionalen Signalweg mit dem Prozeß *Monitor* verbunden. Sendet der Spieler *player* das Signal *Endgame*, so benachrichtigt die Prozeßinstanz *Game* den Prozeß *Monitor* über das Signal *Over*(*player*), damit dieser keine *Bump*-Signale mehr an ihn sendet. Nach der Bestätigung durch den Prozeß *Monitor* beendet sich die Prozeßinstanz.

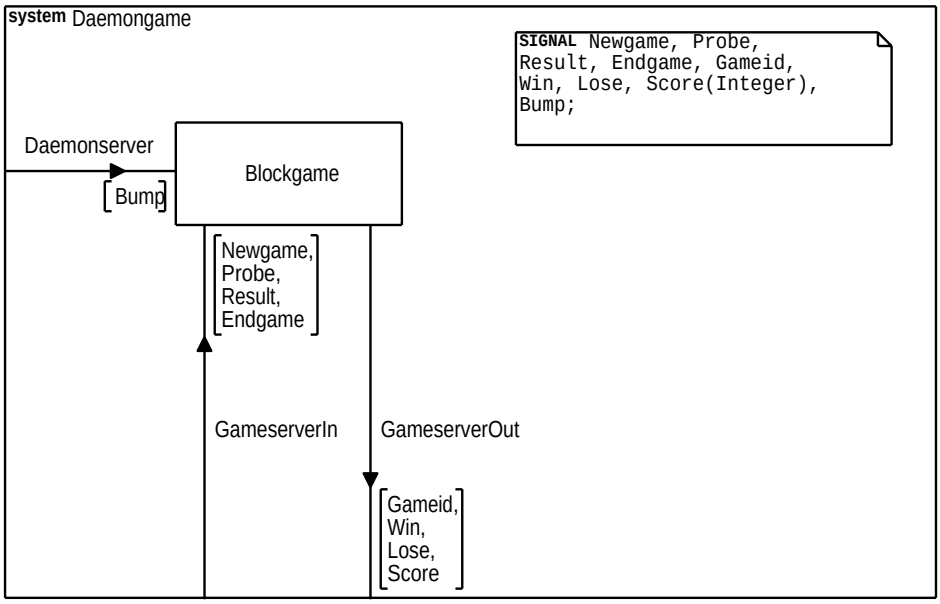


Abbildung A.2: Daemon Game: Systemebene

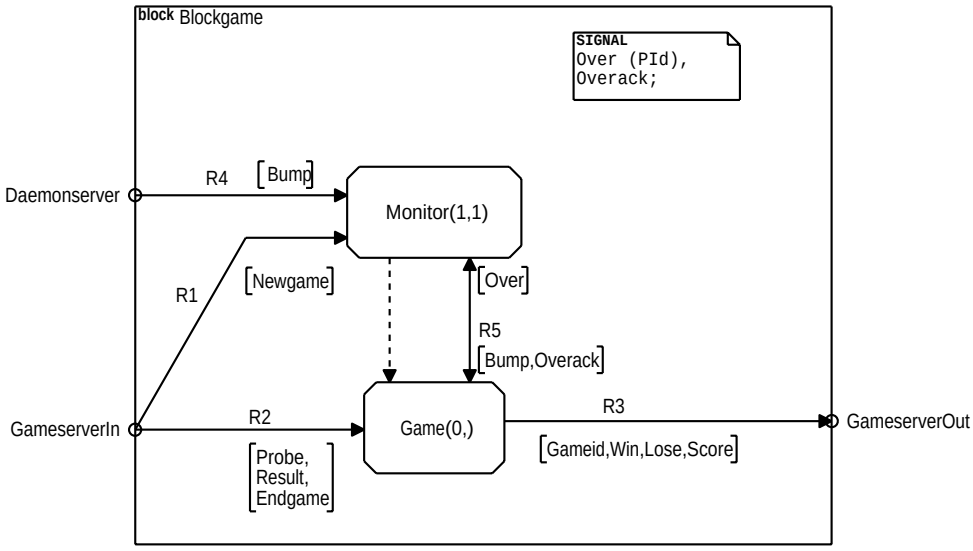


Abbildung A.3: Daemon Game: Blockebene

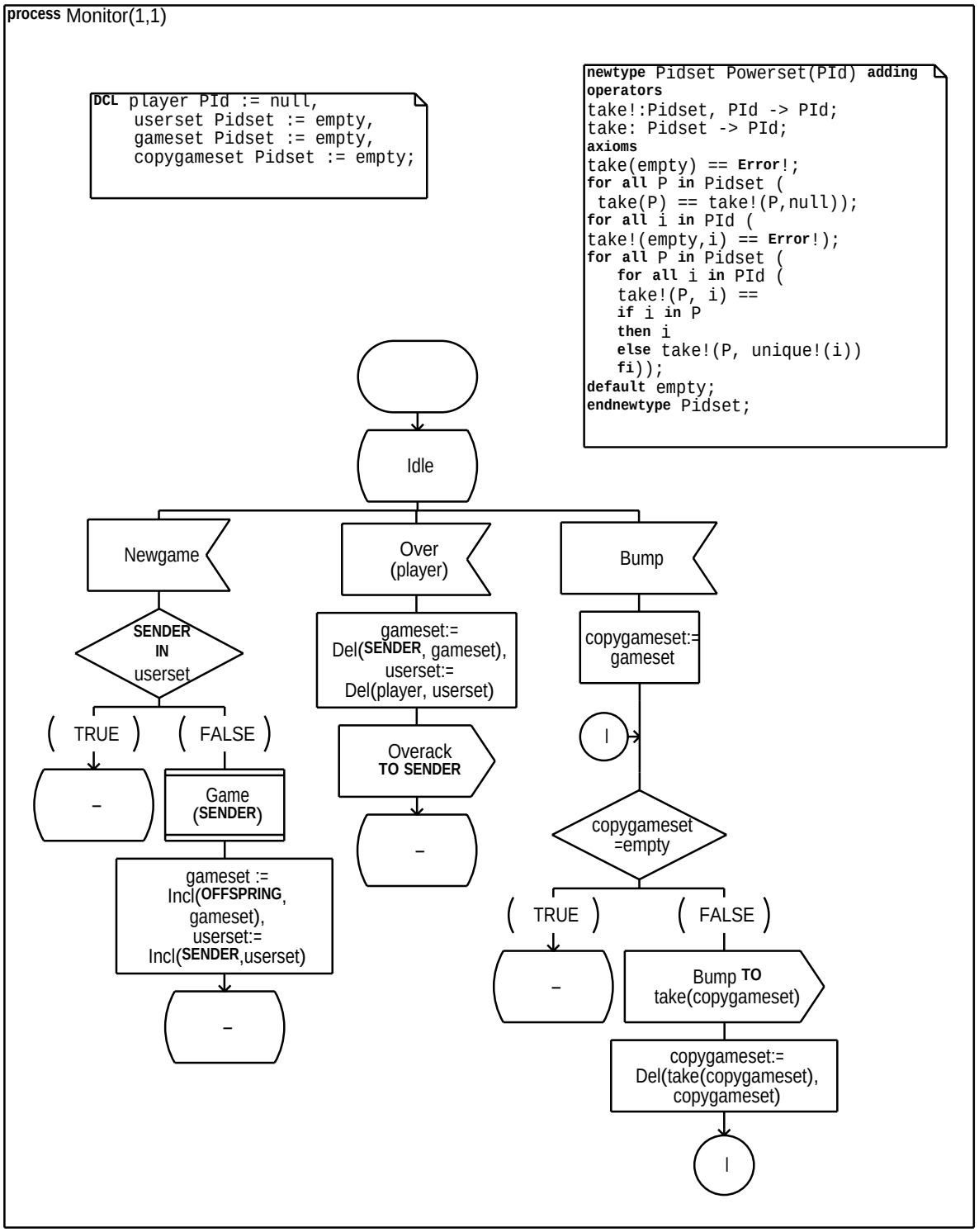


Abbildung A.4: Daemon Game: Prozeß Monitor

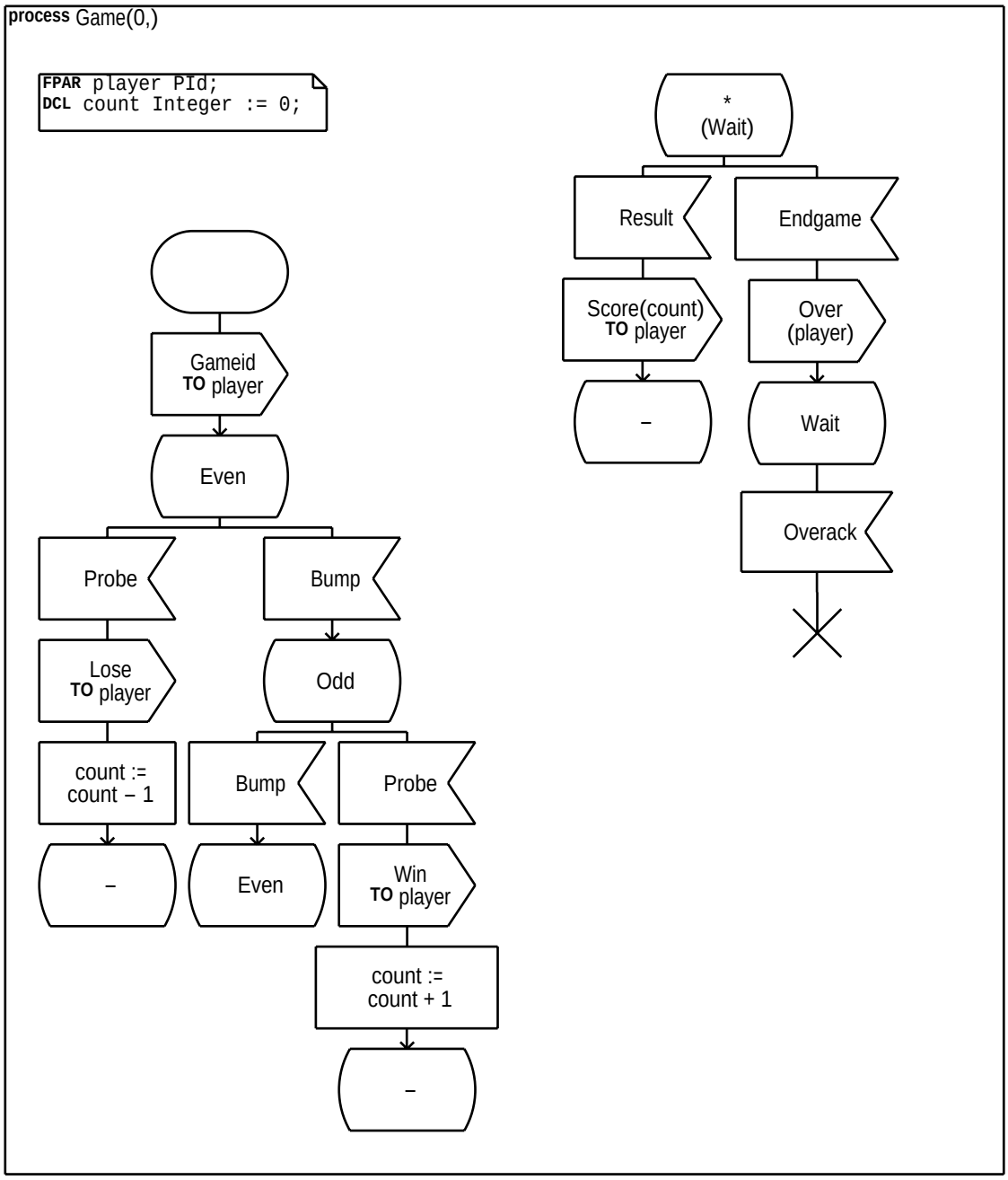


Abbildung A.5: Daemon Game: Prozeß Game

A.2 Dynamische Entwicklung der Systemstruktur

Unser Ziel ist es, die SDL-Spezifikation des Daemon Game gemäß der in Kapitel 5 entwickelten Semantikdefinition in eine FOCUS-Systemspezifikation zu übertragen. Da Spezifikationen in FOCUS eine formale Semantik besitzen, ist somit eine semantische Fundierung der SDL-Spezifikation des Daemon Game gewonnen.
Bevor wir in Abschnitt A.3 die formale Semantik für die Spezifikation angegeben, stellen wir in diesem Abschnitt eine Vorgehensweise vor, die die dynamische Entwicklung der Systemstruktur veranschaulicht. Dies erleichtert das Verständnis der Semantikdefinition und macht zudem deutlich, wie sehr die Struktur eines SDL-Systems durch die Prozeßerzeugung beeinflußt wird.

Die dynamische Entwicklung der Systemstruktur des Daemon Game beschreiben wir in FOCUS durch die Angabe mehrerer Phasen. Unter einer Phase verstehen wir einen Systemzustand, in dem sich weder die Vernetzung der Komponenten untereinander durch Kanäle noch die Anzahl der im System existierenden Komponenten verändern. Das System verhält sich in den einzelnen Phasen also wie ein statisches System. Für jede dieser Phasen beschreiben wir das Verhalten des Systems informell und bestimmen die jeweils aktuellen Mengen der aktiven und privaten Ports, *ap* und *pp*, für die aktuell vorhandenen Komponenten. Dadurch wird die dynamische Veränderung der Kommunikationsverbindungen zwischen den Komponenten greifbar.

Abbildung A.6 zeigt die Modellierung des Systems *Daemon Game* als FOCUS-Netzwerk bestehend aus den Komponenten *PR-Game*$_i$ und *PR-Monitor* und einige Phasen aus dem Systemablauf. Im Gegensatz zur SDL-Spezifikation wird die dynamische Veränderung der Systemkonfiguration deutlich erkennbar. Die SDL-Signalwege $R1, \ldots, R4$ aus der SDL-Spezifikation werden mittels des FOCUS-Kanals *min* erfaßt.

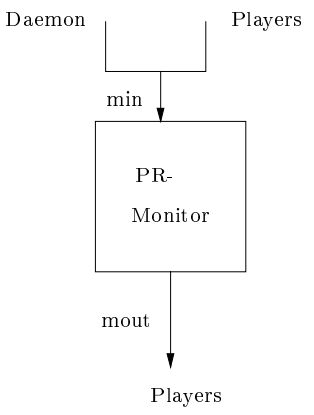
In Tabelle A.1 geben wir für alle im Verlauf des Systems vorkommenden Komponenten die initialen Schnittstellen (I und O) und die privaten Kanalnamen P an. Die Nachrichtenbelegungen der einzelnen Kanäle sind in Tabelle A.2 enthalten. Dabei bezeichnet K die Menge aller im System vorhandenen Kanalnamen:

$$K = \{min,\ mout\} \cup \{res_i,\ input_i \mid i \in Nat\}$$

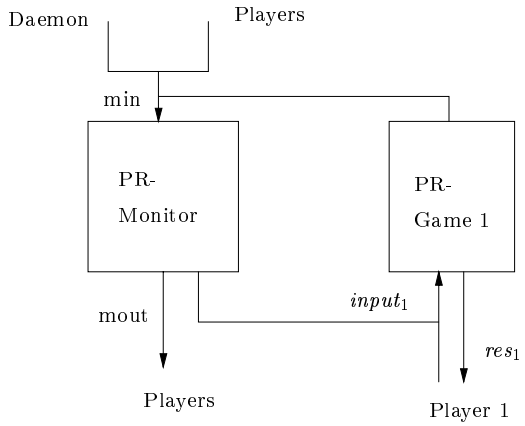
Komponente	I	O	P
PR-Monitor	*min*	*mout*	$\{res_i, input_i \mid i : Nat\}$
$\forall\, i : Nat$: *PR-Game*$_i$	$input_i$	res_i, min	–

Tabelle A.1: Initiale Schnittstellen und private Kanalnamen

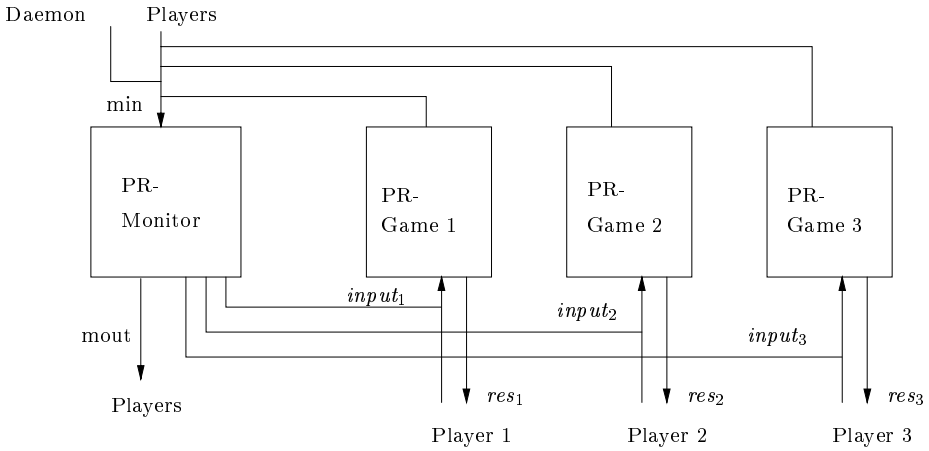
Phase 1 zeigt die Anfangskonfiguration des Systems *Daemon Game*. Die Komponente *PR-Monitor* verfügt über den Eingabekanal *min*, auf dem sie die *Bump*-Signale des Daemons und die Spielanfragen *Newgame* der Benutzer erhält. Im Gegensatz zur SDL-Spezifikation liegt über den Kanal *mout* auch eine Verbindung zwischen *PR-Monitor* und der Umgebung vor, über die *PR-Monitor* Ports an die Umgebung senden kann (siehe Phase 2). *PR-Monitor*



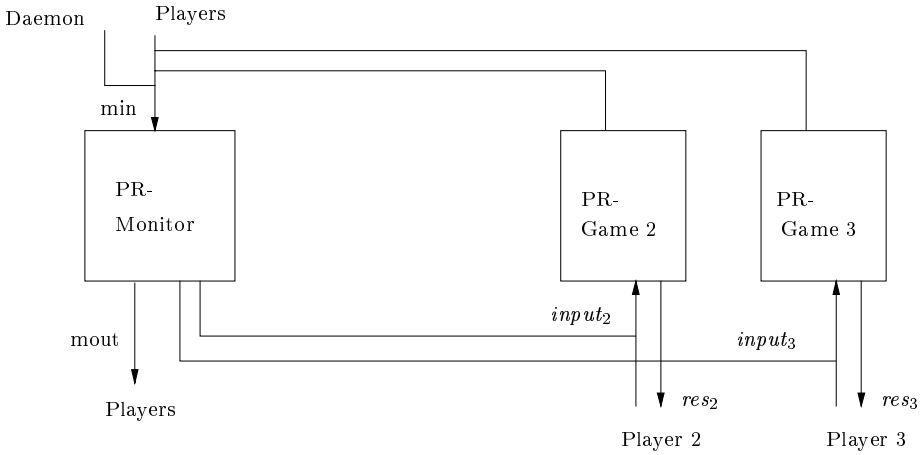
Phase 1



Phase 2



Phase 3



Phase 4

Abbildung A.6: Phasen des Daemon-Game (modelliert in FOCUS)

Kanal $k \in K$	Nachrichten Sig_k
min	$MIN = \{Newgame\ (Pid) \mid Pid : Nat\} \cup \{Bump\} \cup$
	$\{Over\ (i, j) \mid i, j : Nat\} \cup ?!K$
$mout$	$MOUT = ?!K$
$\forall\, i : Nat :$	
$input_i$	$INPUT = \{Bump, Overack\} \cup \{Probe, Result, Endgame\} \cup ?!K$
res_i	$RES = \{Gameid, Win, Lose, Score\ (count) \mid count : Int\} \cup ?!K$

Tabelle A.2: Kanäle und Nachrichten

verfügt zusätzlich über eine Menge von privaten Ports, die sie für die Einbindung von neu erzeugten Komponenten $PR\text{-}Game_i$ in die Systemstruktur benötigt.

Phase 1	ap	pp
PR-Monitor	$?min, !mout$	$\{?!input_i, ?!res_i \mid i : Nat, i \geq 1\}$

Phase 2 gibt die Systemstruktur wieder, nachdem das Signal *Newgame* aus der Systemumgebung eingetroffen ist und die Komponente *PR-Monitor* die Komponente $PR\text{-}Game_1$ erzeugt hat. Um eine Verbindung zwischen dem Spieler und der neu erzeugten $PR\text{-}Game_1$-Komponente aufzubauen, erzeugt *PR-Monitor* die Kanäle res_1 und $input_1$. *PR-Monitor* sendet die Ports $?res_1$ und $!input_1$ über den Kanal *mout* an die Systemumgebung und übergibt die Gegenstücke $!res_1$ und $?input_1$ als Parameter an $PR\text{-}Game_1$. Damit werden zwei direkte Kanalverbindungen zwischen $PR\text{-}Game_1$ und der Systemumgebung geschaffen. Es ist Aufgabe der Systemumgebung sicherzustellen, daß diese Kanalverbindungen vom entsprechend berechtigten Spieler genutzt werden. Für den Kanal $input_1$ verfügt auch *PR-Monitor* über ein Schreibrecht, so daß nun die Spieler und *PR-Monitor* auf diesem Kanal Nachrichten an $PR\text{-}Game_1$ senden können.

Ferner erhält $PR\text{-}Game_1$ das Schreibrecht auf Kanal *min*, so daß neben dem Daemon und den Spielern aus der Umgebung auch $PR\text{-}Game_1$ auf Kanal *min* schreiben kann.

Phase 2	ap	pp
PR-Monitor	$?min, !mout, !input_1$	$\{?!input_i, ?!res_i \mid i : Nat, i \geq 2\}$
$PR\text{-}Game_1$	$?input_1, !min, !res_1$	–

Phase 3 beinhaltet eine Systemkonfiguration, in der bereits drei Spieler vorhanden sind. Für jeden Spieler existiert eine Komponente des Typs *PR-Game*, die jeweils Kanalverbindungen zur Systemumgebung und zur Komponenten *PR-Monitor* besitzt. Es wird deutlich, daß sich die Menge der privaten Ports von *PR-Monitor* verringert hat; die Zahl der im System öffentlichen Kanalverbindungen hat dagegen zugenommen.

Phase 3	ap	pp
PR-Monitor	$?min, !mout,$ $!input_1, !input_2, !input_3$	$\{?!input_i, ?!res_i \mid i : Nat, i \geq 4\}$
PR-Game$_1$	$?input_1, !min, !res_1$	–
PR-Game$_2$	$?input_2, !min, !res_2$	–
PR-Game$_3$	$?input_3, !min, !res_3$	–

Phase 4 zeigt eine Systemkonfiguration, in der nur noch zwei Spieler vorhanden sind. Der Spieler *Player* 1 hat das Signal *Endgame* an die Komponente *PR-Game*$_1$ gesendet. Diese benachrichtigt daraufhin die Komponente *PR-Monitor*. In der SDL-Spezifikation terminiert der Prozeß *PR-Game*, sobald er die Bestätigung von *PR-Monitor* erhalten hat. Das Löschen einer Komponente können wir in FOCUS allerdings nicht direkt modellieren. Wir können jedoch die Schnittstelle der Komponente löschen, so daß die Komponente weder Signale empfangen noch versenden kann. Dies ist gleichbedeutend mit der Terminierung der Komponente, da eine Komponente ohne Schnittstelle kein Verhalten nach außen zeigt.

PR-Game$_1$ sendet an *PR-Monitor* die Ports $!min$ und $?input_1$, so daß die Verbindung zu *PR-Monitor* gelöscht wird. Die Kanalverbindung zwischen *PR-Game*$_1$ und der Systemumgebung lösen wir wie folgt auf: *PR-Game*$_1$ sendet den Port $!res_1$ zusammen mit den anderen Ports an *PR-Monitor*. Von der Systemumgebung verlangen wir, daß diese zusammen mit dem Signal *Endgame* die Ports $?res_1$ und $!input$ ebenfalls an *PR-Monitor* sendet. Damit besitzt *PR-Monitor* alle verfügbaren Schreib- und Leserechte zu den Kanälen $input_1$ und res_1, so daß diese Kanalverbindungen gelöscht und zu privaten Ports von *PR-Monitor* werden. In SDL wird keine Aussage darüber getroffen, was mit Kanälen geschieht, die mit einer nicht mehr vorhandenen Komponente verbunden sind.

Phase 4	ap	pp
PR-Monitor	$?min, !mout,$ $!input_2, !input_3$	$\{?!res_i, ?!input_i \mid$ $i : Nat, i = 1 \vee i \geq 4\}$
PR-Game$_1$	–	–
PR-Game$_2$	$?input_2, !min, !res_2$	–
PR-Game$_3$	$?input_3, !min, !res_3$	–

A.3 Formale Semantik für Daemon Game

In den folgenden Abschnitten geben wir die formale Semantik für die initiale Systemkonfiguration sowie für die Komponenten *PR-Monitor* und *PR-Game*$_i$ an. Wir verwenden dabei das in [GS97] eingeführte Spezifikationsformat `m2m` für zeitunabhängige mobile Komponenten mit many-to-many Kommunikation[1]. Im oberen Teil der Spezifikation werden die initialen Ein- und Ausgabekanäle einer Komponente (den Schlüsselworten `in` bzw. `out` folgend) sowie, falls vorhanden, die Menge der verfügbaren privaten Kanalnamen mit den jeweils dazugehörigen Nachrichtenmengen angegeben (dem Schlüsselwort `priv` folgend). Im

[1] Die logische Kernsprache ANDL ist für die Spezifikation von Systemen mit point-to-point Kommunikation ausgerichtet und kann deshalb nicht verwendet werden.

unteren Teil der Spezifikation erfolgt bei einem Netzwerk die Angabe, aus welchen Komponenten sich das Netz zusammensetzt (siehe Spezifikation von *Daemon Game*, Seite 160), bei einer Basiskomponenten die Verhaltensdefinition (siehe Spezifikation von *PR_Monitor*, Seite 162).

A.3.1 Formale Spezifikation der Systemstruktur

Die initiale Systemkonfiguration des *Daemon Game* besteht aus der Komponente *PR-Monitor*, die über die Kanäle *min* und *mout* mit der Systemumgebung verbunden ist und über die privaten Ports $\{?!res_i,\ ?!input_i \mid i : Nat,\ i \geq 1\}$ verfügt (siehe Phase 1 in Abbildung A.6). Es ergibt sich folgende Spezifikation des Systems:

DaemonGame — m2m

in $min : MIN$

out $mout : MOUT$

$PR\_Monitor\,(min \ \triangleright \ mout \ \Box \ \{res_i, input_i \mid i : Nat\})$

A.3.2 Formale Spezifikation der Komponente *PR-Monitor*

Die SDL-Spezifikation des SDL-Prozesses *Monitor* verändern wir wie folgt:

- In der informellen Beschreibung des Spiels heißt es, daß das System jedem Benutzer einen eindeutigen Identifikator zuweist; in der SDL-Spezifikation greift der Prozeß *Monitor* auf diesen Identifikator über den Ausdruck *Sender* zu. Da nicht näher beschrieben wird, wie diese Identifikatoren den Spielern zugewiesen werden, nehmen wir folgende Vereinfachung vor: Das Signal *Newgame*, mit dem sich ein neuer Spieler beim Spiel anmeldet, führt als Parameter den Identifikator des Spielers bereits mit sich. Damit ist die Systemumgebung für die Zuweisung der Identifikatoren zuständig. Diese Änderung hat keinen Einfluß auf das eigentliche Verhalten des Spiels.
- Wir verzichten auf die Einführung einer lokalen Variablen, die den eindeutigen Identifikator speichert. Der Identifikator wird aus dem Signal *Newgame* extrahiert, an die neu erzeugte Komponente der Art *PR_Game* weitergereicht und in die Menge der aktiven Spieler in *PR-Monitor* aufgenommen; eine zusätzliche Speicherung in *PR-Monitor* ist nicht notwendig.
- Durch das implizite Mischen werden alle Signale *Over*, die *PR-Monitor* über den Kanal *min* erhält, zu einem Strom zusammengeführt, so daß die Information verloren geht, welche Komponente des Typs *PR-Game* jeweils Absender der einzelnen Signale

Over ist. Deshalb erhält das Signal *Over* einen zweiten Parameter. Der erste Parameter gibt den Identifikator des Spielers an, der mit der Komponente $PR\text{-}Game_i$ in Kontakt steht (wie in der SDL-Spezifikation); über den zweiten Parameter wird der Index i der Komponente $PR\text{-}Game_i$ festgehalten (siehe Anmerkung 2 auf Seite 92).

Im folgenden geben wir die formale Spezifikation von *PR_Monitor* an (siehe Seite 162). Das Verhalten der Komponente leitet sich aus den Zustandsübergängen des SDL-Prozesses *Monitor* ab. Bei der Definition der Funktionsgleichungen gehen wir wie in Abschnitt 5.2 beschrieben vor und verwenden für die Prozeßerzeugung das Schema für die Erzeugung einer Menge von Prozeßinstanzen (siehe Abschnitt 5.2.3.2). Da die Menge der gleichzeitig existierenden SDL-Prozeßinstanzen *Game* nicht nach oben beschränkt ist, verwenden wir das Schema ohne den Zähler c für die maximale Anzahl von Prozeßinstanzen.

Die SDL-Datentypen *PId* und *Pidset* bilden wir gemäß der Tabelle in Abschnitt 4.2 auf die SPECTRUM-Datentypen *Nat* und $Set(Nat)$ ab. Die Definition des Datentyps D für den internen Datenzustand von *PR-Monitor* lautet:

$$data\ D = md\,(Uset : Set(Nat), Gset : Set(Nat), i : Nat)$$

- *Uset* entspricht der Variablen *userset* in der SDL-Spezifikation, die die Identifikatoren aller aktiven Spieler beinhaltet.
- *Gset* entspricht der Variablen *gameset*, die aus den Identifikatoren aller aktiven Prozesse des Typs *PR-Game* besteht.
- Die Variable i wird aus dem Schema für die Prozeßerzeugung übernommen und dient zur Numerierung der Komponenten vom Typ *PR-Game*.

Die Nachrichtenmengen für die Kanäle sind wie in Tabelle A.2 definiert.

(1): Die Funktion *Idle* wird aus dem Startsymbol des SDL-Prozesses abgeleitet und liefert, angewandt auf die Kommunikationsgeschichte *in* des Eingabekanals *min*, die Kommunikationsgeschichte *out* für den Ausgabekanal *mout*. Dabei stützt sie sich auf den initialen Datenzustand σ_{init} ab, der sich aus dem Textsymbol des SDL-Prozesses *Monitor* ablesen läßt.

(2): Hier erfolgt die All-Quantifizierung sämtlicher Variablen, die in den darauffolgenden Gleichungen verwendet werden. H_A ist der Typ der ungezeiteten Ströme, die gemäß der Typisierung der Eingabekanäle von *PR-Monitor* als Eingabestrom verarbeitet werden (siehe [GS97] für die formale Definition von H_A).

(3): Diese Zeile enthält die Forderung, daß für jede zu erzeugende Komponente $PR\text{-}Game_j$ eine Funktion existiert, die ihr Verhalten beschreibt; d.h. es existiert eine Funktion $game_j$, die das Prädikat $PR\text{-}Game_j$ erfüllt.

(4) – **(7)**: Wie bei der Definition der Semantik für statische SDL-Systeme werden die Zustandsübergänge des SDL-Prozesses auf Funktionsgleichungen abgebildet.

PR − Monitor m2m

in $min : MIN$

out $mout : MOUT$

priv $\{res_k : RES, input_k : INPUT \mid k : Nat\}$

(1) $Idle\ [\sigma_{init}]\ (\mathsf{in}) = \mathsf{out}$ with $\sigma_{init} = \mathrm{md}\ (\varnothing, \varnothing, 1)$

(2) where $\forall\, s, s' : H_A,\ \sigma : D,\ Pid, l : Nat :$

(3) $\forall\, j : Nat :\ \exists\ game_j \in [\![\ PR - Game_j\ ]\!] :$

(4) $s = \{min \mapsto Newgame(Pid)\}\ \&\ s' \Longrightarrow$
$Idle\ [\sigma]\ (s)\ =\ \mathsf{if}\ Pid \in Uset(\sigma)$
$\mathsf{then}\ Idle\ [\sigma]\ (s')$
$\mathsf{else}\ \{mout \mapsto ?res_{id}\ \&\ !input_{id}\}\ \&$
$(Idle[\sigma']\ \overline{\oplus}\ game_{id}\ (\{!res_{id}, ?input_{id}, !min\},\ Pid))\ (s')$
endif
with $id = i(\sigma)$,
$\sigma'\ =\ \sigma[Uset := Uset \cup \{Pid\},\ Gset := Gset \cup \{id\},\ i := i + 1]$

(5) $s = \{min \mapsto Over(Pid, l)\}\ \&\ s' \Longrightarrow$
$Idle\ [\sigma]\ (s)\ =\ \{input_l \mapsto Overack\}\ \&\ Idle\ [\sigma']\ (s')$
with $\sigma' = \sigma\ [Gset := Gset \setminus \{l\},\ Uset := Uset \setminus \{Pid\}]$

(6) $s = \{min \mapsto Bump\}\ \&\ s' \Longrightarrow$
$Idle\ [\sigma]\ (s)\ =\ \{input_{j1} \mapsto Bump, \ldots, input_{jk} \mapsto Bump\}\ \&\ Idle\ [\sigma]\ (s')$
with $\{j1, \ldots, jk\} = Gset(\sigma)$

(7) $\forall\ \beta\ \in\ ?!K\ :\ s = \{min \to \beta\}\ \&\ s' \Longrightarrow$
$Idle\ [\sigma]\ (s)\ =\ Idle\ [\sigma]\ (s')$

(4): In dieser Gleichung erfolgt die Generierung der Komponente $PR\text{-}Game_{id}$ gemäß unseres Schemas von Seite 91. Der aktuelle Index id wird aus dem Datenzustand mittels $i(\sigma)$ bestimmt. Als Parameter erhält die neu erzeugte Komponente (wie in der SDL-Spezifikation) den Identifikator Pid des neu angemeldeten Spielers. Zusätzlich werden eine Menge von Ports übergeben ($!res_{id},\ ?input_{id},\ !min$), die die Schnittstelle von $PR\text{-}Game_{id}$ zum Spieler in der Systemumgebung und zur Komponente *PR-Monitor* bilden. Die Systemumgebung erhält über den Kanal $mout$ das Schreibrecht für den Kanal $input_{id}$ und das Leserecht für den Kanal res_{id} zugeteilt.

(5): Mit dem Signal $Over(Pid,\ l)$ meldet die Komponente $PR\text{-}Game_{id}$ das Ende ihres Spiels. *PR-Monitor* bestätigt über das Signal *Overack* das Spielende. Der Parameter l des empfangenen Signals gibt dabei an, über welchen Kanal des Typs *input* die Ausgabe zu erfolgen hat. Anschließend werden die Einträge *Uset* und *Gset* des Datenzustands

aktualisiert: Der Spieler *Pid* wird aus der Menge der aktiven Spieler *Uset* gelöscht; die Prozeßinstanz l wird aus der Menge der aktiven Prozeßinstanzen *Gset* gelöscht.

(6): *PR-Monitor* erhält vom Daemon in der Systemumgebung das Signal *Bump* und leitet es über die Kanäle $input_{j1}$ bis $input_{jk}$ an alle aktiven Komponenten des Typs *PR-Game* weiter. Die aktuelle Menge $Gset(\sigma)$ enthält die Indizes aller aktiven Komponenten des Typs *PR-Game* und damit die Indizes für die Kanäle *input*.

(7): Ports, die die Komponente *PR-Monitor* auf ihrem Eingabekanal empfängt, werden automatisch in die Mengen *ap* und *pp* aufgenommen; eine explizite Spezifikation dieses Vorgangs ist nicht erforderlich.

Die Spezifikation von impliziten Transitionen entfällt, da der SDL-Prozeß über nur einen Zustand verfügt und für jedes mögliche Eingabesignal ein Zustandsübergang vorliegt.

A.3.3 Formale Spezifikation der Komponente *PR-Game*

Das Verhalten der Komponente $PR\text{-}Game_i$ wird durch all diejenigen mobilen Funktionen spezifiziert, die das zugehörige Prädikat $PR\text{-}Game_i$ erfüllen. Der Aufbau der Spezifikation (siehe nächste Seite) gleicht dem der Spezifikation von *PR-Monitor*.

(1): Die Funktion *Game*, angewandt auf *in*, liefert die Ausgabe *out*. Das Prädikat $PR\text{-}Game_i$ verfügt zusätzlich über einen formalen Parameter *player* des Typs *Nat*, der den Identifikator des neu angemeldeten Spielers speichert. Beim Erzeugen der Komponente $PR\text{-}Game_i$ wird der aktuelle Parameter von *PR-Monitor* direkt an $PR\text{-}Game_i$ übergeben (siehe Gleichung (4) in der Spezifikation von *PR-Monitor*).

(2): Diese Gleichung modelliert die Starttransition von $PR\text{-}Game_i$, in der über den Kanal res_i das Signal *Gameid* ausgegeben wird. Anschließend wird die Funktion *Even* mit dem Eingabestrom aufgerufen. Sie verfügt neben dem Parameter *player* noch über eine Variable *sc*, die den aktuellen Punktestand des Spielers speichert, der anfangs 0 beträgt. Da es sich insgesamt nur um zwei Variablen handelt, verzichten wir auf die explizite Einführung eines Datenzustands σ.

(3): Hier erfolgt die All-Quantifizierung sämtlicher Variablen, die in den darauffolgenden Gleichungen verwendet werden.

(4) - (12): Für jede Transition des SDL-Prozesses wird eine Funktionsgleichung aufgestellt.

(12): Sobald die Komponente $PR\text{-}Game_i$ von *PR-Monitor* die Bestätigung des Spielendes erhalten hat (über das Signal *Overack*), löst sie ihre Kanalverbindungen zum Spieler in der Systemumgebung und zur Komponente *PR-Monitor* auf. Für die Komponente $PR\text{-}Game_i$ bedeutet dies, daß sie auf dem Kanal *min* die Ports $!res_i$, $?input_i$ und $!min$ an *PR-Monitor* zurücksendet. Das Terminieren des SDL-Prozesses modellieren wir, indem die Komponente auf keine weiteren Eingabenachrichten mehr reagiert und keine weiteren Ausgabenachrichten mehr erzeugt (Ausgabe des leeren Stroms $\langle\rangle$, vgl. Abschnitt 4.3.2.3).

(13): Hier werden für jeden Zustand die impliziten Transitionen definiert.

PR − Game_i (player) m2m

in $input_i : INPUT_i$

out $res_i : RES_i$

(1) $Game(player)\ (\text{in}) = \text{out}$

(2) with $Game(player)\ (\text{in}) = \{res_i \mapsto Gameid\}\ \&\ Even\ [player, 0]\ (\text{in})$

(3) where $\forall\, sc : Int,\ \forall\, s, s' : H_A :$

(4) $s = \{input_i \mapsto Probe\}\ \&\ s' \Longrightarrow$
$Even\ [player, sc]\ (s) \ = \ \{res_i \mapsto Lose\}\ \&\ Even\ [player, sc - 1]\ (s')$

(5) $s = \{input_i \mapsto Bump\}\ \&\ s' \Longrightarrow$
$Even\ [player, sc]\ (s) \ = \ Odd\ [player, sc]\ (s')$

(6) $s = \{input_i \mapsto Probe\}\ \&\ s' \Longrightarrow$
$Odd\ [player, sc]\ (s) \ = \ \{res_i \mapsto Win\}\ \&\ Odd\ [player, sc + 1]\ (s')$

(7) $s = \{input_i \mapsto Bump\}\ \&\ s' \Longrightarrow$
$Odd\ [player, sc]\ (s) \ = \ Even\ [player, sc]\ (s')$

(8) $s = \{input_i \mapsto Result\}\ \&\ s' \Longrightarrow$
$Even\ [player, sc]\ (s) \ = \ \{res_i \mapsto Score(sc)\}\ \&\ Even\ [player, sc]\ (s')$

(9) $s = \{input_i \mapsto Result\}\ \&\ s' \Longrightarrow$
$Odd\ [player, sc]\ (s) \ = \ \{res_i \mapsto Score(sc)\}\ \&\ Odd\ [player, sc]\ (s')$

(10) $s = \{input_i \mapsto Endgame\}\ \&\ s' \Longrightarrow$
$Even\ [player, sc]\ (s) \ = \ \{min \mapsto Over(player, i)\}\ \&\ Wait\ [player, sc]\ (s')$

(11) $s = \{input_i \mapsto Endgame\}\ \&\ s' \Longrightarrow$
$Odd\ [player, sc]\ (s) \ = \ \{min \mapsto Over(player, i)\}\ \&\ Wait\ [player, sc]\ (s')$

(12) $s = \{input_i \mapsto Overack\}\ \&\ s' \Longrightarrow$
$Wait\ [player, sc]\ (s) \ = \ \{min \mapsto !res_i\ \&\ ?input_i\ \&\ !min\}\ \&\ \langle\rangle$

(13) $\forall\, \alpha \notin (\{Probe,\ Bump,\ Result,\ Endgame\}\ \cup\ ?!K) \Longrightarrow$
$Odd\ [player, sc]\ (\{input_i \mapsto \alpha\}\ \&\ s') \ = \ Odd\ [player, sc]\ (s')$

$\forall\, \alpha \notin (\{Probe,\ Bump,\ Result,\ Endgame\}\ \cup\ ?!K) \Longrightarrow$
$Even\ [player, sc]\ (\{input_i \mapsto \alpha\}\ \&\ s') \ = \ Even\ [player, sc]\ (s')$

$\forall\, \alpha \notin (\{Overack\}\ \cup\ ?!K) \Longrightarrow$
$Wait\ [player, sc]\ (\{input_i \mapsto \alpha\}\ \&\ s') \ = \ Wait\ [player, sc]\ (s')$

Anhang B

Fallstudie: Alternating Bit Protokoll

B.1 Formale Semantik

Im folgenden geben wir die semantische Bedeutung der FOCUS-Spezifikation des Alternating Bit Protokolls auf der mathematisch-logischen Ebene von FOCUS an. Auf dieser Ebene wird das Verhalten der SDL-Spezifikation durch eine Menge von stark pulsgetriebenen Funktionen definiert, die über die Angabe des Prädikats `is_abp` bestimmt wird. Dieses Prädikat wird aus der ANDL-Komponente *ABP* abgeleitet und stützt sich auf die Prädikate ab, die aus den übrigen Komponenten des Systems abgeleitet werden.

Nachrichtenmengen:

$$
\begin{aligned}
In_{sys} &= \{din\ (d)\ \mid\ d : Nat\} \\
Out_{sys} &= \{dout\ (d)\ \mid\ d : Nat\} \\
In_{med1} &= \{snd\ (d,\ b)\ \mid\ d : Nat,\ b\ : Bit\} \\
Out_{med1} &= \{snd\ (d,\ b)\ \mid\ d : Nat,\ b : Bit\} \cup \{err\} \\
In_{med2} &= \{ack\ (b)\ \mid\ b : Bit\} \\
Out_{med2} &= \{ack\ (b)\ \mid\ b : Bit\} \cup \{err\} \\
S_{in} &= In_{sys} \cup S_{med2} \\
R_{out} &= Out_{sys} \cup S_{med2}
\end{aligned}
$$

Typisierung der Prädikate:

$$
\begin{aligned}
is\_abp : &\quad (In_{sys}^{\infty} \rightarrow Out_{sys}^{\infty}) \rightarrow Bool \\
is\_med1 : &\quad (In_{med1}^{\infty} \rightarrow Out_{med1}^{\infty}) \rightarrow Bool \\
is\_med2 : &\quad (In_{med2}^{\infty} \rightarrow Out_{med2}^{\infty}) \rightarrow Bool \\
is\_transmitter : &\quad (In_{sys}^{\infty} \times Out_{med2}^{\infty} \rightarrow In_{med1}^{\infty}) \rightarrow Bool
\end{aligned}
$$

$is\_receiver$: $(Out^{\infty}_{med1} \to Out^{\infty}_{sys} \times In^{\infty}_{med2}) \to Bool$

is_fair_merge : $(In^{\infty}_{sys} \times Out^{\infty}_{med2} \to S^{\infty}_{in}) \to Bool$

is_split : $(R^{\infty}_{out} \to Out^{\infty}_{sys} \times In^{\infty}_{med2}) \to Bool$

$is\_pr\_transmitter$: $(S^{\infty}_{in} \to In^{\infty}_{med1}) \to Bool$

$is\_pr\_receiver$: $(Out^{\infty}_{med1} \to R^{\infty}_{out}) \to Bool$

Spezifikation der Prädikate:

is_abp (f) $\stackrel{def}{=}$

$$\begin{aligned}
&\exists\ S,\ R,\ M1,\ M2 :\\
&\quad is_transmitter\ (S)\ \wedge\ is_receiver\ (R)\ \wedge\ is_med1\ (M1)\ \wedge\ is_med2\ (M2)\ \wedge\\
&\quad \forall\ data_in,\ data_out.\\
&\qquad f\ (data_in)\ =\ data_out \longrightarrow \exists\ s_1,\ s_2,\ s_3,\ s_4.\\
&\qquad\quad S\ (data_in, s_4)\ =\ s_1\ \wedge\\
&\qquad\qquad M1\ (s_1)\ =\ s_2\ \wedge\\
&\qquad\qquad\quad R\ (s_2)\ =\ (data_out,\ s_3)\ \wedge\\
&\qquad\qquad M2\ (s_3)\ =\ s_4
\end{aligned}$$

is_transmitter (f) $\stackrel{def}{=}$

$$\begin{aligned}
&\exists\ FM,\ PR_Transmitter.\\
&\quad is_fair_merge\ (FM)\ \wedge\ is_pr_transmitter\ (PR_Transmitter)\ \wedge\\
&\quad \forall\ data_in,\ s_4,\ s_1.\ f\ (data_in, s_4)\ =\ s_1 \longrightarrow \exists\ in.\\
&\qquad\quad FM\ (data_in,\ s_4)\ =\ in\ \wedge\\
&\qquad PR_Transmitter\ (in)\ =\ s_1
\end{aligned}$$

is_receiver (f) $\stackrel{def}{=}$

$$\begin{aligned}
&\exists\ PR_Receiver,\ Split.\\
&\quad is_split\ (Split)\ \wedge\ is_pr_receiver\ (PR_Receiver)\ \wedge\\
&\quad \forall\ s_2,\ s_3,\ data_out.\ f\ (s_2)\ =\ (data_out,\ s_3) \longrightarrow \exists\ out.\\
&\qquad PR_Receiver\ (s_2)\ =\ out\ \wedge\\
&\qquad\quad Split\ (out)\ =\ (data_out,\ s_3)
\end{aligned}$$

is_med1 (f) $\stackrel{def}{=}$

$$\begin{aligned}
&\forall\ s_1,\ s_2.\quad f\ (s_1)\ =\ s_2 \longrightarrow \\
&\qquad \exists\ h : Bit^{\infty} \times In^{\omega}_{med1} \to Out^{\omega}_{med1},\ \exists\ p : Bit^{\infty}. \\
&\qquad\quad \{L\} \copyright\ p\ =\ \infty\ \wedge\ \overline{f\ (s_1)}\ =\ h\ (p,\ \overline{s_1})\ \wedge \\
&\qquad\quad \forall\ s : In^{\omega}_{med1},\ \forall\ m : In_{med1},\ \forall\ p : Bit^{\infty} : \\
&\qquad\qquad h\ (O\ \&\ p,\ m\ \&\ s)\ =\ err\ \&\ h\ (p,\ s)\ \wedge \\
&\qquad\qquad h\ (L\ \&\ p,\ m\ \&\ s)\ =\ m\ \&\ h\ (p,\ s)
\end{aligned}$$

is_med2 (f) $\stackrel{def}{=}$

$$\begin{aligned}
&\forall\ s_3,\ s_4.\quad f\ (s_3)\ =\ s_4 \longrightarrow \\
&\qquad \exists\ h : Bit^{\infty} \times In^{\omega}_{med2} \to Out^{\omega}_{med2},\ \exists\ p : Bit^{\infty}. \\
&\qquad\quad \{L\} \copyright\ p\ =\ \infty\ \wedge\ \overline{f\ (s_3)}\ =\ h\ (p,\ \overline{s_3})\ \wedge \\
&\qquad\quad \forall\ s : In^{\omega}_{med2},\ \forall\ m : In_{med2},\ \forall\ p : Bit^{\infty} : \\
&\qquad\qquad h\ (O\ \&\ p,\ m\ \&\ s)\ =\ err\ \&\ h\ (p,\ s)\ \wedge \\
&\qquad\qquad h\ (L\ \&\ p,\ m\ \&\ s)\ =\ m\ \&\ h\ (p,\ s)
\end{aligned}$$

is_fair_merge (f) $\stackrel{def}{=}$

$$\begin{aligned}
&\forall\ data_in,\ s_4,\ in.\quad f\ (data_in,\ s_4)\ =\ in \longrightarrow \\
&\qquad\qquad (In_{sys} \cup \{\surd\})\ \copyright\ in\ =\ data_in\ \wedge \\
&\qquad\qquad (Out_{med2} \cup \{\surd\})\ \copyright\ in\ =\ s_4
\end{aligned}$$

is_split (f) $\stackrel{def}{=}$

$$\begin{aligned}
&\forall\ out,\ data_out,\ s_3 :\quad f\ (out)\ =\ (data_out,\ s_3) \longrightarrow \\
&\qquad\qquad (Out_{sys} \cup \{\surd\})\ \copyright\ out\ =\ data_out\ \wedge \\
&\qquad\qquad (In_{med2} \cup \{\surd\})\ \copyright\ out\ =\ s_3
\end{aligned}$$

is_pr_transmitter (f) $\stackrel{def}{=}$

$\forall\ in,\ out.\quad f(in) = out \longrightarrow$

$\exists\ idle:\ Bit \times Nat \times S^{\omega}_{in} \to In^{\omega}_{med1}:\ \overline{f(in)} = idle\ [0,\ L]\ (\overline{in}) \wedge$

$\forall\ s : S^{\omega}_{in}.\ \forall\, a,\ b : Bit,\ \forall\ d,\ data : Nat.$

$idle\ [d,\ b]\ (din\ (data)\ \&\ s) = snd\ (data,\ b)\ \&\ ackwait\ [data,\ b]\ (s) \wedge$

$idle\ [d,\ b]\ (ack\ (a)\ \&\ s) = idle\ [d,\ b]\ (s) \wedge$

$idle\ [d,\ b]\ (err\ \&\ s) =\ idle\ [d,\ b]\ (s)$

$ackwait\ [d,\ b]\ (s) =$
case $search(\{ack\ (a),\ err\},\ s)$ of
  $ack\ (a)$: if $a = b$
    then $idle\ [d,\ \neg b]\ (del(ack(a), s))$
    else $snd\ (d,\ b)\ \&\ ackwait\ [d,\ b]\ (del(ack(a), s))$
    endif
  err : $snd\ (d,\ b)\ \&\ ackwait\ [d,\ b]\ (del(err, s))$
endcase

is_pr_receiver (f) $\stackrel{def}{=}$

$\forall\ in,\ out.\quad f\ (in) = out \longrightarrow$

$\exists\ waiting:\ Bit \times Out^{\omega}_{med1} \to R^{\omega}_{out}:\ \overline{f(in)} = waiting\ [0]\ (\overline{in}) \wedge$

$\forall\ s : Out^{\omega}_{med1},\ \forall\ rb,\ sb : Bit,\ \forall\ data : Nat.$

$waiting\ [rb]\ (snd\ (data,\ sb)\ \&\ s) =$
if $rb = sb$
then $ack\ (sb)\ \&\ dout\ (data)\ \&\ waiting\ [\neg rb]\ (s)$
else $ack\ (sb)\ \&\ waiting\ [rb]\ (s)$
endif $\wedge$

$waiting\ [rb]\ (err\ \&\ s) = ack\ (\neg rb)\ \&\ waiting\ [rb]\ (s)$

B.2 Lemmata für den Korrektheitsbeweis

Es folgen die Beweise von Lemma 1 und Lemma 2, die für den Korrektheitsbeweis des Alternating Bit Protokolls in Abschnitt 6.1.5 erforderlich sind. Für die Beweisschritte verwenden wir im wesentlichen die Beziehungen zwischen den Strömen, die sich aus den Gleichungssystemen ergeben, sowie die Funktionsgleichungen für die Funktionen $idle'$, $ackwait'$, $rsend$, $rack$ (siehe Seite 107) und med_1, med_2 (siehe Seite 105) aus den Spezifikationen der Basiskomponenten.

Lemma 1

Seien die Voraussetzungen wie beim Theorem (siehe Abschnitt 6.1.5), also gelte das Gleichungssystem

$$\left.\begin{array}{rcl} sin & = & merge\ (in,\ arecv) \wedge \\ \overline{dsend} & = & idle'\ [b]\ (\overline{sin}) \wedge \\ \overline{drecv} & = & med_1\ (p_1,\ \overline{dsend}) \wedge \\ \overline{out} & = & rsend\ [b]\ (\overline{drecv}) \wedge \\ \overline{asend} & = & rack\ [b]\ (\overline{drecv}) \wedge \\ \overline{arecv} & = & med_2\ (p_2,\ \overline{asend}) \end{array}\right\} (*)$$

mit $\overline{in} = din(m)\ \&\ \overline{in'}$

Dann existieren out', sin', $dsend'$, $drecv'$, $asend'$, $arecv'$, $p_1{}'$, $p_2{}'$ mit

$$\left.\begin{array}{rcl} sin' & = & merge\ (in',\ arecv') \wedge \\ \overline{dsend'} & = & ackwait'\ [\langle\rangle,\ m,\ b]\ (\overline{sin'}) \wedge \\ \overline{drecv'} & = & med_1\ (p_1{}',\ \overline{dsend'}) \wedge \\ \overline{asend'} & = & rack\ [\neg b]\ (\overline{drecv'}) \wedge \\ \overline{out'} & = & rsend\ [\neg b]\ (\overline{drecv'}) \wedge \\ \overline{arecv'} & = & med_2\ (p_2{}',\ ack(b)\ \&\ \overline{asend'}) \end{array}\right\} (**)$$

und $\overline{out} = dout(m)\ \&\ \overline{out'}$

Nachdem der Sender die erste Nachricht $snd(m, b)$ ausgegeben hat, wird ein Protokollzustand erreicht, in dem der Empfänger die Nachricht $dout(m)$ erfolgreich an die Umgebung ausgegeben und sein Kontrollbit aktualisiert hat (auf $\neg b$). Die erforderliche Bestätigung $ack(b)$ für die Nachricht ist jedoch noch nicht vom Sender empfangen worden. Sie bildet die erste Nachricht des Eingabestroms von Medium 2.

Beweis:

Aufgrund der Voraussetzungen gilt: $\{L\}©p_1 = \infty$. Somit erfolgt nach endlich vielen fehlerhaften Übertragungen garantiert eine korrekte Übertragung durch das Medium1, also: $\exists\, p : Bit^*.\ p = O^* \circ \langle L \rangle,\ \exists\ \tilde{p_1} : Bit^\infty.\ \{L\}©\tilde{p_1} = \infty$ mit $p_1 = p \circ \tilde{p_1}$

Wir zeigen die Korrektheit des Lemmas mit struktureller Induktion über der Sequenz p.

Induktionsanfang:

$p = \langle L \rangle$ und damit $p_1 = L\ \&\ \tilde{p_1}$, d.h. das Medium überträgt die erste Nachricht des Senders auf Anhieb korrekt.

Damit folgern wir aus dem Gleichungssystem $(*)$:

$sin = merge\ (in, arecv)$ mit $ft(\overline{sin}) = ft(\overline{in}) = din(m)$, da Medium2 vom Empfänger noch keine Nachricht erhalten und somit noch keine Ausgabe erzeugt hat (folgt aus Definition von med_2 und dem zeitsynchronen Mischen der Ströme in und $arecv$ durch $merge$).

$$\Longrightarrow \exists\, s, sin' : sin = s \circ sin' \wedge \overline{s} = din(m) \wedge sin' = merge\ (in', arecv)$$

$$\overline{dsend} = idle'\ [b]\ (din(m)\ \&\ \overline{sin'}) = snd(m,b)\ \&\ ackwait'\ [\langle\rangle,\ m,\ b]\ (\overline{sin'})$$

$$\Longrightarrow \exists\, ds, dsend' : \underline{dsend = ds \circ dsend'} \wedge \overline{ds} = snd(m,b) \wedge \overline{dsend'} = ackwait'\ [\langle\rangle,\ m,\ b]\ (\overline{sin'})$$

$$\overline{drecv} = med_1\ (L\ \&\ \tilde{p_1},\ snd(m,b)\ \&\ \overline{dsend'}) = snd(m,b)\ \&\ med_1\ (\tilde{p_1},\ \overline{dsend'})$$

$$\Longrightarrow \exists\, dr, drecv' : drecv = dr \circ drecv' \wedge \overline{dr} = snd(m,b) \wedge \overline{drecv'} = med_1\ (\tilde{p_1},\ \overline{dsend'})$$

$$\overline{asend} = rack\ [b]\ (snd(m,b)\ \&\ \overline{drecv'}) = ack(b)\ \&\ rack\ [\neg b]\ (\overline{drecv'})$$

$$\Longrightarrow \exists\, as, asend' : asend = as \circ asend' \wedge \overline{as} = ack(b) \wedge \overline{asend'} = rack\ [\neg b]\ (\overline{drecv'})$$

$$\overline{out} = rsend\ [b]\ (snd(m,b)\ \&\ \overline{drecv'}) = dout(m)\ \&\ rsend\ [\neg b]\ (\overline{drecv'})$$

$$\Longrightarrow \exists\, os, out' : out = os \circ out' \wedge \overline{os} = dout(m) \wedge \overline{out'} = rsend\ [\neg b]\ (\overline{drecv'})$$

Des weiteren definieren wir: $arecv' \stackrel{def}{=} arecv,\ {p_1}' \stackrel{def}{=} \tilde{p_1}$ und ${p_2}' \stackrel{def}{=} p_2$.

Damit folgt: $\overline{arecv'} = med_2\ ({p_2}', ack(b)\ \&\ \overline{asend'})$

Somit ist die Existenz von $sin', dsend', drecv', asend', arecv', out'$ gezeigt
und es gilt: $\overline{out} = dout(m)\ \&\ \overline{out'}$.

Induktionsschritt:

Induktionsannahme: Die Aussage gilt für p.
Wir zeigen: die Aussage gilt für $O\ \&\ p$ und damit für $p_1 = (O\ \&\ p) \circ \tilde{p_1}$, d.h. die erste Nachricht des Senders wird von Medium1 fehlerhaft übertragen.

$sin = merge\ (in,\ arecv)$ mit $ft(\overline{sin}) = ft(\overline{in}) = din(m)$, da Medium2 vom Empfänger noch keine Nachricht erhalten und somit noch keine Ausgabe erzeugt hat (folgt aus Definition von med_2 und dem zeitsynchronen Mischen der Ströme in und $arecv$ durch $merge$).

$$\Longrightarrow \exists\, s, sin^* : sin = s \circ sin^* \wedge \overline{s} = din(m) \wedge sin^* = merge\ (in',\ arecv)$$

$$\overline{dsend} = idle\ [b]\ (din(m)\ \&\ \overline{sin}) = snd(m,b)\ \&\ ackwait\ [\langle\rangle,\ m,\ b]\ (\overline{sin^*})$$

$$\Longrightarrow \exists\, ds, dsend^* : \begin{array}{l} dsend = ds \circ dsend^* \wedge \overline{ds} = snd(m,b) \wedge \\ \overline{dsend^*} = ackwait'\ [\langle\rangle,\ m,\ b]\ (\overline{sin^*}) \end{array}$$

$$\overline{drecv} = med_1\ ((O\ \&\ p) \circ \tilde{p_1},\ snd(m,b)\ \&\ \overline{dsend^*}) = err\ \&\ med_1\ (p \circ \tilde{p_1},\ \overline{dsend^*})$$

$$\Longrightarrow \exists\, dr, drecv^* : drecv = dr \circ drecv^* \wedge \overline{dr} = err \wedge \overline{drecv^*} = med_1\ (p \circ \tilde{p_1},\ \overline{dsend^*})$$

$$\overline{out} = rsend\ [b]\ (err\ \&\ \overline{drecv^*}) = rsend\ [b]\ (\overline{drecv^*})$$

$$\overline{asend} = rack\ [b]\ (err\ \&\ \overline{drecv^*}) = ack(\neg b)\ \&\ rack\ [b]\ (\overline{drecv^*})$$

$$\Longrightarrow \exists\, as, asend^* : asend = as \circ asend^* \wedge \overline{as} = ack(\neg b) \wedge \overline{asend^*} = rack\ [b]\ (\overline{drecv^*})$$

$$\overline{arecv} = med_2\ (p_2,\ ack(\neg b)\ \&\ \overline{asend^*}) = e\ \&\ med_2\ (rt(p_2), \overline{asend^*}),\ e \in \{err,\ ack(\neg b)\}$$

Wir definieren: ${p_2}^* \stackrel{def}{=} rt(p_2)$.

$$\Longrightarrow \exists\, ar, arecv^* : \overline{ar} = e \wedge \overline{arecv^*} = med_2\ ({p_2}^*,\ \overline{asend^*})$$

Wir erhalten folgenden Zusammenhang:

$$sin^* = merge\ (in',\ arecv)\ \wedge\ \overline{dsend^*} = ackwait'\ [\langle\rangle,\ m,\ b]\ (\overline{sin^*})$$

Es gilt (siehe oben): $\overline{arecv} = e\ \&\ \overline{arecv^*}$
Wir führen eine Fallunterscheidung über $\overline{sin^*}$ durch:
Entweder die Nachricht e ist die erste Nachricht des Stroms $\overline{sin^*}$ oder es treten zunächst endlich, aber beliebig viele Eingabenachrichten $din(m),\ m : Nat$ auf:

Fall 1: $\overline{sin^*} = e\ \&\ \overline{t}$ mit $e \in \{err,\ ack(\neg b)\}$ und $t = merge\ (in',\ arecv^*)$

$\overline{dsend^*} = ackwait'\ [\langle\rangle,\ m,\ b]\ (\overline{sin^*})$
$= snd(m,b)\ \&\ ackwait'\ [\langle\rangle,\ m,\ b]\ (\overline{merge\ (in',\ arecv^*)})$
$= idle'\ [b]\ (din(m)\ \&\ \overline{merge\ (in',\ arecv^*)})$ (mit Definition von $idle'$)
$= idle'\ [b]\ (\overline{merge\ (in,\ arecv^*)})$

Wir definieren: $sin^{**} = merge\ (in,\ arecv^*)$

Fall 2: $\overline{sin^*} = (din(m_1)\ \&\ \ldots\ \&\ din(m_k)\ \&\ e) \circ \overline{t}$ mit $e \in \{err,\ ack(\neg b)\}$ und
$t = merge\ (rin,\ arecv^*)$ mit $(din(m_1)\ \&\ \ldots\ \&\ din(m_k)) \circ \overline{rin} = \overline{in'}$ und $m_1, \ldots, m_k : Nat$

$\overline{dsend^*} = ackwait'\ [\langle\rangle,\ m,\ b]\ (\overline{sin^*})$
$= ackwait'\ [din(m_1)\ \&\ \ldots\ \&\ din(m_k),\ m,\ b]\ (e\ \&\ \overline{t})$

$= snd(m, b)$ & $ackwait'\ [din(m_1)$ & $\ldots$ & $din(m_k),\ m,\ b]\ (\overline{t})$
$= idle'\ [b]\ (din(m)$ & $din(m_1)$ & $\ldots$ & $din(m_k)$ & $\overline{t})$ (mit Definition von $idle'$)
$= idle'\ [b]\ (\overline{merge\ (in,\ arecv^*)})$

Wir definieren: $sin^{**} = merge\ (in,\ arecv^*)$

In beiden Fällen folgt damit: $dsend^* = idle'\ [b]\ (\overline{sin^{**}})$

Damit erhalten wir aus obigem Gleichungssystem:

$$\begin{aligned}
sin^{**} &= merge\ (in,\ arecv^*)\ \wedge \\
\overline{dsend^*} &= idle'\ [b]\ (\overline{sin^{**}})\ \wedge \\
\overline{drecv^*} &= med_1\ (p \circ \tilde{p_1},\ \overline{dsend^*})\ \wedge \\
\overline{asend^*} &= rack\ [b]\ (\overline{drecv^*})\ \wedge \\
\overline{out} &= rsend\ [b]\ (\overline{drecv^*})\ \wedge \\
\overline{arecv^*} &= med_2\ (p_2^*,\ \overline{asend^*})
\end{aligned}$$

Nun wenden wir die Induktionsannahme über p an und folgern damit:
$\exists\, sin',\ dsend',\ drecv',\ asend',\ arecv',\ out',\ {p_1}',\ {p_2}'$, so daß das Gleichungssystem $(**)$ erfüllt ist und es gilt: $\overline{out} = dout(m)$ & $\overline{out'}$

□

Lemma 2

Seien die Voraussetzungen wie bei Lemma 1 (siehe Seite 169).
Es existieren $out',\ sin',\ dsend',\ drecv',\ asend',\ arecv',\ {p_1}',\ {p_2}'$ mit

$$\left.\begin{aligned}
sin' &= merge\ (in',\ arecv')\ \wedge \\
\overline{dsend'} &= ackwait'\ [\langle\rangle,\ m,\ b]\ (\overline{sin'})\ \wedge \\
\overline{drecv'} &= med_1\ ({p_1}',\ \overline{dsend'})\ \wedge \\
\overline{asend'} &= rack\ [\neg b]\ (\overline{drecv'})\ \wedge \\
\overline{out'} &= rsend\ [\neg b]\ (\overline{drecv'})\ \wedge \\
\overline{arecv'} &= med_2\ ({p_2}',\ ack(b)\ \&\ \overline{asend'})
\end{aligned}\right\} (**)$$

und $\overline{out} = dout(m)$ & $\overline{out'}$

Dann existieren sin'', $dsend''$, $drecv''$, $asend''$, $arecv''$, ${p_1}''$, ${p_2}''$ mit

$$\left.\begin{array}{rcl} sin'' & = & merge\ (in',\ arecv'') \wedge \\ \overline{dsend''} & = & idle'\ [\neg b]\ (\overline{sin''}) \wedge \\ \overline{drecv''} & = & med_1\ ({p_1}'',\ \overline{dsend''}) \wedge \\ \overline{asend''} & = & rack\ [\neg b]\ (\overline{drecv''}) \wedge \\ \overline{out'} & = & rsend\ [\neg b]\ (\overline{drecv''}) \wedge \\ \overline{arecv''} & = & med_2\ ({p_2}'',\ \overline{asend''}) \end{array}\right\} (***)$$

Es wird von einem Systemzustand ausgegangen, bei dem der Empfänger die erste Nachricht $dout(m)$ des Eingabestroms erfolgreich an die Umgebung gesendet und der Sender sein Kontrollbit geändert, aber die erforderliche Bestätigung des Empfängers noch nicht empfangen hat. Der erreichte Zustand ähnelt dem Anfangszustand des Protokolls vor der Übertragung der ersten Nachricht $din(m)$. Jedoch haben sich die Kontrollbits sowohl beim Sender als auch beim Empfänger geändert und der Eingabestrom aus der Umgebung ist um die erste Nachricht verringert.

Beweis:

Aufgrund der Voraussetzungen gilt: $\{L\}©p_2 = \infty$. Somit erfolgt nach endlich vielen fehlerhaften Übertragungen garantiert eine korrekte Übertragung durch das Medium2, also:

$\exists\, p : Bit^*.\ p = 0^* \circ \langle L \rangle,\ \exists\ \tilde{p_2} : Bit^\infty.\ \{L\}©\tilde{p_2} = \infty$ mit $p_2 = p \circ \tilde{p_2}$.

Wir zeigen die Korrektheit des Lemmas durch strukturelle Induktion über der Sequenz p.

Induktionsanfang:

$p = \langle L \rangle$ und damit $p_2 = L\ \&\ \tilde{p_2}$, d.h. das Medium2 überträgt die erste Nachricht mit dem Kontrollbit des Empfängers auf Anhieb korrekt.

Damit folgt aus dem Gleichungssystem $(**)$:

$\overline{arecv'} = med_2\ (L\ \&\ \tilde{p_2},\ ack(b)\ \&\ \overline{asend'}) = ack(b)\ \&\ med_2\ (\tilde{p_2},\ \overline{asend'})$

$\quad \Longrightarrow \exists\, ar, arecv'' : arecv' = ar \circ arecv'' \wedge \overline{ar} = ack(b) \wedge \overline{arecv''} = med_2\ (\tilde{p_2},\ \overline{asend'})$

Des weiteren definieren wir:

$dsend'' \stackrel{def}{=} dsend' \ \wedge\ drecv'' \stackrel{def}{=} drecv' \ \wedge\ asend'' \stackrel{def}{=} asend' \wedge$
${p_1}'' \stackrel{def}{=} {p_1}' \ \wedge\ {p_2}'' \stackrel{def}{=} {p_2}'$

Es gilt mit $(**)$: $sin' = merge\ (in',\ arecv')$.
Mit $\overline{arecv'} = ack(b)\ \&\ \overline{arecv''}$ folgt folgende Fallunterscheidung über $\overline{sin'}$: Entweder die Nachricht $ack(b)$ ist die erste Nachricht des Stroms, oder es treten zunächst endlich, aber beliebig viele Eingabenachrichten $din(m), m : Nat$ auf:

Fall 1: $\exists\, sin'' : \overline{sin'} = ack(b)\ \&\ \overline{sin''}$ mit $sin'' = merge\ (in',\ arecv'')$

$\overline{dsend''} = ackwait'\ [\langle\rangle,\ m,\ b]\ (ack(b)\ \&\ \overline{sin''}) = idle'\ [\neg b]\ (\overline{sin''})$

Fall 2: $\overline{sin'} = (din(m_1)\ \&\ \ldots\ \&\ din(m_k)\ \&\ ack(b)) \circ \overline{t}$
mit $t = merge(rin, arecv'')$ und $(din(m_1)\ \&\ \ldots\ \&\ din(m_k)) \circ \overline{rin} = \overline{in'}$

$\overline{dsend''} = ackwait'\ [\langle\rangle,\ m,\ b]\ (din(m_1)\ \&\ \ldots\ \&\ din(m_k)\ \&\ ack(b)\ \&\ \overline{t})$
$= ackwait'\ [din(m_1)\ \&\ \ldots\ \&\ din(m_k),\ m,\ b]\ (ack(b)\ \&\ \overline{t})$
$= idle\ [\neg b]\ ((din(m_1)\ \&\ \ldots\ \&\ din(m_k)) \circ \overline{t}) = idle\ [\neg b]\ (\overline{merge\ (in',\ arecv'')})$

Wir definieren: $sin'' = merge\ (in', arecv'')$

In beiden Fällen gilt: $sin'' = merge\ (in', arecv'')\ \wedge\ dsend'' = idle\ [\neg b]\ (\overline{sin''})$

Wir erhalten folgendes Gleichungssystem:

$$\begin{aligned}
sin'' &= merge\ (in',\ arecv'')\ \wedge \\
\overline{dsend''} &= idle'\ [\neg b]\ (\overline{sin''})\ \wedge \\
\overline{drecv''} &= med_1\ ({p_1}'',\ \overline{dsend''})\ \wedge \\
\overline{asend''} &= rack\ [\neg b]\ (\overline{drecv''})\ \wedge \\
\overline{out'} &= rsend\ [\neg b]\ (\overline{drecv''})\ \wedge \\
\overline{arecv''} &= med_2\ ({p_2}'',\ \overline{asend''})
\end{aligned}$$

Damit sind das Gleichungssystem $(***)$ erfüllt und die Aussage bewiesen. Das Kontrollbit ist nun bei Sender und Empfänger alterniert, der Eingabestrom ist um die erste Nachricht verringert, der Sender befindet sich wieder im Zustand `idle`, charakterisiert durch die Funktion $idle'$.

Induktionsschritt:

Induktionsannahme: Die Aussage gilt für p.
Wir zeigen: die Aussage gilt für $O\&p$ und damit für $p_2 = (O\&p) \circ \tilde{p_2}$, d.h. die erste Nachricht des Empfängers wird von Medium2 fehlerhaft übertragen.

$\overline{arecv'} = med_2\ ((O\ \&\ p) \circ \tilde{p_2},\ ack(b)\ \&\ \overline{asend'}) = err\ \&\ med_2\ (p \circ \tilde{p_2},\ \overline{asend'})$

$\Longrightarrow \exists\, ar, arecv^* : arecv' = ar \circ arecv^* \wedge \overline{ar} = err \wedge \overline{arecv^*} = med_2\ (p \circ \tilde{p_2},\ \overline{asend'})$

$sin' = merge\ (in',\ arecv')$ mit $ft(\overline{arecv'}) = err$

Wir führen folgende Fallunterscheidung über $\overline{sin'}$ durch: Entweder die Nachricht err ist die erste Nachricht des Stroms oder es treten zunächst endlich, aber beliebig viele Eingabenachrichten $din(m), m : Nat$ auf:

Fall 1: $\overline{sin'} = err\ \&\ \overline{sin^*}$

$\overline{dsend'} = ackwait'\ [\langle\rangle,\ m,\ b]\ (err\ \&\ \overline{sin^*}) = snd(m,b)\ \&\ ackwait'\ [\langle\rangle,\ m,\ b]\ (\overline{sin^*})$
mit $sin^* = merge\ (in',\ arecv^*)$

Fall 2: $\overline{sin'} = (din(m_1)\ \&\ \ldots\ \&\ din(m_k)\ \&\ err) \circ \overline{t}$

$t = merge\ (rin,\ arecv^*)$ mit $(din(m_1)\ \&\ \ldots\ \&\ din(m_k)) \circ \overline{rin} = \overline{in'}$

$\overline{dsend'} = ackwait'\ [\langle\rangle,\ m,\ b]\ (din(m_1)\ \&\ \ldots\ \&\ din(m_k)\ \&\ err\ \&\ \overline{t})$
$= ackwait'\ [din(m_1)\ \&\ \ldots\ \&\ din(m_k),\ m,\ b]\ (err\ \&\ \overline{t})$
$= snd(m,b)\ \&\ ackwait'\ [\langle\rangle,\ m,\ b]\ ((din(m_1)\ \&\ \ldots\ \&\ din(m_k)) \circ \overline{t})$
$= snd(m,b)\ \&\ ackwait'\ [\langle\rangle,\ m,\ b]\ (\overline{merge\ (in',\ arecv^*)})$

Für beide Fälle gilt:
$sin^* = merge\ (in',\ arecv^*) \wedge \overline{dsend'} = snd(m,b)\ \&\ ackwait'\ [\langle\rangle,\ m,\ b]\ (\overline{sin^*})$

$\Longrightarrow \exists\, ds, dsend^* : dsend' = ds \circ dsend^* \wedge \overline{ds} = snd(m,b) \wedge$
$\overline{dsend^*} = ackwait'\ [\langle\rangle,\ m,\ b]\ (\overline{sin^*})$

$\overline{drecv'} = med_1\ ({p_1}',\ snd(m,b)\ \&\ \overline{dsend^*}) = e\ \&\ med_1(rt({p_1}'),\ \overline{dsend^*}),\ e \in \{snd(m,b), err\}$

$\Longrightarrow \exists\, dr, drecv^* : drecv' = dr \circ drecv^* \wedge \overline{drecv^*} = med_1\ (p_1^*,\ \overline{dsend^*})$ mit $p_1^* = rt({p_1}')$

$\overline{asend'} = rack\ [\neg b]\ (e\ \&\ \overline{drecv^*}) = ack(b)\ \&\ rack\ [\neg b]\ (\overline{drecv^*})$ für $e \in \{snd(m,b),\ err\}$

$\Longrightarrow \exists\, as, asend^* : asend' = as \circ asend^* \wedge \overline{as} = ack(b) \wedge \overline{asend^*} = rack\ [\neg b]\ (\overline{drecv^*})$

$\overline{out'} = rsend\ [\neg b]\ (e\ \&\ \overline{drecv^*}) = rsend\ [\neg b]\ (\overline{drecv^*})$ für $e \in \{snd(m,b), err\}$

wähle: out^* mit $\overline{out^*} = out'$

mit der Zerlegung von $\overline{asend'}$ folgt: $\overline{arecv^*} = med_2\ (p_2^*,\ ack(b)\ \&\ \overline{asend^*})$

Damit erhalten wir folgendes Gleichungssystem:

$$\begin{aligned}
sin^* &= merge\ (in',\ arecv^*)\ \wedge \\
\overline{dsend^*} &= ackwait'\ [\langle\rangle,\ m,\ b]\ (\overline{sin^*})\ \wedge \\
\overline{drecv^*} &= med_1\ (p_1^*,\ \overline{dsend^*})\ \wedge \\
\overline{asend^*} &= rack\ [\neg b]\ (\overline{drecv^*})\ \wedge \\
\overline{out^*} &= rsend\ [\neg b]\ (\overline{drecv^*})\ \wedge \\
\overline{arecv^*} &= med_2\ (p \circ \tilde{p_2},\ ack(b)\ \&\ \overline{asend^*})
\end{aligned}$$

Nun wenden wir die Induktionsannahme über p an und folgern damit die Existenz von sin'', $dsend''$, $drecv''$, $asend''$, $arecv''$, ${p_1}''$, ${p_2}''$, so daß das Gleichungssystem $(***)$ erfüllt ist.

□