SDL-Datenkonzepte - Analyse und Verbesserungen

Dissertation

zur Erlangung des akademischen Grades
Doktor der Naturwissenschaften (doctor rerum naturalium)
im Fach Informatik

eingereicht an der

Mathematisch-Naturwissenschaftlichen Fakultät II
der Humboldt-Universität zu Berlin

von

Diplom-Informatiker Ralf Schröder
geboren am 21.08.1966 in Luckenwalde

....
Prof. Dr. Jürgen Mlynek
Präsident der Humboldt-Universität zu Berlin

....
Prof. Dr. Elmar Kulke
Dekan der Mathematisch-Naturwissenschaftlichen Fakultät II

Gutachter/in:

1. Prof. Dr. Joachim Fischer

2. Dr. Andreas Prinz

3. Prof. Dr.-Ing. Ulrich Herzog

Tag der mündlichen Prüfung 26. März 2003

Zusammenfassung

SDL in der 1996 standardisierten Sprachversion ist zur Zeit die im Telekommunikationsbereich am weitesten verbreitete Sprache zur Spezifikation von Protokollen. Ein wesentlicher Aspekt der Sprachentwicklung seit 1988 ist auch die Verfügbarkeit einer formalen Basis semantischer Konzepte. Für das Datenkonzept der Sprache wurde auf die algebraischen Technik ACT ONE zurückgegriffen. Obwohl Anspruch als auch praktischer Wert von SDL in der Ausführbarkeit als Spezifikationstechnik liegt, wird dieses gerade durch das verwendete Datenmodell beeinträchtigt. Verdeckt wird dieses Problem durch die Bereitstellung von vordefinierten Datentypen. Durch die Erweiterung von SDL um objektorientierte Konzepte im Jahr 1992 und durch die allgemein wachsende Bedeutung der Daten in Protokollbeschreibungen treten die vorhandenen Sprachprobleme bei den Daten immer mehr in der Vordergrund. Individuelle Lösungen zur Spezifikation von Daten in verfügbaren SDL-Werkzeugen sind die Folge.

In der vorliegenden Arbeit werden sowohl die praxismotivierten Unzulänglichkeiten als auch die formalen Unstimmigkeiten im SDL-Datenkonzept aufgezeigt. Auf der Grundlage einer systematischen Analyse werden ein allgemeiner Anforderungskatalog und eine Methodik für Veränderungen am Datenkonzept erarbeitet. Zusätzlich werden wichtige Sprachmodifikationen mit dem Schwerpunkten Ausdruckskraft und Ausführbarkeit vorgestellt und bewertet. Es steht somit ein Instrumentarium zur Verfügung, das den unterschiedlichen SDL-Interessengruppen bei der Bewertung und Nutzung von SDL-Veränderungen dienlich ist.

Die in der Arbeit vorgestellten Modifikationen des Datenkonzepts basieren auf den langjährigen Erfahrungen des Autors bei der Implementierung und dem Einsatz von Werkzeugen, die mit verschiedenen projektspezifischen Zielstellungen SDL-Beschreibungen in ausführbare Programme überführen. Die Kombination von SDL mit einer weiteren Spezifikationstechnik, ASN.1, spielt hier eine besondere Rolle. Durch die aktive Mitarbeit des Autors bei der SDL-Sprachstandardisierung werden in der Arbeit auch Vorschläge präsentiert, die über das Potential der vorhandenen Werkzeuge hinausgehen. Das schließt beispielsweise die Bewertung der neuen, in der Praxis noch nicht etablierten, SDL-Version aus dem Jahr 2000 mit ein.

Summary

SDL in the language version which was standardized in 1996 is the most-used language in the telecommunication domain for the specification of protocols today. An essential aspect of the language development since 1998 is the availability of a formal basis for semantic concepts. The algebraic technique ACT ONE is used for the data concept of the language. Although the requirement and the practical value of SDL is the execution a specification technique, this is impaired straight by the used data model. The problem is hidden by the supply of pre-defined data types. Because of the introduction of object oriented concepts in 1992 and because of the generally increasing importance of data for the protocol description the existing language problems are taking more and more attention. Individual solutions for the specification of data are the consequence with available SDL tools.

In the presented document are pointed out the praxis motivated inadequacies as well as the formal discrepancy of the data concept. A general requirement catalogue and a methodology are designed for language modifications based on a systematic inspection of the SDL data concept. Furthermore important language modifications are introduced and evaluated with the focus to expression power and to execution. Instruments are provided thus, which are helpful to different SDL interest groups for the evaluation and for the application of SDL modifications.

In the document presented data modifications are based on years of experience of the author in the implementation and application of tools that compile SDL specifications with different project-specific objectives into executable programs. The combination of SDL with a further specification technology, ASN.1, plays an important role here. Because of the active role of the author in the SDL standardization process also suggestions are presented going beyond the potential of the existing tools. That includes for example the evaluation of the new, in practice not yet established SDL version, published in 2000.

Inhaltsverzeichnis

Inhaltsverzeichnis iii

Abbildungsverzeichnis vii

Einleitung 1

Kapitel 1
Die Sprache SDL 5

1.1 Historische Entwicklung von SDL 5
1.2 Wichtige SDL-Sprachkonzepte 6
1.3 Definition der Sprache SDL im Sprachstandard 10
1.4 SDL ab dem Jahr 2000 12
1.5 Die SDL-Entwicklungsumgebung SITE 14
1.5.1 Austausch von SDL-Beschreibungen 15
1.5.2 Syntaktische Analyse 16
1.5.3 Semantische Analyse 16
1.5.4 Codegenerierung 17
Codegeneratoren 17
SDL-Bibliotheken 18

Kapitel 2
SDL-Datenbeschreibungen 21

2.1 ACT ONE-Basismodell in SDL 21
2.1.1 Definitionen und Bezeichner 22
2.1.2 Signaturen und Sorten 22
2.1.3 Variablen 23
2.1.4 Terme einer Sorte 23
2.1.5 Zuweisungskompatibilität 24
2.1.6 Werte 24
2.1.7 Prinzip der partiellen Datendefinition 26
2.1.8 Grundlegende Probleme 26
2.2 Zusätzliche Datenkonzepte 27
2.2.1 Hierarchische Strukturierung von Datendefinitionen 27
2.2.2 Vererbung 29
2.2.3 Algorithmische Operatoren 31
2.2.4 Zuweisungen 33
2.2.5 Teilbereichstypen 33
2.2.6 Transformationsbasierte Ausdrücke 34
2.2.7 Strukturelle Beschreibungen 36
2.2.8 Abstrakte Sorten 36
2.2.9 Spezifikation von Fehlern 36
2.2.10 Vordefinierte Sorten 37
2.3 Externe Datenbeschreibungen 38
2.4 Schlußfolgerung 39
Anforderungskatalog für neue Datenkonzepte 39

Kapitel 3
Methoden für neue Datenkonzepte 41

3.1 Eine Systematik für neue Datenkonzepte 41
3.2 Externe Beschreibungen 42

3.2.1 Annotierte SDL-Beschreibungen 43
Vollständige Transformation 43
Einbettung 44
3.2.2 Externe Sorten 45
3.2.3 Transformation 45
3.3 Erweiterung von SDL 46
3.3.1 Sprachspezifische Erweiterungen 46
3.3.2 Sprachunabhängige Erweiterungen 46
3.4 Substitution des Datenkonzepts 47
3.4.1 Sprachspezifische Ersetzung 47
3.4.2 Neuentwurf 48
3.5 Veränderung der normativen Sprachdefinition 48

Kapitel 4
Realisierte alternative Datenkonzepte 51
4.1 Verwendung von Programmiersprachen 51
4.1.1 Annotationen 52
4.1.2 Programmfragmente 52
4.1.3 Konsequenzen 53
4.2 Integration mit ASN.1 55
4.2.1 ASN.1 im Überblick 55
4.2.2 Standardisierte Kombination mit ASN.1 59
4.2.3 ASN.1-Einbettung mit SITE 61
4.2.4 ASN.1-*macro*-Anwendungen 64
ERROR 64
OPERATION 65
4.2.5 Codierung von ASN.1-Daten 67
4.2.6 ASN.1-ANY 68
4.2.7 Realisierung von Datentyperweiterungen (*extensibility*) 69
4.2.8 Zusammenfassung 70
4.3 Kombination mit IDL/ODL 71
4.3.1 ODL im Überblick 72
4.3.2 ODL-nach-SDL-Abbildung 73

Kapitel 5
Datenbasierte Kommunikation 75
5.1 Externe Kommunikation allgemein 75
5.1.1 Signalbasierte Kommunikation 76
Externe Signalformate 77
Verbindungen 77
5.1.2 Spracherweiterungen 78
5.2 Externe Kommunikation mit ASN.1 79
5.2.1 Kommunikation im Projekt PLATINUM 79
5.2.2 Kommunikation im Projekt VESUV 81
Datenbankzugriffe 81
Signalbasierte Kommunikation 81
5.2.3 Basiskommunikation mit CORBA 81
5.3 Externe Kommunikation mit IDL/ODL 82
5.3.1 Standardisierte Transformation von ODL nach SDL 83
5.4 Zusammenfassung 83

Kapitel 6
Neue objektorientierte Datenkonzepte 85
6.1 Neue Begriffe im Kontext von SDL-Daten 85

6.1.1 Ausnahmebehandlung (SDL-2000) ... 90
Vordefinierte Ausnahmen ... 90
Ausnahmen kombiniert mit ACT ONE ... 91
6.2 Transformationsbasierte Datenerweiterungen ... 91
6.2.1 Prozesse für Referenzen auf Werte ... 92
6.2.2 Sammelobjektverwaltung ... 94
Spezifikation der Transformationsziele für *Base* und *Sort* ... *96*
Konstruktion ... 100
Zugriff auf Werte von Referenzen ... 101
Vererbung ... 101
Zuweisungen ... 101
Methoden ... 101
Speicherverwaltung ... 103
6.2.3 Varianten der Sammelobjektverwaltung ... 103
Verteilung der Speicherprozesse ... 103
Konstruktoren ... 103
Veränderung der Zuweisungsregeln ... 103
Globale Variablen ... 104
Referenzen auf Referenzen ... 104
Speicherverwaltung ... 105
Alternative Datentypkonstruktionen ... 106
6.2.4 Einzelobjektverwaltung ... 106
Spezifikation der Transformationsziele für *Base* und *Sort* ... *107*
Konstruktion ... 110
Zugriff auf Werte von Referenzen ... 110
Vererbung ... 110
Zuweisungen ... 111
Methoden ... 111
Speicherverwaltung ... 112
6.2.5 Varianten der Einzelobjektverwaltung ... 112
Konstruktoren ... 112
Wurzel aller Objekttypen ... 112
Speicherverwaltung ... 113
Veränderte Zuweisungen ... 113
Sichtbarkeitsregeln ... 116
Alternative Datentypkonstruktionen ... 116
6.2.6 Variationen beider Grundmodelle ... 117
Überladene Methoden ... 117
Mehrfachvererbung ... 118
Zugriffssynchronisation ... 118
6.2.7 Fazit ... 119
6.3 Substitution des Datenkonzepts durch ITU-T ... 120
6.3.1 Grundlegende Eigenschaften ... 120
Definition of Datentypen ... 121
Definition von Kommunikationsschnittstellen ... 122
6.3.2 Vererbung ... 122
6.3.3 Zuweisungen mit Referenzen ... 123
6.3.4 Referenzkonstruktionen und Gültigkeit ... 123
6.3.5 Bindung von Methoden und Operatoren ... 124
6.3.6 Fazit zur ITU-T-Lösung ... 124
6.4 Einschätzung der neuen Datenkonzepte ... 125

Fazit und Ausblick ... 129

Anhänge

A Abkürzungen 131
B Literatur 133
C Index 138

Abbildungsverzeichnis

Bild 1: Aufbau des SDL-96-Sprachstandards 11

Bild 2: Entwicklungsumgebung SITE 14

Bild 3: SITE-C++-Bibliotheken 19

Bild 4: Beispiel einer SDL-Hierarchie 28

Bild 5: Lineare Abbildung bei Sortenvererbung 29

Bild 6: Beispiel *Enum* und *Enum2* *30*

Bild 7: Veränderung der Datenkonzepte in SDL 42

Bild 8: Integration externer Beschreibungen in SDL 43

Bild 9: Definition der externen Sorte *int* *44*

Bild 10: Notwendige Änderungen am Sprachstandard für ein neues Datenkonzept 48

Bild 11: Beispiel einer Spezifikation mit „*SDL in Combination with ASN.1*“ 59

Bild 12: Beispiel für die ASN.1-Integration mit SITE 62

Bild 13: ASN.1-*macro*-Definition *ERROR* *64*

Bild 14: ASN.1-*macro*-Definition *OPERATION* *66*

Bild 15: Schema zur Abbildung von vordefinierten SDL-Sorten auf ASN.1 67

Bild 16: Konfiguration von Verbindungen basierend auf Kanalnamen 78

Bild 17: Adressierung in PLATINUM 80

Bild 18: Beispiel einer Spezifikation mit Ausnahmen 91

Bild 19: Referenzen mit Prozeßvariablen 93

Bild 20: Referenzen im Speicherprozeß 94

Bild 21: Prozedur *print* *102*

Bild 22: Referenzen mit Referenzobjekten 107

Bild 23: Polymorphe Zuweisungen für Objekt- und Wertetypen 113

Bild 24: Zuweisungen für Objekt- und Wertetypen mit Projektion 115

Bild 25: SDL-2000-konforme Zuweisungen für Objekt- und Wertetypen 116

Einleitung

Die Sprache SDL ist im Laufe ihrer Entwicklung vor allem wegen ihrer weltweiten industriellen Akzeptanz eine der bedeutendsten Spezifikationstechniken von Telekommunikationssystemen und insbesondere von standardisierten Telekommunikationsprotokollen geworden.

Die Stärken der Sprache liegen zum einen in der intuitiv verständlichen Semantik sämtlicher Kernkonzepte und der grafischen Repräsentation mit ausgezeichneter Werkzeugunterstützung. Zum anderen ist die Sprache sehr lebendig. Von der ITU-T standardisiert und innerhalb der ITU-T als Spezifikationstechnik empfohlen, wird die Sprache ständig weiterentwickelt, wobei die Rückwärtskompatibilität gewährleistet wird. Auf die Entwicklung der Sprache haben neben den Nutzern der Sprache (Telekommunikationsorganisationen der verschiedenen Länder, Herstellungsfirmen von Telekommunikationskomponenten) auch SDL-Werkzeughersteller Einfluß genommen. Mit dem Mandat der Deutschen Telekom AG ist auch der Lehrstuhl Systemanalyse von Prof. Fischer der Humboldt-Universität zu Berlin, an dem die vorliegende Arbeit entstand, seit etwa 1991 aktiv an der SDL-Entwicklung beteiligt.

Einen besonderen Aspekt in der Sprachentwicklung stellt das Datenkonzept dar. Bei aktuellen Protokollen aus der Schmalband- und Breitband-ISDN-Familie oder dem IN-Bereich hat die Komplexität der Datenbeschreibungen gegenüber der Beschreibung des Protokollverhaltens dramatisch zugenommen. Die vorliegende Arbeit ist dem Datenkonzept von SDL gewidmet, welches von SDL-Anwendern wegen zu abstrakter, in ihren Verwendungsmöglichkeiten veralteter Sprachkonzepte und von Werkzeugherstellern aus Implementierungsgründen kritisiert wird. Nach einer Bestandsaufnahme, die diese Mängel identifiziert, wird eine Systematik zur Verbesserung des Datenkonzepts erarbeitet. Anhand dieser Systematik kann der Nutzen verschiedenster Modifikationen am SDL-Datenkonzept eingeschätzt werden.

Mit Hilfe der am Lehrstuhl implementierten SDL-Werkzeugkette SITE wurden insbesondere vom Autor neue Datenkonzepte entwickelt und getestet. Neben der Integration von Programmiersprachen in SDL ist auch die Integration alternativer Beschreibungstechniken wie ASN.1 und IDL/ODL besonders interessant.

ASN.1 wurde im OSI-Kontext zur abstrakten Beschreibung von Daten zusammen mit einer Codierungsvorschrift für eine byteweise Übertragung von Werten standardisiert. Auf die Kombination von SDL mit ASN.1, eine der Hauptarbeitsgebiete des Autors, wurde seit dem Beginn der SITE-Entwicklung ein besonderes Gewicht gelegt. In zahlreichen Projekten mit Industriepartnern wie der Deutschen Telekom AG und der Siemens AG wurde diese Kombination ständig weiterentwickelt. Viele Erfahrungen zur ASN.1-Problematik gingen durch die Expertentätigkeit bei der ITU-T auch in die standardisierte Kombination ein. IDL wird in CORBA-Architekturen für die Beschreibung von Schnittstellen eingesetzt. Die CORBA-Architektur definiert neben der Codierung der zu übertragenden Werte auch ein Protokoll zur Ansteuerung der Schnittstellen. Am Lehrstuhl wird diese Entwicklung unter dem besonderen Blickwinkel der Telekommunikation aktiv verfolgt. Insbesondere der ITU-T-Standard für ODL, eine telekommunikationsspezifische Erweiterung von IDL, entstand unter Leitung von Prof. Fischer.

Beide Beschreibungstechniken, ASN.1 und IDL, unterstützen also nicht nur die Beschreibung von Daten, sondern auch Kommunikationsaspekte. Deshalb werden in der Arbeit ebenfalls Möglichkeiten der Kommunikation von SDL mit externen Programmkomponenten als Berührungspunkt zum Datenkonzept untersucht.

Einen weiteren Schwerpunkt für Veränderungen am SDL-Datenkonzept bildet die verbesserte Nutzung objektorientierter Daten. Vom Autor wurden dazu einige Aspekte mit den am Lehrstuhl verfügbaren SDL-Werkzeugen prototypisch implementiert, um zumindest die potentielle Funktionsfähigkeit zu demonstrieren. Die Auseinandersetzung mit dieser Thematik wird auch im Zusammenhang mit der jüngsten Sprachversion SDL-2000 geführt, für die bisher keine Werkzeugunterstützung verfügbar ist.

Die Entwicklung eines Verständnisses für realistische Erwartungen an SDL-Werkzeugen, bezüglich der Datenimplementierung, ist ein wesentliches Ziel dieser Arbeit. Insbesondere wird Anwendern durch die erarbeitete Systematik ein Instrumentarium in die Hand gegeben, Stärken und Schwächen der eingesetzten SDL-Werkzeuge zu beurteilen.

Technische Bemerkungen

Im Kontext von SDL werden eine Vielzahl englischer Begriffe auch ohne deutsche Übersetzung verwendet. Falls es zu einem englischen Begriff keine gut verständliche Übersetzung in die deutsche Sprache gibt, wird eine kursive Notation (z.B. *decision*-Konstruktion) verwendet. Ebenfalls kursiv werden im laufenden Text Bezeichner von Programmbeispielen abgehoben (z.B. die Variable *var*). Eine Ausnahme bilden im laufenden Text verwendete Schlüsselwörter, die fett gesetzt sind (z.B. der C-Datentyp **int**).

Querverweise zwischen den Abschnitten der vorliegenden Arbeit enthalten stets die zur Orientierung hilfreiche Abschnittsnummer, für den inhaltlichen Bezug die kursiv ausgeführte Überschrift des Abschnitts und zum schnellen Nachlesen auch die Seitenangabe in Klammern (z.B. Kapitel 1 „*Die Sprache SDL*" (S.5)).

Sprachpräfixe (z.B. SDL-Modulkonzept) werden verwendet, sobald der Bezug auch andere Interpretationen zuläßt. Für Abkürzungen sei auf den Anhang A „*Abkürzungen*" (S.131) verwiesen. Zur Unterstützung der Lesbarkeit sind alle wichtigen Begriffe bei ihrer Einführung im laufenden Text unterstrichen und zusätzlich im Anhang C „*Index*" (S.138) zusammengefaßt.

Diese Arbeit nutzt die Abkürzung SDL für die Sprachversion aus dem Jahr 1996. Eine Erläuterung zu den Versionen befindet sich im Abschnitt 1.1 „*Historische Entwicklung von SDL*" (S.5).

Inhaltlicher Aufbau der Arbeit

In dieser Arbeit werden viele Begriffe und Konzepte aus der SDL-Welt und der objektorientierten Methodologie verwendet. SDL wird deshalb in gestraffter und in einer für diese Arbeit notwendigen Form im Kapitel 1 „*Die Sprache SDL*" (S.5) erklärt. Für syntaktische Einzelheiten der Sprache muß jedoch auf die Standard-SDL-Literatur, z.B. [BHS91], [EHS97], [OFM+94], [Z.100] verwiesen werden. Objektorientierung ist ein weitläufig verwendeter Begriff. In der vorliegenden Arbeit wird Objektorientierung als eine Zusammenfassung oder Anwendung der Sprachkonzepte Vererbung, Polymorphie, Virtualität, Parametrisierung und Datenkapselung genutzt. SDL realisiert viele dieser Einzelkonzepte mit einer wohldefinierten Semantik. Deshalb ist SDL eine objektorientierte Spezifikationstechnik. Bei der SDL-Einführung wird darauf hingewiesen, welches objektorientierte Paradigma auf das jeweils vorgestellte Sprachkonzept anwendbar ist. Die Definition der objektorientierten Konzepte orientiert sich in dieser Arbeit also an SDL, auch wenn es in der Literatur andere mögliche Interpretationen gibt.

Kapitel 2 setzt sich kritisch mit dem vorhandenen SDL-Datenkonzept auseinander und liefert den für diese Arbeit zentralen und von anderen Kapiteln immer wieder aufgegriffenen Anforderungskatalog für die Modifikation des SDL-Datenkonzepts. Welche Möglichkeiten sich überhaupt bieten, Daten mit SDL ausdrucksstärker beschreiben zu können, wird im Kapitel 3 systematisch untersucht. Vokabular und auch dargestellte Konsequenzen der verschiedenen Ansätze sind für folgende Kapitel essentiell.

In SDL-Werkzeugen wurden auf Drängen der Nutzer bereits viele Veränderungen in Bezug auf das Datenkonzept realisiert, begonnen mit der Einbettung von Programmiersprachen, bis hin zur Verwendung abstrakterer Beschreibungstechniken wie ASN.1 oder IDL. Mit dem Hintergrundwissen der vorangegangenen Kapitel läßt sich der Wert dieser, im Kapitel 4 vorgestellten Verbesserungen abschätzen.

Im Kapitel 5 wird ein ganz besonderer Aspekt des Datenkonzepts beleuchtet: das Zusammenspiel mit dem SDL-Basiskonzept Kommunikation. Obwohl Kommunikation nur ein Randaspekt aus der Sicht des Datenmodells ist, wird eine Diskussion durch die Kombination von SDL mit externen Sprachen, die insbesondere zur Beschreibung von Kommunikationselementen entworfen wurden, wichtig zur Einschätzung des Nutzens der entsprechenden Kombination. ASN.1 und IDL, eingeführt im Kapitel 4, spielen hier eine wesentliche Rolle.

Kapitel 6 befaßt sich mit Veränderungen am Datenkonzept unter dem besonderen Blickwinkel der Objektorientierung. Es werden unter Ausnutzung unterschiedlicher Modellansätze für ein Referenzkonzept in SDL Spracherweiterungen erarbeitet, die zusammen mit der ITU-T-Entwicklung für SDL-2000 bewertet werden. Im Gegensatz zum Kapitel 4 sind diese Erweiterungen in der SDL-Praxis mangels breiter Werkzeugunterstützung noch nicht etabliert.

Die werkzeugorientierten Kapitel 4 und 5 sind unabhängig vom strategisch orientierten Kapitel 6. Demnach könnte den Basiskapiteln 1 bis 3 das Kapitel 6 unmittelbar folgen.

Abgrenzung

Die vorliegende Arbeit entstand als wissenschaftliche Komponente einer mehr als siebenjährigen Projektarbeit u.a. mit der Deutschen Telekom AG und der Siemens AG. Deshalb stehen Implementierbarkeit und praktischer Nutzen von Konzepten immer im Vordergrund der Betrachtungen. Obwohl viele Details in den Kapiteln aus praktischen Erfahrungen heraus entstanden sind und diese auch als Basis eines Softwareentwurfs genutzt werden können, versteht sich diese Arbeit nicht als technische Anleitung. Der Schwerpunkt liegt vielmehr auf der systematischen Untersuchung verschiedener Möglichkeiten mit entsprechenden Hinweisen für vorhandene, insbesondere eigene Implementierungen.

Für die im Jahr 2000 verabschiedete Sprachversion gibt es bislang und auch in näherer Zukunft weder Werkzeuge noch relevante Anwendungen für die neuen Datenkonzepte. Deshalb ist die SDL-Grundlage dieser Arbeit die Sprachversion aus dem Jahr 1996. SDL-2000 ist lediglich eine der untersuchten Verbesserungsvarianten von SDL-96.

Danksagung

Im Rahmen meiner Tätigkeit in verschiedenen Drittmittelprojekten am Lehrstuhl Systemanalyse im Institut für Informatik der Humboldt-Universität zu Berlin beschäftigte ich mich immer wieder mit der Datenthematik als ein zu lösendes Software- und SDL-Sprachproblem im Umfeld der SITE-Werkzeuge. Prof. Fischer, als Lehrstuhlinhaber, überzeugte mich, die Auseinandersetzung mit dieser Problematik auch wissenschaftlich zu führen. Viele Kollegen und Studenten dieses Lehrstuhls waren bzw. sind an der SITE-Entwicklung beteiligt und haben mich durch ihre eigenen Arbeiten bei der Bewältigung technischer Details unterstützt. Ein ganz besonderer Dank geht dabei an Prof. Fischer, für die zahlreichen kritischen Hinweise zur schriftlichen Fassung der Arbeit.

Kapitel 1

Die Sprache SDL

Dieses Kapitel gibt einen allgemeinen Überblick der Sprache SDL. Innerhalb eines langen historischen Entwicklungswegs gewachsen, besitzt SDL heute eine sehr komplexe, standardisierte Sprachdefinition. Deshalb werden nach einem historischen Abriß der Sprache, die allgemeinen SDL-Konzepte und der komplexe Aufbau der Sprachdefinition aus dem Jahr 1996 erläutert.

Zum Ende des Jahres 2000 wurde von der ITU-T eine weitere Sprachrevision verabschiedet, zu der es bisher allerdings keine Werkzeugunterstützung zur technischen Absicherung aller neuen Konzepte gibt. Deshalb werden die Neuerungen in diesem Kapitel ebenfalls, aber getrennt von SDL-96 dargestellt.

Schließlich werden auch die SDL-Entwicklungswerkzeuge vorgestellt, die unter der aktiven Mitwirkung des Autors entstanden sind. Mit diesen, unter dem Namen SITE bekannten Werkzeugen, steht eine leistungsfähige Technologie zur Verfügung, mit der es unter anderem möglich ist, sowohl vorhandene als auch neue SDL-Sprachkonzepte zu testen. Aus dem Einsatz von SITE in industriellen Projekten entstanden viele der in dieser Arbeit vorgestellten Lösungen.

1.1 Historische Entwicklung von SDL

Die Sprache SDL wurde in ihrem Kern (grafische Notation für Zustandsdiagramme) 1976 erstmalig standardisiert (CCITT- Orange Book) und in regelmäßigen Abständen den aktuellen Anforderungen der Nutzer angepaßt.

Während der Phase von 1976 bis 1984 wurden die grafischen Elemente verbessert und ein dynamisches Prozeß- und Signalkonzept hinzugefügt. Der Sprachstandard von 1984 enthielt auch erstmalig Richtlinien, wie SDL zu nutzen ist.

Ab 1984 wurde der Aspekt der Softwareentwicklung mit SDL berücksichtigt. Die 1988 verabschiedete Sprachversion SDL-88 war dabei ein wichtiger Meilenstein. Folgende Grundsteine wurden u.a. mit diesem Sprachstandard gelegt:

- Alle telekommunikationsspezifischen, grafischen Elemente der Sprache (z.B. Telefonhörer) wurden durch allgemeine Sprachkonzepte ersetzt. Damit konnte SDL in vielen Bereichen zur Beschreibung von Abläufen verteilter Systeme eingesetzt werden.
- Es wurde eine textuelle Notation eingeführt. Zur sprachlichen Unterscheidung der SDL-Varianten wurden die Namen SDL/PR für textuelle und SDL/GR für grafische Notationen vergeben. Textuelle Notationen sind für Werkzeuge wesentlich einfacher als grafische Notationen zu verarbeiten, so daß die Entwicklung von Werkzeugen forciert werden konnte.
- Ein Datenkonzept wurde in die Sprache integriert. Damit wurde ein Detaillierungsgrad von Spezifikationen möglich, der eine automatische und auch vollständige Programmgenerierung zuließ.
- Die Semantik der Sprache wurde normativ unter Verwendung formaler Mechanismen erklärt.

Die wachsende Bedeutung objektorientierter Entwurfs- und Programmiertechniken, mit dem Hintergrund der Wiederverwendbarkeit von Softwarekomponenten, führte zu einer weiteren Überarbeitung des SDL-Sprachstandards. Motor dieser Entwicklung waren Arbeiten im Rahmen eines skandinavischen Forschungsprojekts, in denen ein objektorientiertes SDL [HaPe91]

vorgeschlagen wurde. Auf dieser Grundlage wurde 1992 die im wesentlichen noch heute gültige Sprachdefinition für SDL-92 [Z.100] von der ITU-T verabschiedet. Eine breite Anwendung der objektorientierten Konstruktionen war erst Jahre später zu beobachten. Die Ursachen sind zum einen die sehr späte Veröffentlichung des Sprachstandards Anfang 1994 durch die ITU-T und zum anderen die ungenügende Unterstützung aller neuen Konzepte durch entsprechende Software.

Bei der Beschreibung neuerer Systeme erkannte man, daß das vorhandene Datenkonzept den wachsenden Anforderungen nicht gerecht werden konnte. Etwa 1995 wurde auf Empfehlung von ETSI [ETSI93] und dem Drängen der von der ITU-T repräsentierten SDL-Nutzerorganisationen eine standardisierte Kombination von SDL und ASN.1 [Z.105] als eine alternative Sprachvariante mit dem Namen *„SDL in Combination with ASN.1"* entwickelt. ASN.1 [X.208] wurde als Datenbeschreibungssprache für das OSI-Referenzmodell [X.200] entwickelt. Eine häufige Anwendung der Sprache ist die standardisierte Beschreibung auszutauschender Daten in Kommunikationsprotokollen unterschiedlichster Art.

Verschiedene Korrekturen und kleine Verbesserungen an der SDL-92-Sprachdefinition wurden 1996 von der ITU-T in Form eines Anhangs zum Standard Z.100 [Z100A1] veröffentlicht. Die Sprache mit diesen Änderungen wird als SDL-96 bezeichnet. Zur gleichen Zeit erfolgte die Definition des Austauschformats CIF [Z.106], welches die Nutzung grafischer SDL-Elemente aus SDL/GR-Editoren verschiedener Hersteller ohne wesentliche Veränderung des Layouts ermöglichte.

Zum Ende des Jahres 2000 wurde eine weitere Sprachversion verabschiedet, die zu diesem Zeitpunkt sowohl unvollständig als auch instabil war. Bis heute ist diese Sprachdefinition noch in Bewegung, da massiv auftretende Fehler in den Sprachkonzepten zu bereinigen sind. Neben einem neuen Datenmodell enthält dieser Standard viele Sprachverbesserungen und eine neue formale Basis für die Semantikdefinition.

Die Bezeichnung SDL bezieht sich immer auf die letzte offizielle Sprachversion, d.h. eigentlich auf SDL-2000. Da SDL-2000 in der Praxis noch nicht etabliert ist, wird in dieser Arbeit die Abkürzung SDL als Synonym für die Sprache SDL-96 verwendet.

1.2 Wichtige SDL-Sprachkonzepte

Den Kern der Sprache SDL bilden erweiterte[1], endliche Automaten, die per Signalaustausch asynchron untereinander kommunizieren. Eine grafische Syntax und objektorientierte Strukturierungskonzepte sorgen für ein modernes Erscheinungsbild der Sprache. Wichtige Sprachkonzepte werden im folgenden kurz vorgestellt.

Grundzustand: Ein erweiterter, endlicher Automat besteht in SDL aus genau einem initialen Zustand *start*, dem besonderen Zustand *stop* und weiteren namentlich benennbaren Zuständen. Erreicht der Automat den Zustand *stop*, so endet seine Existenz. Um die Verwechslung mit anderen Zustandsbegriffen zu vermeiden, werden diese Automatenzustände als Grundzustände bezeichnet.

Zustandsübergang: Durch die Stimulierung eines Automaten mit Signalen kann sich sein Grundzustand ändern. Zwischen dem Lesen des Signals aus dem Empfangspuffer (*input*-Konstruktion) und dem Wechsel des Grundzustands (*terminator*-Konstruktion) können Aktionen ausgeführt werden. Die gesamte Abfolge wird als Zustandsübergang (*transition*-Konstruktion) bezeichnet.

Aktion: Aktionen sind Verhaltenselemente, z.B. Zuweisungen von Variablen, die nacheinander ausgeführt werden.

---

1. Die Erweiterungen erlauben lokale Variablen, Zeitgeber und Empfangspuffer für Signale.

Aktionsverzweigung: Mit einer *decision*-Konstruktion kann abhängig von einem Datenwert die Ausführung von Aktionen verzweigt werden. Für jeden, bei der Ausführung möglichen Wert muß eine Verzweigung existieren, sonst ist das weitere Verhalten der SDL-Spezifikation unbestimmt. Die Bindung einer Verzweigung kann an ganze Wertebereiche und alle nicht explizit angegebenen Werte (*else*-Zweig) erfolgen. Eine spezielle Variante der Aktionsverzweigung gestattet auch eine nichtdeterministische Ausführung von Aktionen.

Empfangspuffer: Jeder Zustandsautomat besitzt einen Empfangspuffer für Signale. Mit Ausnahme priorisiert zu behandelnder Signale erfolgt die Stimulierung des Zustandsautomaten in der Empfangsreihenfolge der Signale. Gelesene Signale werden aus dem Empfangspuffer entfernt. Ein empfangenes aber noch nicht erwartetes Signal kann zurückgestellt werden (*save*-Konstruktion), d.h. es wird im aktuellen Grundzustand so behandelt, als wäre es noch nicht empfangen worden. Der Automat verharrt im Zustand, wenn sich nach seinem letzten Zustandsübergang kein Signal oder nur zurückgestellte Signale im Empfangspuffer befinden.

Verhaltensgraph: Ein Verhaltensgraph beschreibt das potentielle Verhalten eines Automaten (oder einer Menge von Automaten). Die Knoten des zusammenhängenden Graphen bilden die Grundzustände, die Kanten die Zustandsübergänge. Der Grundzustand *start* kann als globales Minimum für die durch den zeitlichen Ablauf gerichteten Zustandsübergänge angesehen werden. Durch Strukturierungskonzepte (Service, Prozedur) kann ein Zustandsautomat in mehrere Teilgraphen aufgeteilt werden.

Prozeß: Ein Zustandsautomat wird dynamisch mit genau einer Prozeßinstanz repräsentiert. Die Beschreibung erfolgt als Prozeßinstanzmenge, die neben dem Verhaltensgraphen auch die Beschreibung der Kommunikationsschnittstelle und andere Definitionen enthält. Mit der Erzeugung einer Prozeßinstanz als Aktion in einer anderen Elternprozeßinstanz oder initial beim Start des SDL-Systems beginnt das dynamische Verhalten des Zustandsautomaten (Ausführung eines Prozesses) mit dem initialen Zustandsübergang ohne Stimulierung durch ein Signal.

Der Begriff Prozeß wird verwendet, wenn aus dem Nutzungskontext die Interpretation als Prozeßinstanz, Prozeßinstanzmenge oder Prozeßdefinition eindeutig ist.

Signal: Die Übertragung von Informationen zwischen Prozeßinstanzen erfolgt mit verschiedenen Signalinstanzen, die auf der Grundlage von Signalbeschreibungen gebildet werden, ausschließlich über Kommunikationspfade. Eine Signalbeschreibung legt den Signalbezeichner und Datenparameter, die mit übertragen werden können, fest.

Der Begriff Signal wird verwendet, wenn die Unterscheidung zwischen Signalinstanz und Signalbeschreibung durch den verwendeten Kontext möglich ist.

Gesendete Signale werden nach der Übertragung im Empfangspuffer des flexibel adressierbaren Prozesses eingetragen. Die Übertragung beginnt mit der Aktion Senden (*output*-Konstruktion) und ist unabhängig vom weiteren Zustandsverhalten des Senders (asynchroner Signaltransport). Existiert kein Empfänger oder ist er wegen fehlender Kommunikationspfade nicht erreichbar, wird ein gesendetes Signal verworfen.

Prozedur: Ein Teil eines Verhaltensgraphen, dessen dynamischer Verlauf bei genau einer Aktion beginnt und alle möglichen Fortsetzungen auch in genau einer Aktion enden, kann in eine Prozedur ausgelagert und durch einen Prozedurruf ersetzt werden. Prozedurrufe sind also Aktionen innerhalb eines Zustandsüberganges, die eine Strukturierung in weitere Zustandsübergänge erlauben. Verschachtelte und insbesondere rekursive Prozedurrufe sind möglich.

Als Strukturierungskonzept für den Zustandsautomaten werden Prozeduren im Kontext der rufenden Prozeßinstanz ausgeführt, egal wo sie in der SDL-Beschreibung definiert sind. Prozeduren besitzen keinen eigenen Empfangspuffer. Es ist möglich, daß Prozeduren selbst keinen Zustandsübergang sondern nur Aktionen enthalten. Derartige Prozeduren werden oft zur Beschreibung von Datenmanipulationen verwendet.

entfernte Prozedur: Das Senden eines Signals an einen anderen Prozeß und das ausschließliche Warten auf ein Antwortsignal ist ein typisches Verhaltensmuster für kommunizierende Prozesse. In SDL gibt es dafür ein syntaktisches Konzept: entfernte Prozeduren.

Ein *server*-Prozeß gibt bekannt, daß er auf Anforderung eine Prozedur aufruft und die Beendigung der Prozedur mit allen Ergebnissen dem rufenden *client*-Prozeß zurückmeldet. Der *client*-Prozeß stellt alle Signale, selbst Rückfragen des *server*-Prozesses solange zurück, bis die erwartete Antwort eintrifft. Wenn diese Antwort nicht eintreffen kann, weil der Server auf die Beantwortung einer Rückfrage wartet oder der Ruf einer entfernten Prozedur an einem nicht existierenden *server*-Prozeß erfolgte, dann ist der *client*-Prozeß für immer blockiert (*dead lock*). Entfernte Prozeduren sind also ein nicht unterbrechbares, synchrones Kommunikationskonzept.

Service: Der Zustandsautomat eines Prozesses kann alternativ auf mehrere innere Servicedefinitionen aufgeteilt werden. Jeder Service enthält einen Teilgraphen. Die Zerlegung hat folgende Eigenschaften:

a. die Menge der Grundzustände der Teilautomaten sind paarweise disjunkt,
b. Zustandsübergänge wechseln nur zu Grundzuständen, die lokal zur Zerlegung definiert sind,
c. die Mengen der Signalbezeichner für die Stimulierung der Teilautomaten sind paarweise disjunkt,
d. jede Zerlegung besitzt genau einen *start*-Zustand.

Die Ressourcen des Prozesses (Empfangspuffer, Variablen) werden gemeinsam genutzt. Da der Automat lediglich zerlegt wird, erfolgen die Zustandsübergänge auch verschiedener Services stets nacheinander.

Block: Prozeßinstanzmengen können in Blöcken zusammengefaßt werden. Auch Blöcke selbst können Baumstrukturen bilden, wobei gilt, daß ein Block entweder aus Prozessen oder weiteren Blockstrukturen besteht.

System: Das System ist die Wurzel der Blockhierarchie. Eine SDL-Spezifikation kann höchstens ein System enthalten.

Kommunikationspfad: Kommunikationspfade sind Sprachelemente (Kanäle, Signalrouten, Gates), die Services, Prozesse und Blöcke miteinander bzw. mit der SDL-Systemumgebung verbinden und dabei die transportierbaren Signalmengen, die Richtung des Signaltransports und Verzögerungen (nur für Kanäle) beschreiben.

Bezeichner: In SDL-Beschreibungen sind für Definition (z.B. System, Block, Signal, ...) i. Allg.[1] Namen zu vergeben. Mit einem Bezeichner wird eine Definition referenziert. Dieser Bezeichner enthält neben dem Namen der Definition in SDL zusätzlich eine Information über die Art der Definition und den vollständigen Pfad zur Definition. Verschiedene Arten von Definitionen können mit dem gleichen Namen versehen sein. Der Pfad zur Definition beginnt mit dem System oder einem Modul und folgt der hierarchischen Strukturierung.

Eine Definition ist nur sichtbar, wenn sie auf der gleichen Hierarchiestufe oder in einer höheren definiert ist bzw. von einem Modul importiert wird. Mit Ausnahme von Operatordefinitionen dürfen Definitionen nicht den gleichen Bezeichner besitzen.

Modul: Definitionen können in einer *package*-Konstruktion separat erfolgen, sofern sie nicht direkt eine Instanz repräsentieren[2]. Zusammen mit den entsprechenden Import- und Exportmechanismen wird eine derartige Konstruktion Modul genannt.

---

1. Bei *channel*- und *substructure*-Definitionen können Namen auch weggelassen werden.

2. Instanzen repräsentieren lokale Variablen, Systeme, Blöcke, Prozesse, Services und auch Module selbst.

Spezifikation: Eine Spezifikation oder SDL-Beschreibung setzt sich aus höchstens einem System und beliebig vielen Modulen zusammen. Nur eine Spezifikation, die auch ein System enthält, kann ausgeführt werden. Soll der Ausführungsaspekt betont werden, spricht man auch von einem SDL-Programm.

SDL-Systemumgebung: Jedes System ist in eine SDL-Systemumgebung eingebettet, die vom Sprachstandard nicht näher definiert ist. Ein System kann Signale an diese Umgebung senden und Signale von der Umgebung empfangen.

Betrachtet man eine konkrete Implementierung eines SDL-Programms, so entspricht die SDL-Systemumgebung einer Laufzeitumgebung.

Datentyp: Auf der Grundlage von Datentypen gibt es Werte und Operatoren. Der Begriff des Datentyps setzt sich aus mehreren Teilkonzepten zusammen, die im Kapitel 2 „*SDL-Datenbeschreibungen*" (S.21) genauer erläutert werden.

Variable: Variablen sind Speicher für Werte. Jede Instanz einer Definition mit lokalen Variablen hat einen eigenen Speicher und den exklusiven Zugriff für diese Variablen. Das gilt insbesondere für rekursive Prozedurrufe.

Typdefinition: Definitionen, für die ein Vererbungskonzept definiert ist, sind Typdefinitionen. Das sind in SDL neben den System-, Block-, Prozeß-, Service- und Datentypdefinitionen, die explizit mit dem Schlüsselwort **type** gekennzeichnet sind, auch Prozeduren und Signale.

typbasierte Definition: Aus System-, Block-, Prozeß- und Servicetypdefinitionen kann man typbasierte Definitionen bilden. Diese entsprechen konzeptionell den System-, Block-, Prozeß- und Servicedefinitionen.

Typ: Mit einer Typdefinition wird ein konkreter Typ beschrieben. Dazu gehört insbesondere der Name des Typs.

Der Begriff „Typ" wird als allgemeines Vererbungskonzept in dieser Arbeit nicht als Kürzel für „Datentyp" verwendet.

Vererbung: Vererbung von Typen ist in SDL mit Hilfe einer Transformation erklärt. Im wesentlichen werden bei der Vererbung die Inhalte einer vorhandenen Definition - dem Basistyp - in die Ableitung kopiert[1].

Virtualität: Blocktyp-, Prozeßtyp-, Servicetyp-, Prozedurdefinitionen und Zustandsübergänge können als virtuell gekennzeichnet sein, sofern sie sich direkt in einem Typ befinden. Eine virtuelle Struktur kann bei der Vererbung des definierenden Typs redefiniert werden. Diese Redefinition ist eine Ersetzung aller bereits erfolgten Verwendungen[2] der virtuellen Struktur.

In einer vollständigen Systembeschreibung ist die gesamte Vererbungsstruktur bekannt. Damit reduziert sich Virtualität auf ein Transformationsmodell ohne dynamische Verhaltensaspekte. Das Konzept der Virtualität in SDL ist insbesondere nicht auf Operatordefinitionen anwendbar.

abstrakter Typ: Die Definition eines Typs kann auf der Grundlage einer formalen Parametrisierung erfolgen. Ein formaler (Kontext-) Parameter ist eine unvollständige Definition einer SDL-Struktur, die im wesentlichen Bezeichner bereitstellt und optional Beschränkungen für die spätere Aktualisierung enthält. Bei einer Vererbung oder Instanzierung der parametrisierten Definition kann der formale Parameter durch einen passenden Definitionsbezeichner

---

1. Bei dieser Kopie werden auch Kontextparameter aktualisiert. Für Datentypen ist Vererbung im Abschnitt 2.2.2 „*Vererbung*" (S.29) genau erklärt.

2. Der Begriff „Instanzierung" anstelle von „Verwendung" ist nicht intuitiv. Das betrifft beispielsweise einen Prozedurruf. Allerdings schließt der Begriff Verwendung nicht die Ableitung eines neuen Typs aus der virtuellen Typkonstruktion ein.

ersetzt werden. Solange diese Parameter nicht ersetzt sind, kann eine derartige Typdefinition nicht zur Bildung einer typbasierten Definition, Signalinstanz oder Variablen bzw. zum Ruf einer Prozedur verwendet werden. In diesem Sinne sind parametrisierte Typen abstrakte Typen.

1.3 Definition der Sprache SDL im Sprachstandard

Allgemein wird eine Computersprache (kurz Sprache) sowohl durch ihre Syntax, als auch ihre statische und dynamische Semantik charakterisiert [ASU92].

Syntax: Unter einer Syntax versteht man die Definition einer kontextfreien Grammatik. Diese klärt, was ein gültiger Satz der Sprache ist. Oft wird noch zwischen lexikalischen und syntaktischen Regeln bei der Definition der Syntax unterschieden.

statische Semantik: Zwischen den syntaktischen Konstruktionen gibt es Abhängigkeiten, die nicht mit einer kontextfreien Grammatik beschreibbar sind. Die statische Semantik definiert diese zusätzlichen Einschränkungen für einen gültigen Satz der Sprache.

dynamische Semantik: Die dynamische Semantik definiert, wie die syntaktischen Konstruktionen zu interpretieren sind.

SDL ist eine formal definierte Sprache. Das bedeutet, daß die Charakterisierung der Sprache mit Hilfe einer formalen Notation erfolgt.

Der ITU-T-Sprachstandard von SDL ist ein Dokument mit mehreren normativen[1] und nicht normativen[2] Anhängen (vgl. Bild 1). Das zentrale Dokument [Z.100] enthält eine vollständige Sprachdefinition, wobei sowohl die statische als auch dynamische Semantik zunächst nur informal erklärt sind.

Syntax von SDL: Es gibt zwei konkrete Syntaxvarianten der Sprache SDL. Bekannt geworden ist SDL durch die grafische Notation SDL/GR. Grundlage vieler Werkzeuge für den Austausch und die Speicherung von SDL-Spezifikationen ist eine weitere Syntaxform, die textuelle Repräsentation SDL/PR. Beide Sprachvarianten sind durch formale Syntaxregeln im Hauptdokument Z.100 definiert.

statische Semantik von SDL: Neben der konkreten Syntax wird im Hauptdokument Z.100 eine abstrakte Syntax definiert. Diese abstrakte Syntax ist einerseits eine Verallgemeinerung der konkreten syntaktischen Repräsentation von SDL und andererseits Grundlage der formalen dynamischen Semantik.

Konkrete syntaktische Konstruktionen, die eine direkte Entsprechung in der abstrakten Syntax besitzen, sind Kernkonzepte von SDL. Alle Kernkonzepte besitzen einen Abschnitt „*Semantics*“ in der Sprachdefinition, wo das Verhalten der Konstruktion informal beschrieben ist. Andere konkrete Syntaxkonstruktionen werden durch geeignete Transformationen auf die Kernkonzepte von SDL abgebildet. Ein Abschnitt „*Model*“ in der Sprachdefinition beschreibt die entsprechende Transformation informal. Beispielsweise werden so entfernte Prozeduren auf ein Kommunikationsszenario mit Signalaustausch abgebildet.

Die Regeln zur Transformation der konkreten in die abstrakte Syntax sind Bestandteil der statischen, formalen Semantik von SDL. Die formale Beschreibung der Transformationsregeln, sowie aller statischen Tests erfolgt mit dem Kalkül MetaIV [BjJo82].

1. A bis D, E ist für zukünftige Entwicklungen reserviert, F1, F2, F3, Addendum 1
2. I und II

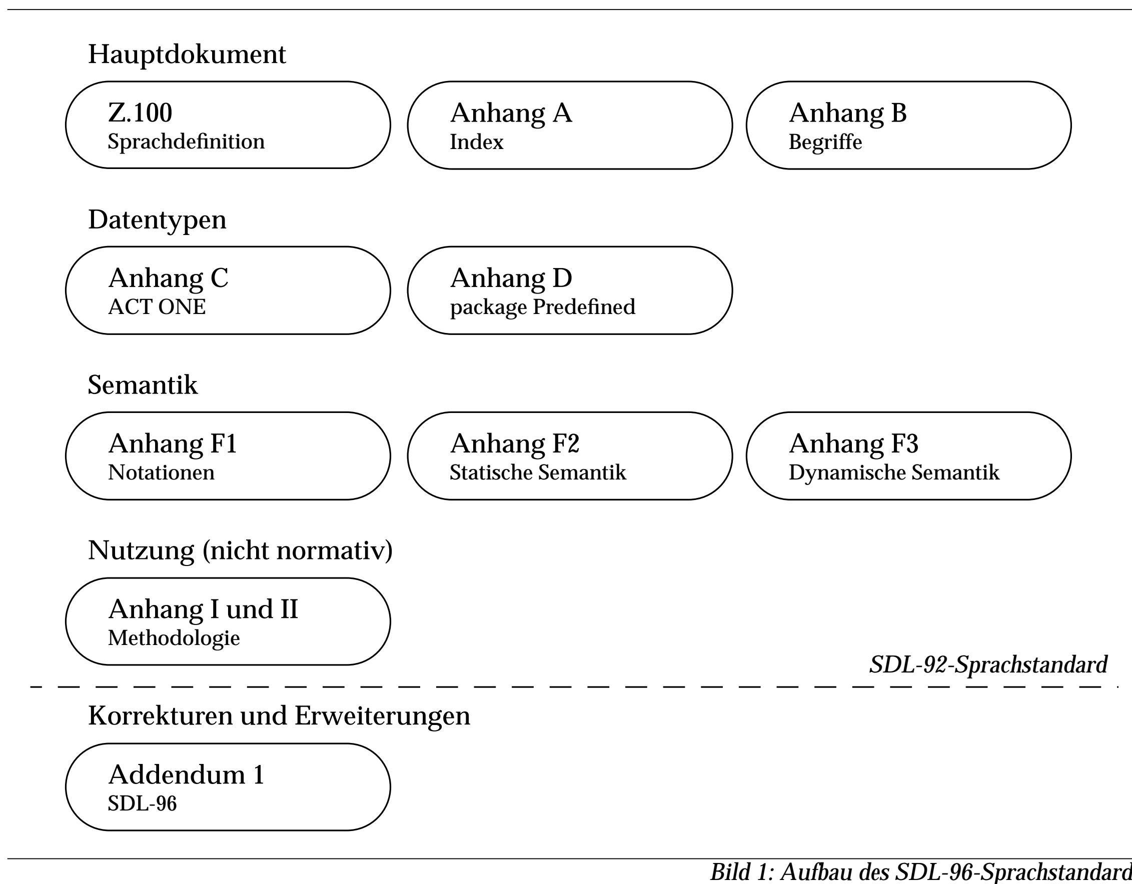

Bild 1: Aufbau des SDL-96-Sprachstandards

dynamische Semantik von SDL: Ausgangspunkt der formalen dynamischen Semantik ist ein SDL-Programm, das in der abstrakten Grammatik vorliegt. Mit einer Kombination aus MetaIV und CSP [Hoa85] wird eine Interpretation für die abstrakte Grammatik beschrieben. Die Interpretation ist die Basis für die prinzipielle Ausführung eines SDL-Programms.

Die formalen Definitionen sind in den normativen Anhängen F1 bis F3 [Z.100F] enthalten. Diese beinhalten im Einzelnen:

Anhang F1: allgemeine Erklärungen der im Sprachstandard verwendeten Notationen,

Anhang F2: die statische Semantik und

Anhang F3: die dynamische Semantik.

Auf der Grundlage von Syntax und Semantik gibt es in jeder Sprache ein Strukturierungskonzept. Betrachtet man SDL hinsichtlich der Eigenschaften, die sich aus der statischen und dynamischen Semantik ergeben, lassen sich weitere allgemeine Konzepte identifizieren:

- erweiterte endliche Zustandsautomaten als Prozeßkonzept,
- Kommunikation und
- Datenbeschreibungen.

Der Übergang zwischen diesen Konzepten ist fließend. Beispielsweise ist jede Datendefinition Teil der Strukturierung von SDL, Datenwerte werden im SDL-Prozeß erzeugt und bei der Kommunikation übertragen. Deshalb wird es auch bei der Betrachtung eines Konzepts immer Berührungspunkte mit anderen Konzepten geben.

Speziell das Datenkonzept von SDL besitzt eine besondere Eigenschaft: es ist konform zu einem algebraischen Datenmodell bekannt als ACT ONE [EhMa85]. Diese spezielle Form der Datennotation wird im Anhang C der SDL-Sprachdefinition [Z.100C], allerdings ohne Bezüge zur konkreten Syntax von SDL, umrissen. Es ist möglich, mit einer ACT ONE-basierten Datenbeschreibung alle primitiven Datentypen, z.B. ganze Zahlen oder Zeichenketten, mit der Sprache selbst zu definieren. Eine normative Sammlung der wichtigsten primitiven Datentypen bzw. auch wichtiger Konstruktionsvorschriften für komplexe Datentypen enthält der normative Anhang D der Sprachdefinition [Z.100D].

Zur Unterstützung der Lesbarkeit der Sprachdefinition gibt es weitere Anhänge.

Anhang A: Mit den sowohl konkreten als auch abstrakten Syntaxregeln werden Hilfssymbole (die jeweils linken Seiten einer Grammatikregel) definiert. Der normative Anhang A [Z100A] ist ein Index auf alle Definitionen und deren Verwendung im Hauptdokument Z.100.

Anhang B: Der Anhang B [Z100B] gibt einen alphabetisch geordneten Überblick aller wichtigen Begriffe der Sprache SDL.

Anhang I und II: Erläuterungen zur Anwendung von SDL und bibliographische Hinweise befinden sich in nicht normativen Anhängen der Sprachdefinition [Z100I], [Z100II].

Fehler und kleine Verbesserungen der 1992 verabschiedeten Sprachdefinition wurden von der ITU-T 1996 nur in einer normativen Änderungsliste [Z100A1] zusammengefaßt.

Der SDL-Sprachstandard hat eine Reihe von Schwächen:

- Eigenschaften, die sich aus dem formalen Modell ableiten lassen, entsprechen nicht immer der intuitiv erwarteten Semantik.
- Vereinzelt gibt es Widersprüche zwischen dem informalen Text und dem formalen Regelwerk, die der Komplexität der Sprache geschuldet sind. Entsprechend einer Erklärung im Anhang F1 sind diese Widersprüche zu Gunsten des informalen Texts zu lösen. Oft ist die Reihenfolge der Anwendung von Transformationsregeln für einzelne SDL-Konzepte das Problem, da das komplexe Zusammenspiel der Transformationsregeln nicht überschaubar ist. Beispielsweise ist Vererbung in SDL kein Kernkonzept und die Transformation im Zusammenspiel mit der Auflösung von Bezeichnern sehr kompliziert.
- Die ACT ONE-Konformität des Datenmodells ist zwar gefordert (Z.100 Abschnitt 5.2.1 „*Data type definitions*“) aber nicht formal belegt. Das wird insbesondere an den Berührungspunkten des Datenkonzepts mit anderen SDL-Konzepten deutlich.
- Die Pflege und Weiterentwicklung des SDL-Sprachstandards mit allen Anhängen ist extrem aufwendig. Insbesondere die Anhänge zur formalen Semantik sind nicht elektronisch verfügbar[1], welches Änderungen an der Semantik verhindert und letztendlich den praktischen Nutzen der formalen Semantik erheblich einschränkt.

1.4 SDL ab dem Jahr 2000

Die ITU-T veröffentlichte 2000 eine weitere SDL-Revision in Form einer kompletten Neuauflage des Sprachstandards. Diese Änderungen betreffen sowohl umfangreiche syntaktische Konzepte als auch formale Semantikaspekte.

Da sich die vorliegende Arbeit an einigen Stellen schon auf SDL-2000-Konzepte bezieht, werden die wichtigsten Konzepte auch kurz umrissen. Es ist offen, mit welchen Details und Präzisierungen die neuen Konzepte tatsächlich in SDL-2000-Werkzeuge integriert werden.

---

1. ITU-T akzeptiert nur spezifische elektronische Dokumentationsformate. Die formale Semantik wurde mit Werkzeugen erstellt, die nicht in das Dokumentationsschema der ITU-T paßten.

lexikalische Änderungen: In SDL-2000 unterscheidet zwischen Groß- und Kleinschreibung. Aus Kompatibilitätsgründen gibt es aber mehrere Varianten für die Schreibweise der Schlüsselwörter. Einige Regeln für lexikalische Einheiten wurden aus meist technischen Gründen geringfügig geändert.

Anpassung von SDL an UML: Eine der am häufigsten genutzten Notationstechnik für den abstrakten, objektorientierten Modellwurf von Softwaresystemen ist UML [OMG97]. Um den Übergang von einem UML-Entwurf zu einem semantisch wesentlich fixierterem SDL-System zu erleichtern, unterstützt SDL eingeschränkt die Symbolik von UML. Einige Konzepte von UML wurden auch direkt als neue SDL-Konzepte übernommen, z.B.

- verschachtelte Module und Zustände,
- informale Assoziationen.

interface-**Konzept:** Die Zerlegung einer Objektbeschreibung in Schnittstellen und Implementierungen ist ein praktikables Entwurfsprinzip. Ein Interface beschreibt eine adressierbare Kommunikationsschnittstelle, die beispielsweise von einem Prozeß implementiert werden kann.

Agent: Blöcke, Prozesse und Services können als Strukturen mit sehr ähnlichen Eigenschaften angesehen werden, insbesondere da Blöcke in SDL-2000 auch dynamisch erzeugt werden können. Deshalb ist es ein konsequenter Schritt, alle drei Konzepte einem abstrakten Strukturierungskonzept unterzuordnen. Diese Strukturen werden Agenten genannt.

algorithmische Notation: Die Beschreibung einfacher algorithmischer Abläufe in SDL ist im Vergleich zu anderen Sprachen aufwendig. Das liegt größtenteils an der grafischen Ausrichtung von SDL. Algorithmische Notationen sind eine Zusammenfassung neuer textueller Syntaxelemente, die genau diese Lücke der Sprachdefinition füllen.

Ausnahmebehandlung: Mit einem Konzept zur Ausnahmebehandlung können Situationen, die bisher zu einem undefinierten Verhalten des SDL-Programms führten (z.B. dynamische Fehler), durch ein geeignetes SDL-Codefragment abgefangen werden.

Ersetzung des Datenkonzepts: Das neue Datenkonzept wird Referenzen und Polymorphie unterstützen. Die semantische Basis des neuen Datenkonzepts gliedert sich direkt in die Definition der formalen Semantik von SDL ein. Algebraische Spezifikationen auf der Grundlage des ACT ONE-Modells sind nicht mehr möglich.

dynamische Semantik: Die Notation der dynamischen Semantik wird durch ASM (*Abstract State Machines*) ersetzt [GGP99]. Die intuitive Semantik der SDL-Kernkonzepte soll dabei unverändert bleiben. Die neue dynamische Semantik enthält wesentlich mehr Kernkonzepte als bisher, um die statische Semantikdefinition von den Transformationskonzepten zu entlasten.

statische Semantik: An einer Methodik zur Beschreibung der statischen Semantik wird derzeit innerhalb einer Expertengruppe der ITU-T gearbeitet. Fernziel ist eine elektronische Darstellung, die eine automatische Referenzimplementierung gestattet. Die auch am Lehrstuhl von Prof. Fischer entstandenen Arbeiten von Prinz [Pri01] und Piefel [Pie00] widmen sich dieser Thematik.

1.5 Die SDL-Entwicklungsumgebung SITE

Viele der in der Arbeit untersuchten Problemstellungen entstanden aus dem durchgängigen Einsatz des Autors bei der Implementierung einer SDL-Werkzeugumgebung mit dem Namen SITE (*SDL Integrated Tool Environment*) [Schr02a]. Die Werkzeugumgebung besteht im wesentlichen aus Compiler- und Erklärungskomponenten für SDL/PR. Einen Überblick über die gegenwärtig verfügbaren Komponenten gibt Bild 2. Für die Erzeugung einer grafischen SDL-

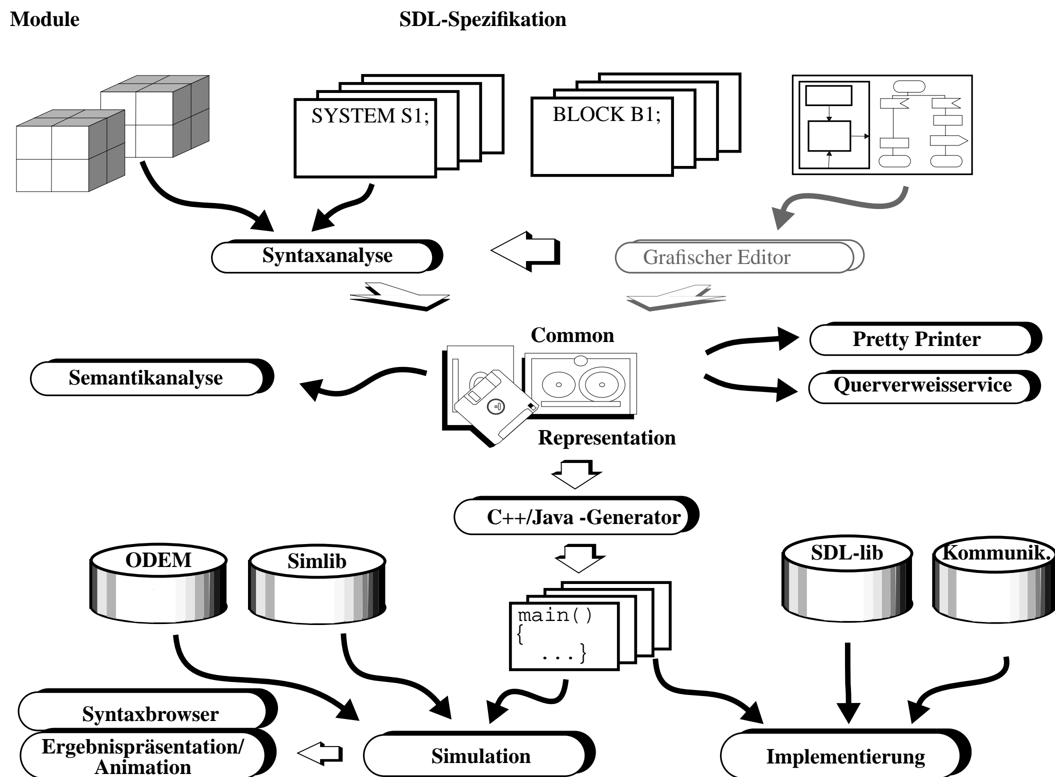

Bild 2: Entwicklungsumgebung SITE

Beschreibung wird auf kommerzielle SDL-Editoren mit SDL/PR-Schnittstelle zurückgegriffen.

Die Entwicklung der SITE-Werkzeuge begann 1993 im Rahmen des Teilprojekts YE2 des Sonderforschungsbereichs 342 der Deutschen Forschungsgemeinschaft [FiHo93]. Bereits zu diesem Zeitpunkt wurde die Realisierung aller SDL-92-Sprachkonstruktionen [AFW93] und die im Abschnitt 4.2.3 „*ASN.1-Einbettung mit SITE*" (S.61) diskutierte Kombination von SDL mit der Sprache ASN.1 [FiSc93] vorbereitet.

Ein zentrales Entwurfsprinzip von SITE ist der Austausch der Spezifikationen zwischen den eigenständig operierenden Komponenten. Mit diesem Ansatz können einerseits Komponenten zu Forschungszwecken prototypisch, insbesondere durch studentische Arbeiten, isoliert entwikkelt und andererseits stabile und in industriellen Produkten eingesetzte Werkzeuge gewartet werden.

Das Austauschformat und die wichtigsten SITE-Komponenten werden in den folgenden Abschnitten vorgestellt. Sie werden als Beispiele für mögliche Realisierungen an vielen Stellen der Arbeit herangezogen. Die Implementierung der Datenkonzepte in den SITE-Komponenten erfolgte größtenteils durch den Autor der vorliegenden Arbeit.

1.5.1 Austausch von SDL-Beschreibungen

Die Repräsentation einer SDL-Spezifikation im Hauptspeicher einer SITE-Komponente als abstrakter Grammatikbaum ist für alle Komponenten gleich. Das Metawerkzeug *Kimwitu* [EiBe96] liefert die technologische Grundlage zur Definition und komfortablen Bearbeitung eines Grammatikbaumes. In der SITE-Technologie wird die Definition der abstrakten Grammatik als *Common Representation* (CR) bezeichnet. Eine Instanz der CR im Hauptspeicher einer SITE-Komponente kann automatisch und eineindeutig in eine ASCII-Repräsentation transformiert werden. Das wird für die Zwischenspeicherung in Dateien ausgenutzt. Eine CR-Instanz in einem beliebigen Speichermedium wird als CR einer SDL-Spezifikation bezeichnet.

Die Definition eines Austauschformats für SDL erfolgte erstmalig von Schade [Scha92] in Anlehnung an die Definition einer abstrakten Grammatik im Sprachstandard. Wegen der ASN.1-Erweiterungen aber auch des Problems, auf die konkrete Syntax bezogene Fehlermeldungen zu erstellen, konnte die abstrakte Grammatik der SDL-Sprachdefinition nicht direkt verwendet werden. Die aktuelle CR-Version ist der konkreten Syntax von SDL strukturell sehr ähnlich und unterstützt:

- SDL entsprechend des aktuellen Standards von 1996,
- SDL in Kombination mit ASN.1 gemäß der Beschreibung im Abschnitt 4.2.3 „*ASN.1-Einbettung mit SITE*" (S.61),
- ITU-ODL-Strukturen für die in Abschnitt 4.3 „*Kombination mit IDL/ODL*" (S.71) vorgeschlagene Integration von ODL in SDL,
- Erweiterungen von SDL, hauptsächlich zum Test von SDL-2000-Konzepten[1],
- SDL/GR durch die Abspeicherung aller Layouteigenschaften der Spezifikation[2].

Für den Programmierer von SITE-Komponenten stehen zwei Entwicklerkomponenten bereit, mit denen die Struktur der CR-Definition bzw. einer CR-Instanz komfortabel erklärt wird.

Das 1996 verabschiedete Austauschformat CIF für SDL/GR [Z.106] basiert auf einer durch Kommentierung angereicherten SDL/PR-Spezifikation. Die vom Autor betreute strategische Diplomarbeit von Dumke [Dum99] realisiert eine prototypische SITE-Komponente, die eine CIF-Instanz in eine CR-Darstellung überführen kann. Dieser Ansatz gestattet eine werkzeuggestützte Analyse und Weiterverarbeitung der Grafikinformationen einer Spezifikation.

Mit dem Ausbau der SITE-Komponenten zu einer integrierten Entwicklungsumgebung entstand das Problem, einen Informationsfluß über Eigenschaften der konkreten CR zwischen den Komponenten unterstützen zu müssen. Es wurde beschlossen, als Grundlage dieser Kommunikation CORBA [OMG98] zu benutzen. Im Rahmen der vom Autor betreuten Diplomarbeit von Mersewsky [Mer98] wurde die erste SITE-Komponente - ein Querverweisdienst - mit einer derartigen CORBA-Schnittstelle entwickelt. Basierend auf dem Querverweisdienst entstand auch ein externes Programm zum Editieren von SDL-Entwurfsmustern [Cis99], [Schw99].

---

1. algorithmische Notationen, Gatetypen als eine Vorgängerversion des *interface*-Konzepts, Referenzen, objektorientierte Datenelemente

2. Kommentierung, Formatierung des SDL/PR-Textes, Aufbau von SDL-Grafikelementen

1.5.2 Syntaktische Analyse

Die Generierung einer CR-Instanz aus einer oder gegebenenfalls mehreren SDL-Spezifikationen ist Aufgabe der Syntaxanalyse [Lut93]. SITE bietet derzeit nur Analysekomponenten für textuell vorliegende Spezifikationen. In den Arbeiten von Schwalbe [Schw97] wurde auch eine inkrementelle Syntaxanalyse entwickelt, die für einen von Kurzbach [Kur94] implementierten syntaxorientierten SDL/PR-Editor eingesetzt wurde. Die Pflege und Weiterentwicklung dieses Editors im Rahmen von SITE wurde nicht fortgesetzt, da

- die Basistechnologie *Interviews* [LCI92] nicht mehr weiterentwickelt wird und
- da es keine relevanten Anwendungen eines SDL/PR-Editors gab. Spezifikationen werden grundsätzlich als SDL/GR mit verfügbaren Editoren entwickelt.

Nach logischen Gesichtspunkten besteht die Syntaxanalyse aus unterschiedlichen Analyseteilen für SDL, ASN.1 und ITU-ODL. Die sprachspezifischen Quelltexteinschlüsse können selektiv bei der Übersetzung der Syntaxanalyse aktiviert werden.

Experimentelle Versionen der Syntaxanalyse sind die CIF-Analyse aus dem vorigen Abschnitt, die Unterstützung der Sprache *„SDL in Combination with ASN.1“* [Z.105] im Rahmen der betreuten Arbeiten von Dumke [Dum96] und Mersewsky [Mer97] sowie eine Analyse für SDL kombinierbar mit ASN.1 für die jeweils aktuellsten Sprachversionen (SDL-2000 bzw. X.680-ASN.1).

1.5.3 Semantische Analyse

Die erfolgreiche Durchführung der Semantikanalyse von SITE stellt die statische Konsistenz der CR einer Spezifikation sicher. Dabei wird die Quelldatei der CR-Instanz nicht verändert. Andere SITE-Komponenten verlassen sich auf die statische Korrektheit der CR und verhalten sich undefiniert, wenn diese nicht gegeben ist.

Die erste Version der Semantikanalyse (vgl. [Scha94]) realisierte den in der formalen SDL-Semantik vorgesehenen Transformationsansatz. Wegen gravierender Probleme wurde diese vollständige Transformation einer SDL-Beschreibung jedoch aufgegeben.

- Der Speicherbedarf der Analyse bei großen Spezifikationen war nicht mehr abzudecken bzw. führte zu unzumutbaren Laufzeiten, oft schon bei einfachen SDL-88-Spezifikationen.
- Fehlermeldungen, die sich auf transformierte Konstruktionen bezogen, waren unverständlich. Ein Konzept zur Rückinterpretation fehlte.
- Auch andere SITE-Komponenten, z.B. der Codegenerator, benötigen semantische Informationen, beispielsweise darüber, auf welche Definitionen Bezeichner verweisen. Da diese Komponenten nicht transformationsbasiert arbeiten, waren die Ergebnisse der Semantikanalyse nicht verwendbar.

In den betreuten Arbeiten von Böhme [Böh97] zur C++-Codegenerierung wurden diese Probleme behoben. Die CR-Instanz wird jetzt nur noch in Ausnahmefällen transformiert. Die eigentliche Transformationssemantik von SDL wird nur noch für Symboltabellen realisiert.

Der Funktionsumfang der Semantikanalyse ist nicht nur auf das Finden statischer Fehler beschränkt. Es werden auch Warnungen erzeugt, die potentielle Fehler der Spezifikation anzeigen. Das betrifft beispielsweise nicht genutzte Strukturen oder Aktionsverzweigungen, die nicht alle möglichen Fälle abdecken. Eine weitere Fehlerklasse zeigt potentielle Implementierungsfehler in der Semantikanalyse an. Fehlermeldungen können wahlweise mit dem syntaktischen Kontext (nützlich zur Positionierung grafischer Editoren) oder mit der SDL/PR-Position (Dateiname, Zeile und Spalte) ausgegeben werden.

Oft besteht der Wunsch, daß verschiedene SDL-Systeme eine untereinander konsistente Schnittstelle zur Systemumgebung besitzen. Diese Konsistenzeigenschaft ist nicht standardisiert. Eine SITE-spezifische Analyse wurde für verschiedene Projekte auf der Basis kompatibler Signalbeschreibungen an Kanälen zur SDL-Systemumgebung realisiert.

Die semantische Unterstützung alternativer Sprachen beschränkt sich derzeit auf ASN.1 gemäß der Beschreibung aus Abschnitt 4.2.3 „*ASN.1-Einbettung mit SITE*" (S.61). Eine ITU-ODL-Unterstützung entsprechend des Abschnitts 4.3.2 „*ODL-nach-SDL-Abbildung*" (S.73) befindet sich in der Entwicklungsphase.

Der bereits erwähnte Querverweisdienst ist eine abgerüstete Variante der Semantikanalyse, die durch Hinzufügen einer CORBA-Schnittstelle und Zusatzsoftware diesen umfangreichen Dienst realisiert. Durch die gemeinsame Nutzung der Quellen werden korrigierte Fehler der Semantikanalyse auch automatisch in den Querverweisdienst eingearbeitet.

1.5.4 Codegenerierung

Die Erzeugung eines ablauffähigen Programms aus einer SDL-Beschreibung begann basierend auf der Programmiersprache C++ [Str92] zunächst ausschließlich mit dem Ziel einer simulativen Ausführung [AFW93]. Diese Zielstellung wurde durch viele Projekte modifiziert, die aktuell verfügbaren Werkzeuge zur Codegenerierung werden im ersten Unterabschnitt erläutert.

Zur Programmcodegenerierung gehören stets zwei untrennbare Komponenten: der Codegenerator und eine passende SDL-Bibliothek. Der zweite Unterabschnitt befaßt sich mit einem Überblick zu den Bibliotheken.

Codegeneratoren

Es wird mit SITE angestrebt, für jeweils eine Zielsprache genau einen Codegenerator zu entwikkeln und zu warten. Das setzt voraus, daß die SDL-Bibliothek, auf die der generierte Code aufsetzt den Generator von der Zielanwendung entkoppelt. Allerdings muß der Generator für unvorhergesehene Projektforderungen aber auch wegen der Weiterentwicklung der Zielsprachen immer wieder angepaßt und verbessert werden, so daß der Idealfall einer stabilen Schnittstellen zwischen generiertem Code und SDL-Bibliothek praktisch nicht erreicht wird.

SDL-nach-C++: C++ ist eine weltweit eingesetzte, standardisierte [C++98] Programmiersprache. Der generierte Code [Lut94], [Böh97] basiert auf einer Schnittstelle für SDL-Struktureinheiten, die als C++-Klassen in einer Bibliothek realisiert sind. Deshalb ist der generierte Code in seiner Struktur der SDL-Spezifikation sehr ähnlich, portabel und relativ unabhängig vom Einsatzziel.

Wegen der Probleme mit dem internen Datenkonzept von SDL (siehe Abschnitt 2.1.8 „*Grundlegende Probleme*" (S.26)) enthält dieses Werkzeug auch einen ASN.1-nach-C++-Compiler mit Unterstützung für BER gemäß [X.209].

Für den Einsatz des generierten C++-Codes für reale Softwareprojekte werden eine Vielzahl von Optimierungsoptionen, C++-Makrokonstruktionen (z.B. zur Ausblendung von Codefragmenten für die Programmablaufverfolgung), *template*-Dateien (z.B. für die Erzeugung der Datei *Makefile*) und partielle Übersetzung von SDL- bzw. ASN.1-Modulen unterstützt. Da der generierte Code auch für Simulationen auf SDL-Niveau verwendet wird, ist die Testunterstützung für reale Programme stets Bestandteil des generierten Codes.

SDL-nach-Java: Java [SM99], [Kün97] ist nicht zuletzt durch die Anwendung in WEB-Browsern eine der bekanntesten objektorientierten Sprachen der heutigen Zeit. Die Java-Codegenerierung [Neu97] ist der C++-Codegenerierung sehr ähnlich, weshalb viele Prinzipien und Softwarekomponenten der SDL-nach-C++-Codegenerierung in die Java-Codegenerierung übernommen wurden. Bei der Entwicklung des Generators wurde der Schwerpunkt bisher projektbedingt auf die Realisierung der ASN.1-Datentypen mit BER-Unterstützung gelegt.

SDL-nach-SDL: Dieses vom Autor entwickelte SDL-Werkzeug ist auch als *pretty printer*-Komponente bekannt. Hauptaufgabe ist die Umformung einer SDL/PR-Beschreibung. Neben der Bereitstellung gebräuchlicher Ausgabeformate für SDL/PR-Beschreibungen ist der SDL-nach-SDL-Compiler in der Lage, Modifikationen vorzunehmen.

- Es werden alle Kommentare aus der vorhandenen SDL-Beschreibung entfernt und eine automatische Strukturierung durchgeführt. Über zahlreiche Optionen kann Einfluß auf die Regeln zur Strukturierung genommen werden. Schlüsselwörter und Zeichenketten werden hervorgehoben, sofern dieses vom Ausgabeformat unterstützt wird.

  Diese Funktionalität wird benötigt, wenn eine aus SDL/GR automatisch erzeugte SDL/PR-Beschreibung durch einen menschlichen Nutzer weiterverarbeitet wird.
- Die SITE-spezifische Kombination von SDL mit ASN.1 (siehe Abschnitt 4.2.3 „*ASN.1-Einbettung mit SITE*" (S.61)) ist syntaktisch nicht standardkonform. Mit dem SDL-nach-SDL-Compiler kann eine SDL-Beschreibung konform zum Standard Z.105 generiert werden. Auf der Grundlage der Arbeiten in [Mer97] steht auch der umgekehrte Weg prototypisch zur Verfügung.
- Mit Hilfe des SITE-Querverweisdiensts kann für das Ausgabeformat HTML eine Querverweisstruktur für Bezeichner auf deren Definition erzeugt werden.

Die Erweiterung für eine automatische Transformation von SDL-96 nach SDL-2000 ist möglich.

SDL-Bibliotheken

Die Schnittstellen des generierten C++- bzw. Java-Codes werden von sehr verschiedenen Bibliotheken implementiert. So können aus SDL-Spezifikationen Simulationsprogramme, prototypische und sogar reale Softwarekomponenten erzeugt werden. Die Bibliotheken bestehen wiederum aus zwei wesentlichen[1] Teilbibliotheken.

SDL-Bibliothek: Die SDL-Bibliothek stellt alle Definitionen bereit, die der generierte C++/Java-Code erwartet. Auch ist schon eine gewisse Grundfunktionalität verschiedener SDL-Strukturen implementiert, beispielsweise sind alle vordefinierten Daten schon vorhanden. Generator- und Bibliotheksversion müssen jeweils übereinstimmen, da Änderungen am Generator oft Schnittstellen zur Bibliothek betreffen. In der Praxis sind die erforderlichen Anpassungen an die jeweils aktuelle Generatorversion möglich und insbesondere nicht weiter aufwendig.

Kommunikations- und Verteilungsaspekte der Laufzeitumgebung werden selbst in der SDL-Bibliothek mit einer Schnittstelle entkoppelt, deshalb ist sie immer noch unabhängig von Betriebssystemarchitekturen. Bei der Verwendung unterschiedlicher Laufzeit- und Kommunikationsbibliotheken variiert diese Schnittstelle jedoch, so daß eine entsprechende Anpassung stets neu, allerdings unabhängig vom Generator vorgenommen werden muß.

Laufzeit- und Kommunikationsbibliothek: Diese Bibliothek stellt die eigentliche Basisfunktionalität für Kommunikation (in dieser Arbeit als Basiskommunikation bezeichnet) und Ablaufverhalten bereit. In Industrieprojekten, die SITE-Codegenerierung nutzen, wird diese Bibliothek von den Projektpartnern vorgegeben. Deshalb sind die angepaßten SDL-Bibliotheken projektspezifisch.

Einen Überblick der verfügbaren C++-Bibliothekskomponenten gibt Bild 3. Die Java-basierte SDL-Bibliothek ist eine Prototypentwicklung, die keine externe Kommunikation von SDL-Systemen gestattet und das Java-*thread*-Konzept zur verteilten Ausführung nutzt.

1. Bibliotheken aus dem Betriebssystem und Softwareprodukte für Teilaspekte (z.B. Codierung von Werten) werden hier nicht betrachtet.

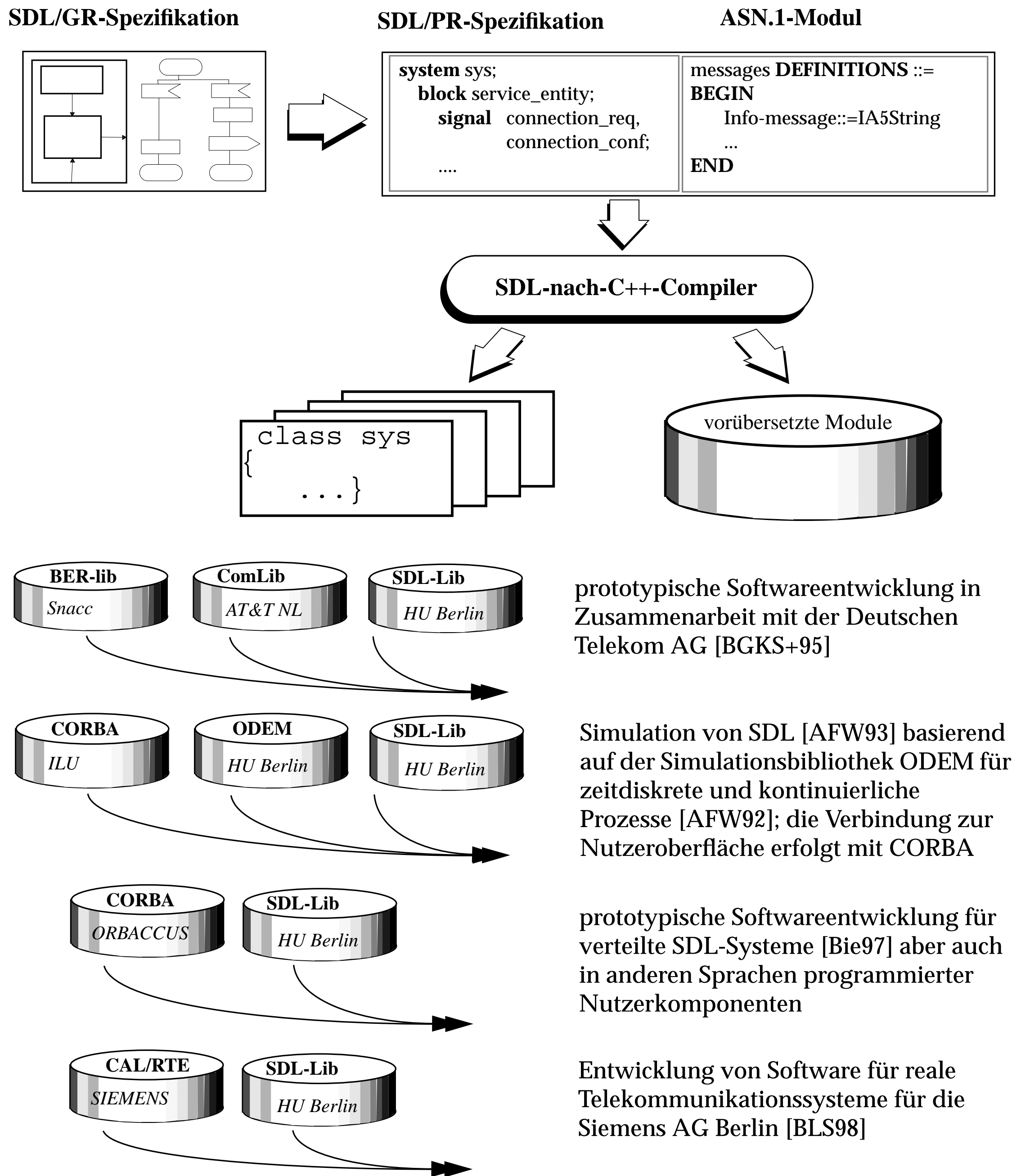

Bild 3: SITE-C++-Bibliotheken

Kapitel 2

SDL-Datenbeschreibungen

Nachdem im vorhergehendem Kapitel SDL sehr allgemein vorgestellt wurde, widmet sich dieses Kapitel ausschließlich dem von SDL-96 unterstützten Datenkonzept. Neben einer überblicksmäßigen Darstellung werden spezifische Probleme hinsichtlich eingeschränkter oder ungebräuchlicher Verwendung konkreter Notationen und der Implementierbarkeit herausgearbeitet. Eine Verallgemeinerung dieser Probleme führt zu einem Katalog von Anforderungen für Änderungen des vorhandenen SDL-Datenkonzepts, der in den nachfolgenden Kapiteln immer wieder zu Einschätzungen herangezogen wird. Die Relevanz sowohl der aufgestellten Katalogpunkte als auch der zugrundeliegenden Problemdiskussion ist natürlich an die Zielstellung des Einsatzes einer Spezifikation gebunden.

Bei der Einführung des Datenkonzepts wird schrittweise vorgegangen.

1. Zunächst werden Datenkonzepte diskutiert, deren Semantik sich ausschließlich mit dem ACT ONE-Modell erklären lassen. Die Zusammenfassung dieser Konzepte ergibt das ACT ONE-Basismodell von SDL.
2. Das ACT ONE-Basismodell besitzt als Einbettung in die Sprache SDL Berührungspunkte zu anderen Konzepten der Sprache (z.B. Strukturierung). Das führt zu Erweiterungen der Notationen des Basismodells, die insbesondere auch wesentliche technische Verbesserungen bei der Entwicklung von Datenspezifikationen mit sich bringen. Der Abschnitt 2.2 „*Zusätzliche Datenkonzepte*" (S.27) ist diesen Erweiterungen gewidmet.
3. Neben der formalen Definition von Daten gibt es ein syntaktisches Konzept, externe Datenbeschreibungen in SDL zu integrieren. Die Semantik dieser Konstruktionen ist aus der Sicht von SDL offen. Wegen dieses Gegensatzes zur formalen Definition von SDL wird dieses Datenkonzept separat behandelt.

Im letzten Abschnitt wird schließlich aus den zuvor identifizierten Einschränkungen und Problemen der Forderungskatalog für die Veränderung des SDL-Datenmodells abgeleitet.

2.1 ACT ONE-Basismodell in SDL

Wie bereits im Abschnitt 1.3 „*Definition der Sprache SDL im Sprachstandard*" (S.10) bemerkt, ist die Verwendung des ACT ONE-Modells zur Erklärung der Datensemantik bindend, obwohl es eine alternative, ebenfalls standardisierte Semantikdefinition in den Anhängen F2 und F3 des Sprachstandards Z.100 gibt.

Im Rahmen dieser Arbeit werden nur die wesentlichen Eigenschaften und Begriffe herausgearbeitet, ohne die sich daraus ergebenen Sachverhalte mathematisch zu beweisen. Viele Beweise von Eigenschaften erfordern einen exakteren Zugang[1], der für die eigentliche Problematik, Datenbeschreibungen zu nutzen und zu verbessern, nicht notwendig ist.

1. Eine ausführliche Darstellung der ACT ONE-Theorie findet man in [EhMa85].

2.1.1 Definitionen und Bezeichner

Das Datenkonzept von SDL kennt zwei Arten von Definitionen.

Datentypen: Es gibt mehrere Möglichkeiten, einen Datentyp zu definieren: als Sorte, abstrakte Sorte und *syntype*-Konstruktion. Die dabei verwendeten Bezeichner für die Datentypen seien in einer Menge S zusammengefaßt.

Datenwerte: Alle Bezeichner, die an der Bildung von Datenwerten beteiligt sind, werden in der Menge W zusammengefaßt. Derartige Bezeichner verweisen auf die Definitionen von Variablen, Literalen, Operatoren und Konstanten.

Wegen der unterschiedlichen Art der Definitionen sind Bezeichner für Datentypen stets verschieden von Datenwerten. Die Mengen S und W sind also disjunkt.

2.1.2 Signaturen und Sorten

Eine Signatur ist ein Element aus der Menge $W \times S^n \times S$, wobei n eine natürliche Zahl ist. Geordnete Paare für den Spezialfall $n = 0$ werden als Literale bezeichnet, die Tripel für $n > 0$ heißen Operatoren. Der Bezeichner des Literals bzw. des Operators ist die jeweils erste Komponente des Tupels. Sofern sich die Signaturen unterscheiden, können gleiche Bezeichner für Operatoren mehrfach verwendet werden. Dieses Konzept wird wie allgemein üblich, Überladung genannt.

Syntaktisch heißt die Definition eines Datentyps mit seinen Signaturen Sorte. Eine Sortendefinition hat immer einen Namen, der Bezeichner der Sorte ist ein Element der Menge S. Literale werden nur mit ihrem Namen in einer Sorte definiert, die Signaturen ergeben sich implizit aus den Bezeichnern des Literals und der definierenden Sorte. Bei Operatoren sind die Signaturen explizit anzugeben. Daraus folgt, daß die Bindung eines aus logischer Sicht zur Sorte gehörenden Operators an die Sorte nicht zwingend ist. Es gilt, daß für jeden in einer Signatur vorkommenden Bezeichner, der Element der Menge S ist, eine Sortendefinition in der SDL-Spezifikation vorhanden sein muß. Literal- und Operatordefinitionen in einer Sortendefinition sind optional.

Beispiel:

```
newtype Boolean
    literals True, False;
    operators
        "and" : Boolean, Boolean -> Boolean;
endnewtype;
```

Die Signaturen der Sorte *Boolean* sind die Literale *(True, Boolean), (False, Boolean)* und der Operator *("and", (Boolean,Boolean), Boolean).*

Signaturen werden im folgenden in der SDL-spezifischen Notation angegeben:

$$True\colon \rightarrow Boolean,\ False\colon \rightarrow Boolean,\ "and"\colon Boolean, Boolean \rightarrow Boolean.$$

Zur Beschreibung sehr großer oder unendlicher Mengen von Literalen einer Sorte können reguläre Ausdrücke (Z.100 Abschnitt 5.3.1.14 „*Name class literals*“) verwendet werden. Beispielsweise ist die Menge der Literale für ganze Zahlen so definiert:

```
newtype Integer
    /* Ganze Zahlen bestehen aus mindestens einer Ziffer,
       führende Nullen sind erlaubt.
    */
    literals nameclass ('0':'9')* ('0':'9');
    ...
endnewtype;
```

Für derartige Konstruktionen ist in der statischen Semantik ein Transformationsmodell definiert. Die Menge aller zum regulären Ausdruck passenden lexikalischen Einheiten wird berechnet, um daraus Literaldefinitionen zu konstruieren.

literals 0,1,2,3,4,5,6,7,8,9,00,01,02,03,04,05,06,07,08,09,10,...;

Ohne die Nutzung regulärer Ausdrücke wäre die Spezifikation einer unendlich großen Anzahl von Literalen nicht möglich.

2.1.3 Variablen

Eine Variable ist ein geordnetes Paar der Menge $W \times S$. Da die Art der Definition von Operatoren, Literalen und Variablen gleich ist, dürfen Bezeichner von Variablen aus der Menge *W* nicht mit Bezeichnern von Literal- oder Operatordefinition übereinstimmen. Die Sorte einer Variablen ist das zweite Element des geordneten Paars.

Die Definition von Variablen kann unterschiedliche syntaktische Formen annehmen:

- lokale Variablen (in *process*-, *service*-, *procedure*-Definitionen),
- formale Parameter (von *process*-, *procedure*-Definitionen),
- Iteratoren für Werte eines Datentyps (für axiomatische Definitionen),
- Konstanten.

2.1.4 Terme einer Sorte

Die Definition der Terme einer Sorte erfolgt induktiv:

1. Wenn $lit: \rightarrow s$ als Literal einer Sorte s definiert ist, dann ist der Bezeichner des Literals ein Term der Sorte *s*.
2. Wenn v eine Variable der Sorte s ist, so ist der Bezeichner der Variablen ein Term der Sorte *s*.
3. Seien $t_1, ..., t_n$ jeweils Terme der Sorten $s_1, ..., s_n$ und op ein Operator mit der Signatur $op{:}s_1, ..., s_n \rightarrow s_{res}$. Dann ist $op(t_1, ..., t_n)$ als Anwendung des Operators auf die gegebenen Terme ein Term der Sorte s_{res}.
4. Nur mit den Regeln 1, 2 und 3 lassen sich Terme einer Sorte konstruieren[1].

Beispiel: Mit der schon vorher eingeführten Definition von *Boolean* gibt es unendlich viele Terme von *Boolean*, z.B.:

True
"and"(True, False)
"and"(True, "and"(True,False))

Neben dem Begriff des „Terms“ wird in SDL auch der Begriff des „Ausdrucks“ verwendet. Das sind eigentlich nur unterschiedliche syntaktische Strukturen. Terme gibt es nur innerhalb von Datendefinitionen. Wird ein Term außerhalb einer Datendefinition notiert, beispielsweise bei einer Zuweisung im Prozeß, so ist dieser Term in seiner syntaktischen Notation ein Ausdruck. Ausdrücke können zusätzliche syntaktische Konstruktionen enthalten. Feld- oder Indexzugriffe aus dem Abschnitt 2.2.6 „*Transformationsbasierte Ausdrücke*” (S.34) aber auch der Zugriff auf die aktuelle Zeit des SDL-Systems mit dem Schlüsselwort **now** sind Beispiele für derartige Erweiterungen.

1. Ein andersartig konstruierter Term ergibt einen statischen Fehler.

2.1.5 Zuweisungskompatibilität

Jede Verwendung eines Terms oder Ausdrucks im Kontext eines SDL-Sprachkonstrukts unterliegt zusätzlichen semantischen Regeln. Bezogen auf Zuweisungen und die Übergabe von Parametern sind das die Zuweisungsregeln. In vielen Fällen ist durch den Sprachkontext ein Datentypbezeichner vorgegeben, der entsprechend den Zuweisungsregeln die Nutzung der Terme verschiedener Sorten einschränkt. Alle für die Terme zulässigen Sorten sind dann zuweisungskompatibel zur vorgegebenen Sorte. Dieser Begriff ist natürlich auch für die Übergabe von Parametern bei Operatoren anwendbar, in gültigen Termen sind alle aktuellen Parameter von Operatoranwendungen zuweisungskompatibel.

In SDL gilt, daß zwei verschiedene Sorten nicht zuweisungskompatibel sind.

Beispiel: Angenommen, es gäbe neben *Boolean* eine weitere Sorte *Null*:

newtype Null
  literals Null;
endnewtype;

und zwei Zuweisungen:

dcl b Boolean := "and"(Null, True),
  n Null := True;

Dann ist der Term *"and"(Null, True)* auf der Grundlage der Signaturen nicht konstruierbar, folglich kein gültiger Term. Betrachtet man den Aspekt der Zuweisungskompatibilität, dann ist der erste aktuelle Parameter des Operators *"and"* ein Term der Sorte *Null*. Der formale Parameter des Operators ist jedoch *Boolean*. Das ist ein Verstoß gegen die Zuweisungsregeln.

Auch der Term *True* ist kein gültiger Term von *Null* wie eigentlich in der zweiten Zuweisung vorausgesetzt.

Die Eigenschaft, daß zwischen zwei jeweils verschiedenen Sorten keine Zuweisungskompatibilität besteht, wird strenges Datentypkonzept genannt. Die Überprüfung, daß die Zuweisungsbedingungen nicht verletzt werden, wird auf der Grundlage einer vorliegenden Datenbeschreibung durch die statische Semantikanalyse vorgenommen.

2.1.6 Werte

Betrachtet man die Terme *True* und *not False*, so sind diese semantisch gleich. Diese Termgleichheit wird durch den Begriff des Werts repräsentiert und in SDL mit Axiomen[1] beschrieben. Ein Axiom besteht aus zwei Termen, die semantisch gleich sind. Mit Hilfe von Ableitungsregeln und gegebenen Axiomen läßt sich auch für nicht in Axiomen explizit angegebene Terme feststellen, ob sie zu anderen Termen semantisch gleich sind. Termgleichheit ist letztendlich eine Relation über alle Terme einer Sorte, die insbesondere die Eigenschaft einer Äquivalenzrelation besitzt. Entsprechend des Hauptsatzes für Äquivalenzrelationen zerfallen damit die Terme einer Sorte in disjunkte Äquivalenzklassen. Diese Termklassen sind die Werte der Sorte. In diesem Sinne sind Terme Repräsentanten von Werten. SDL nutzt für Termgleichheit das Symbol "==", welches nicht mit dem Vergleichsoperator "=" zu verwechseln ist.

Beispiel: Eine Spezifikation der Sorte *Boolean* mit Axiomen wäre

1. Das entsprechende ACT ONE-Konzept sind Gleichungen.

```
newtype Boolean
    literals True, False;
    operators
        "and" : Boolean, Boolean -> Boolean;
        "not"   : Boolean -> Boolean;

    axioms
        "and"(True, True)    ==    True;
        "and"(True, False)   ==    False;
        "and"(False, True)   ==    False;
        "and"(False, False)  ==    False;
        "not"(True)    == False;
        "not"(False)   == True;
endnewtype;
```

Es gibt zwei Ableitungsregeln für Terme:

1. Besitzen die beiden Terme eines Axioms keine Variablen, können diese Terme innerhalb eines Terms ausgetauscht werden. Beispielsweise gilt:

   "and"(True,"and"(True,True)) == "and"(True,True) == True == "not"(False)

2. Enthalten die Axiome Variablen, so können die Variablen durch Terme der entsprechenden Sorten der Variablen ersetzt werden. Innerhalb eines Axioms müssen gleiche Variablen natürlich immer mit dem gleichen Term ersetzt werden. Diese Regel wird an einer Erweiterung der Sorte *Boolean* demonstriert:

   ```
   operators
           "or" : Boolean, Boolean -> Boolean;
   axioms
           for all b1, b2 in Boolean(
               "or"(b1, b2)    == "not"("and"("not"(b1), "not"(b2)));
           );
   ```

   Betrachtet man jetzt den Term *"or"(True,False)*, so paßt er genau zur linken Seite des Axioms. Es gelten folgende Termgleichungen:

   "or"(True,False) == "not"("and"("not"(True), "not"(False))) ==
   "not"("and"("False,True)) == "not"(False) == True

Die Angabe einer Kette von Termgleichungen ist ein Nachweis, daß sich Terme in der gleichen Termklasse befinden. Ohne Axiome sind alle Terme einer Sorte semantisch verschieden. Die Bestimmung der Zugehörigkeit eines Terms zu einem Wert ist die Auswertung oder Berechnung des Terms. Ist der Term die Anwendung eines Operators auf andere Terme, nennt man das auch die Berechnung des Operators.

Es lassen sich auch allgemeine Eigenschaften beweisen, beispielsweise gibt es für die Sorte *Boolean* genau zwei Termklassen[1]:

[*True, "and"(True,True), "and"(True,"and"(True,True)), ...*] und
[*False, "and"(False,True), "and"(False,"and"(True,True)), ...*]

Die Anwendbarkeit eines Axioms für Ableitungsregeln kann auch an eine Bedingung gebunden werden, die auf Termgleichheit basiert. Diese bedingten Axiome sind für Einschränkungen des Wertebereichs von Variablen sinnvoll. Die vordefinierte Sorte *Integer* enthält beispielsweise eine Regel

```
a>=0 and b>a == True ==> a / b == 0;
```

1. Der Beweis kann hier induktiv über die Anzahl der "*and*", "or"- und "not"-Operatoren in einem Term der Sorte *Boolean* erfolgen (strukturelle Induktion über die Termkonstruktion von *Boolean*).

Für die Verwendung von Literalen basierend auf regulären Ausdrücken gibt es eine spezielle Variante von Variablendefinitionen, die einen Zugriff auf die Zeichenrepräsentation des Literalnamens gestattet. Das ist für unendliche Literalmengen notwendig. Beispielsweise werden auf diese Art die Literale der Zahlen in die entsprechenden Termklassen eingebunden:

```
map for all a,b,c in Integer literals (
    Spelling(a) == Spelling(b) // Spelling(c), Length(Spelling(c)) == 1
        ==> a == b * (9+1)+c;
);
```

Die über den Rahmen einer Einführung in die ACT ONE-Problematik hinausgehenden Details der letzten beiden Beispielfragmente sind in den vordefinierten Datentypen [Z.100D] enthalten.

2.1.7 Prinzip der partiellen Datendefinition

Erst die Fixierung der Menge aller Axiome und Signaturen in den verschiedenen Sorten einer Spezifikation ergibt eine vollständige Datenspezifikation, da die Verteilung der Axiome auf die Sorten beliebig ist. Einschränkend gilt jedoch für SDL, daß Axiome einer Sorte nicht die Termklassen anderer Sorten verändern dürfen. Im Beispiel

```
newtype Enum
    literals one, two, three;
    operators lower_than_three : Enum -> Boolean;
    axioms
        lower_than_three(one) == True;
        lower_than_three(two) == True;
endnewtype;
```

wird nicht spezifiziert, daß *lower_than_three(three)* zu einer der Termklassen mit dem Repräsentanten *True* oder *False* gehört. Unter der Annahme, daß diese Spezifikation nicht in einer anderen Sorte erfolgt, ist *lower_than_three(three)* ein neuer Wert der Sorte *Boolean*. Dementsprechend ist laut Semantik von SDL die Definition der Sorte *Enum* falsch.

2.1.8 Grundlegende Probleme

Mit der semantischen Fundierung von Daten mit ACT ONE-Konzepten sind fundamentale Probleme der prinzipiellen Berechenbarkeit, des Tests von Bedingungen der statischen Semantik und letztendlich der Nutzerakzeptanz verbunden.

Berechenbarkeit: Im Abschnitt „*Term rewriting with equations*“ in [EhMa85] wird darauf hingewiesen, daß es keinen allgemeinen Algorithmus zur Bestimmung der Termgleichheit gibt. Bezieht man diese Aussage konkret auf die Operatoren, folgt sofort, daß man aus einer beliebigen Datenbeschreibung keinen Algorithmus zur Berechnung eines Operators generieren kann. In der SDL-Praxis gibt es jedoch Werkzeuge, die eine Codegenerierung aus ACT ONE-basierten Datentypen zumindest eingeschränkt unterstützen. Dafür gibt es zwei Gründe:

a. Es gibt Einschränkungen für Axiome, die eine endliche, nicht zyklische und vor allem algorithmisch bestimmbare Folge von Termumformungen sicherstellen, mit denen nachgewiesen werden kann, ob zwei Terme in der gleichen Termklasse liegen oder nicht. Allerdings ist die Frage, ob eine gegebene Spezifikation die Einschränkungen erfüllt, unter Umständen nicht entscheidbar.

b. Man nutzt ein Programm, daß die vordefinierten Datentypen oder zu Datentypen erweiterbare Schemen im Sinne der ACT ONE-Semantik nachbildet. Oft werden sogar solche

Programme verwendet, die durch die Beschränkung der unendlich großen Datentypen bewußt von der vorgegebenen Semantik abweichen. Damit lassen sich auch relativ komplexe Datenstrukturen entsprechend der vordefinierten Schemen implementieren.

statische Semantik: Der Test der statischen Bedingung, daß eine Sorte nicht die Termklassen einer anderen Sorte verändert, ist problematisch. Ein SDL-Werkzeug müßte beispielsweise für jeden Wert von *Enum* aus dem vorigen Abschnitt den Operators *lower_than_three* anwenden und die Zugehörigkeit zur Termklasse bestimmen, das dem Problem der Termgleichheit entspricht. Darüber hinaus kann die Sorte auch unendlich viele Werte besitzen, so daß bei komplizierten Sorten, z.B. der Beschreibung von Strukturen, dieser Fehler nicht so offensichtlich ist. In diesen Fällen wird der Nutzer auf eine Werkzeugunterstützung angewiesen sein.

Akzeptanz: SDL wird von Softwareentwicklern genutzt, die Datenbeschreibungen in Form üblicher Programmiersprachen erwarten und die axiomatischen Beschreibungsmöglichkeiten, nicht zuletzt wegen mangelhafter Werkzeugunterstützung ablehnen.

2.2 Zusätzliche Datenkonzepte

Erweiterungen der ACT ONE-basierten Notationen sind untereinander relativ unabhängige Sprachkonzepte. Deshalb sind die folgenden Ausführungen im Gegensatz zum vorigen Abschnitt direkt in eine Vorstellung des Sprachkonzepts und einer Problemdiskussion gegliedert.

2.2.1 Hierarchische Strukturierung von Datendefinitionen

SDL hat ein hierarchisches Strukturierungskonzept. Definitionen sind nur sichtbar, wenn sie auf der gleichen Hierarchiestufe oder in einer höheren definiert sind bzw. aus einem SDL-Modul importiert werden. Die Definition von Daten ist in dieses Konzept im wesentlichen integriert. Sowohl Operatoren als auch Literale widersprechen der streng hierarchischen Sichtbarkeit, sie sind auch außerhalb der definierenden Sorte sichtbar. Signaturen und Terme dürfen nur aus sichtbaren Bezeichnern konstruiert werden. Auch die Berechnung eines Terms erfolgt nur auf der Grundlage von Axiomen, die in sichtbaren Sorten enthalten sind[1].

Ein Bezeichner enthält die Information, in welcher Hierarchiestufe die zugehörige Definition steht, in Form einer vollständigen Pfadangabe (das *full qualifier*-Konzept). Der Pfad besteht aus einer Aufzählung des Namens und der Art der jeweils definierenden Struktur bis zum SDL-System oder einem SDL-Modul. Demzufolge müssen nur die Namen von Definitionen der gleichen Art in einer Hierarchiestufe eindeutig sein, wobei es für Operatoren noch zusätzlich das Überladungskonzept (vgl. 2.1.2 „*Signaturen und Sorten*" (S.22)) gibt. Im Bild 4 gibt es zwei Literale mit dem Namen *l* aber den unterschiedlichen Bezeichnern *System s / Datentyp s / Datenwert l* bzw. *System s / Block b2 / Prozeß p / Datentyp s / Datenwert l*. Auch die Vergabe des Namens *s* ist wegen der Definitionsarten kein Problem: *System s / Block b2 / Prozeß p / Datentyp s* bzw. *System s / Block b2 / Prozeß p / Datenwert s* sind die Bezeichner der Sorte bzw. der Variablen. Natürlich darf in beiden Sorten *s* keine Variable mit dem Namen *l* eingeführt werden. Die Bezeichner für Literal und Variable wären gleich.

1. Genau diese Eigenschaft motiviert die Regel, daß Axiome nicht die Wertestruktur anderer Sorten ändern dürfen.

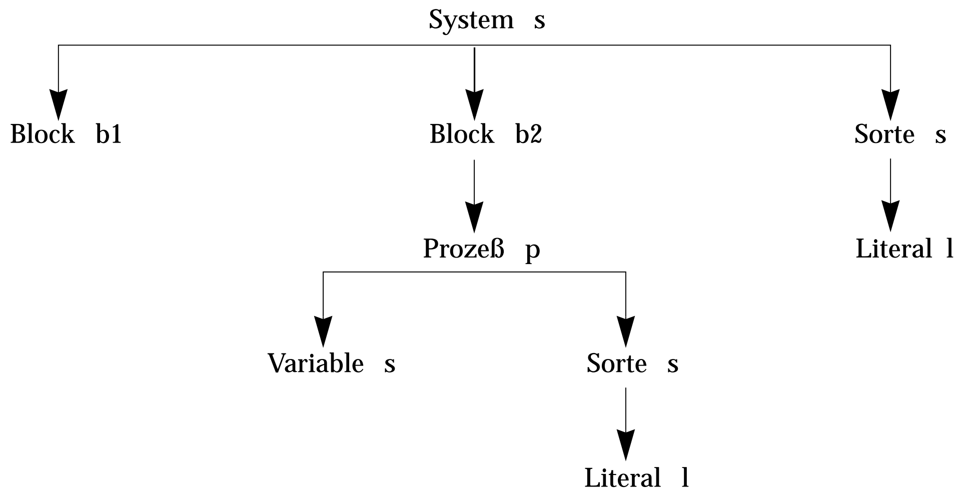

Bild 4: Beispiel einer SDL-Hierarchie

Sofern die Bezeichnung einer Definition entsprechend vorgegebener Regeln[1] eindeutig ist, können hierarchische Bestandteile des Bezeichners entfallen. Die Art der referenzierten Definition wird in Bezeichnern nie angegeben, da sie sich aus der Verwendung ergibt. Bei einer statischen Analyse einer SDL-Spezifikation müssen die fehlenden Pfadbestandteile eines Bezeichners ergänzt werden. Das ist die Auflösung der Bezeichner, das Ergebnis ist ein voll qualifizierter Bezeichner. Im Bild 4 ist *b2* innerhalb des Prozesses *p* eindeutig, es gibt aber zwei sichtbare Sorten *s*. Eine eindeutige Bezeichnung wäre *Prozeß p / Datentyp s*.

Problem mit Bezeichnerauflösungen: Die informale Sprachbeschreibung von SDL verwendet den Begriff der Auflösung eines Bezeichners nur sehr inkonsequent. Hersteller von SDL-Werkzeugen sind hier auf die formale Sprachdefinition angewiesen. Das ist aber durch die Anwendung der äußerst komplexen Transformationsmodelle für Vererbung und Module ohne entsprechende Werkzeugunterstützung nur noch in Teilaspekten nachvollziehbar. Die formale Semantik ist nicht elektronisch verfügbar, entsprechend ist das Verhalten der SDL-Werkzeuge bei nicht eindeutigen Bezeichnerauflösungen verschieden.

Problem mit Ausdrücken: Alle Arten von Zahlen sind in SDL Literale, d.h. es gibt viele Literale mit gleichen Namen. Beispielsweise ist *0* ein Literal der Sorten *Integer*, *Real*, *Duration* und *Time*. Um zu vermeiden, daß bei Bezeichnern von derartigen Literalen auch der hierarchische Bestandteil mit anzugeben ist, gibt es Regeln, die eine eindeutige Zuordnung von Namen entsprechend der erwarteten Sorte erlauben (*resolution by context*). Diese Regeln sind insbesondere bei Operatorrufen nicht immer intuitiv anwendbar.

Einschränkung bei verschachtelten Sortendefinitionen: Die Definition einer Sorte direkt in einer Sorte ist nicht möglich. In Kombination mit Kontextparametern kann die Nutzung innerer Sorten für Hilfsdatentypen sinnvoll verwendet werden.

1. In Kombination mit Vererbung, Kontextparametern und Modulen sind diese Regeln äußerst komplex, daß eine detaillierte Darstellung den Rahmen einer Einführung sprengt.

2.2.2 Vererbung

Das allgemeine SDL-Konzept, eine abgeleitete Definition mittels Vererbung aus einer Basisdefinition zu erzeugen, ist auch auf Sorten anwendbar. Die Besonderheiten des zugrundeliegenden Transformationsmodells für die Erklärung der Semantik soll hier für Sorten näher betrachtet werden. Aus mathematischer Sicht ist eine Sorte eine (Term-) Algebra. Leitet man eine neue Algebra aus einer vorhanden ab, so ist der Begriff der linearen Abbildung[1] wichtig, um zu beschreiben, daß sich ein Operator in der neuen Algebra wie in der ursprünglichen Algebra verhält. Genau dieses mathematische Konzept wird für Vererbung von Sorten in SDL verwendet.

Entsprechend des Transformationsmodells werden Literal- und Operatordefinition der Basissorte in die Ableitung kopiert, d.h. es wird eine neue Definition dieser Literale und Operatoren mit einem, dem neuen Pfad entsprechenden Bezeichner erzeugt. Die kopierten Literale sind damit Literale der abgeleiteten Sorte. Bei der Kopie der Operatorsignaturen werden alle Bezeichner der Basissorte durch den Bezeichner der Ableitung ersetzt, es ändert sich also die Signatur der Operatoren. Diese Relation der Signaturen der Basissorte und der Ableitung wird als kovariant bezeichnet. Damit hat die abgeleitete Sorte Terme entsprechend der neuen Signaturen. Mittels spezieller syntaktischer Konstruktionen können die Namen von Literalen und Operatoren bei der Vererbung geändert bzw. Literale oder Operatoren von der Vererbung ausgeschlossen werden.

Sowohl Basissorte als auch die Ableitung sind verschiedene Sortendefinitionen. Die Regeln der Zuweisungskompatibilität aus Abschnitt 2.1.5 „*Zuweisungskompatibilität*“ (S.24) gelten unabhängig von der Vererbungsbeziehung, es besteht also keine Zuweisungskompatibilität zwischen Basissorte und Ableitung.

Die Axiome der Basissorte werden nicht kopiert. Statt dessen wird ein, dem Nutzer verborgener Operator eingeführt, der eine lineare Abbildung von Werten der Basissorte auf Werte der Ableitung bereitstellt. Der häufige Anwendungsfall für Vererbung ist die Definition zusätzlicher Operatoren ohne die Werte einer Sorte zu verändern. Hier ist diese lineare Abbildung eineindeutig. In der Ableitung können neben neuen Operatoren auch weitere Literale und Axiome spezifiziert werden, so können zwei Werte der Basisklasse in der Ableitung zu einem Wert zusammengefaßt werden, aber auch wegen neuer Literale und Operatoren Werte hinzukommen.

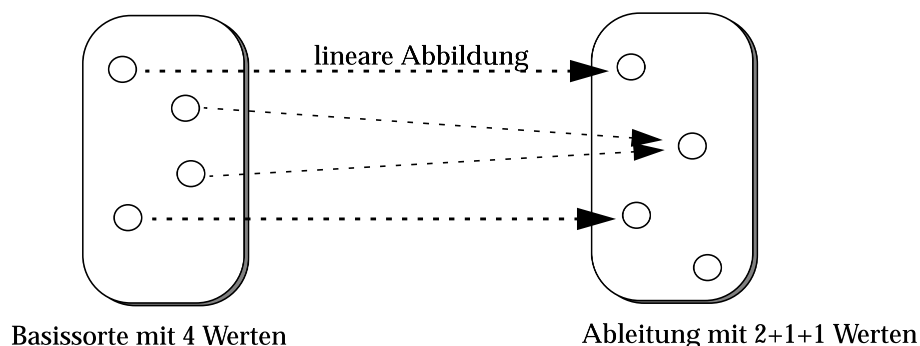

Bild 5: Lineare Abbildung bei Sortenvererbung

Da durch die lineare Abbildung die Anwendung der Operatoren auf neue Werte nicht definiert ist, müssen entsprechende Axiome für die neuen Werte in der Ableitung spezifiziert werden.

1. Seien A und B Algebren mit je einem Operator op:A->A bzw. OP:B->B, dann ist L:A->B eine lineare Abbildung, wenn für alle Elemente a aus A gilt L(op(a)) = OP(L(a)). Diese Eigenschaft kann für mehr als zwei Sorten und komplexe Signaturen verallgemeinert werden.

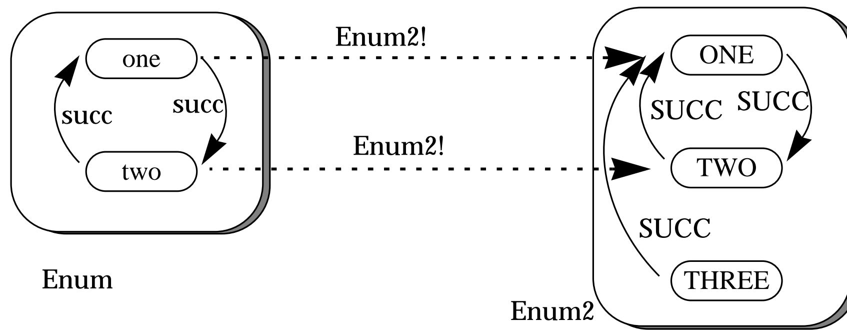

Bild 6: Beispiel Enum und Enum2

Beispiel:

newtype Enum
  literals one, two;
  operators succ: Enum -> Enum;
  axioms
    succ(one) == two;
    succ(two) == one;
endnewtype;

newtype Enum2 **inherits** Enum **operators all adding**
  literals three;
  axioms
    succ(three) == one;
endnewtype

Die Sorte *Enum2* ist entsprechend der semantischen Transformation wie folgt definiert[1]:

newtype Enum2
  literals ONE, TWO, THREE;
  operators
    SUCC : Enum2 -> Enum2;
    Enum2! : Enum -> Enum2; /* `die lineare Abbildung` */

  axioms
    /* `axiomatische Definition der linearen Abbildung` */
    Enum2!(one) == ONE;
    Enum2!(two) == TWO;
    for all e **in** Enum(
      Enum2!(succ(e)) == SUCC(Enum2!(e));
    );
    /* `und das neue Axiom aus Enum2 für den neuen Wert */`
    SUCC(THREE) == ONE;
endnewtype;

Schon an dem einfachen Nachfolgeoperator *succ* erkennt man, daß eine lineare Abbildung zur Erklärung von Vererbung nicht immer dem entspricht, was man intuitiv erwarten würde.

Werteproblem: Der Operator *SUCC* verhält sich sicher nicht wie erwartet. Fügt man zusätzlich das Axiom

1. Alle in *Enum2* definierten Operator- und Literalnamen sind zur besseren Unterscheidung von Namen der Sorte *Enum* groß geschrieben.

SUCC(TWO) == THREE;

ein, so sieht man schon im Bild 6, wenn man den Pfeil für *SUCC* von *TWO* nach *THREE* mit einzeichnet, daß die Terme *ONE* und *THREE* plötzlich einen Wert repräsentieren. Die Ableitung durch sukzessive Anwendung der Axiome liefert den Beweis:

ONE == Enum2!(one) == Enum2!(succ(two)) == SUCC(Enum2!(two)) ==
SUCC(TWO) == TREE

In diesem Beispiel ist das falsche Verhalten des Operators *SUCC* wegen der endlichen Wertestruktur noch gut erkennbar. Insbesondere bei Operatoren, deren Signaturen noch andere Sorten einbeziehen, gibt es aber auch Fälle, in denen die Wertestruktur der anderen Sorten verändert wird. Man benötigt also Werkzeugunterstützung, die jedoch wegen der Schwierigkeit, Termgleichheit zu berechnen (vgl. 2.1.8 „*Grundlegende Probleme*" (S.26)), sehr beschränkt ist.

Vergleicht man das Vererbungskonzept von SDL für die Sorten mit den Konzepten etablierter objektorientierter Sprachen ergeben sich weitere Probleme.

Polymorphieproblem: Von einer objektorientierten Sprache würde man erwarten, daß Datentypen mit Vererbungsbeziehung entsprechend geeigneter Regeln, zuweisungskompatibel sind. Bei Zuweisungen zwischen verschiedenen Datentypen ergibt sich sofort die Frage nach dem Verbleib der Komponenten des Originals, die nicht zur zugewiesenen Seite passen. Der einfache Fall ist die Konvertierung des Originals in den erwarteten Datentyp. Es ist aber auch sinnvoll die Zuweisung komplett durchzuführen. Nach einer derartigen Zuweisung gibt es einen Unterschied zwischen der statischen Datentypbeschreibung der zugewiesenen Seite und dem dynamisch vorhandenen Wert. Möglicherweise sind nicht alle Eigenschaften des dynamischen Werts durch die statische Beschränkung des Benutzungskontexts sichtbar. Derartige Zuweisungen werden, wie allgemein üblich, als polymorph bezeichnet.

Erst durch ein Polymorphiekonzept wird es möglich, Aktionen basierend auf dem dynamischen Datentyp eines Werts zu definieren. Diese dynamische Bindung von Aktionen wird als späte Bindung bezeichnet, die Definition mit den Aktionen ist virtuell. SDL besitzt zwar das Konzept virtueller Definitionen, aber nicht im Kontext von Datendefinitionen.

Konvertierungsproblem: In objektorientierten Programmiersprachen ist die Konvertierung eines Werts der Ableitung in einen Wert der Basisdefinition eine übliche, implizit definierte Operation. Die implizite Definition der linearen Abbildung in SDL dagegen entspricht der inversen Konvertierungsoperation für Werte der Basissorte in Werte der Ableitung. Diese gegensätzliche Semantik ist eines der grundlegenden Probleme, wenn es darum geht, in Programmiersprachen übliche Vererbungsstrukturen, z.B. das Hinzufügen von Feldern in einer Struktur, mit Sorten nachzubilden.

2.2.3 Algorithmische Operatoren

Neben der Berechnung von Operatoren, basierend auf Axiomen (axiomatische Operatoren), gibt es in SDL eine weitere syntaktische Notation, die es erlaubt, die Berechnung eines Operators algorithmisch zu beschreiben (algorithmische Operatoren).

Beispiel:
```
newtype Enum
    literals one, two;
    operators succ : Enum -> Enum;
    operator succ; fpar e Enum; returns Enum;
        start;
            decision e;
                (one) : return two;
                (two) : return one;
            enddecision;
    endoperator;
endnewtype;
```

Das semantische Modell für algorithmisch definierte Operatoren sind zustandslose Prozeduren mit Rückgabewert. Bei der Berechnung von Termen mit algorithmischen Operatoren werden die algorithmischen Operatoranwendungen rekursiv durch den Rückgabewert ersetzt. Erst Terme, die keine algorithmischen Operatoren enthalten, werden entsprechend des ACT ONE-Modells berechnet.

Bei der Definition algorithmischer Operatoren sind einige Besonderheiten zu beachten.

Zyklenproblem: Mit einem algorithmischen Operator kann ein unendlicher Zyklus innerhalb des Operators oder durch rekursives Aufrufen des Operators beschrieben werden.

```
operator loop; fpar i Integer; returns Integer;
        start;
    loop:   task i := i+1;
            decision i>0;
                (True): join loop;        /* innerer Zyklus */
                else:   return loop(i);   /* rekursiver Zyklus */
            enddecision;
endoperator;
```

Diese Operatordefinition ist aus Sicht der SDL-Semantik kein Problem, da algorithmische Operatoren wie Prozeduren interpretiert werden. Der Aufruf eines derartigen Operators ist natürlich ein gültiger Term im Sinne des ACT ONE-Modells. Allerdings kann die Termklasse für diese Operatoranwendungen nicht bestimmt werden, da die Interpretation nicht endet. Das widerspricht dem ACT ONE-Modell, in dem jeder Term einer Termklasse zugeordnet werden kann.

Konstruktorproblem: Das Ergebnis der Berechnung eines algorithmischen Operators ist letztendlich ein Term, der keine algorithmischen Operatoren enthält. Es muß also schon Terme basierend auf Literalen oder Operatoren geben. Algorithmische Operatoren sind also niemals Konstruktoren von Werten einer Sorte. Deshalb gibt es auch keine nichtleere Sorte, die nur aus algorithmischen Operatoren besteht.

Für die Nutzung algorithmischer Operatoren gibt es Einschränkungen.

Einschränkung von Axiomen: Ein Term eines Axioms darf keine algorithmischen Operatoren enthalten.

Wenn ein algorithmischer Operator für alle Argumente berechenbar ist, kann man formal eine axiomatische Darstellung durch eine Aufzählung aller möglichen Operatorrufe mit den entsprechenden Ergebnissen finden. Beispielsweise wäre eine derartige axiomatische Beschreibung des *succ*-Operators aus dem *Enum*-Beispiel am Anfang des Abschnitts

```
newtype Enum
    literals one, two;
    operators succ : Enum -> Enum;
    axioms
        succ(one) == two;
        succ(two) == one;
endnewtype;
```

Es gibt jedoch keine direkte sprachliche Unterstützung für dieses Modell, so daß eine Anwendung dieses Prinzips für unendlich große Wertebereiche nicht möglich ist.

Einschränkung der Vererbung: Algorithmische Operatoren werden nicht vererbt, d.h. im Transformationsmodell für die Vererbung von Sorten werden algorithmischen Operatoren nicht in die Ableitung kopiert. Eine explizite Redefinition algorithmischer Operatoren ist natürlich möglich.

2.2.4 Zuweisungen

Werte können durch Zuweisungen von Ausdrücken an Variablen gebunden werden. Die Bindung bleibt erhalten, bis eine neue Zuweisung an diese Variable erfolgt oder die Variable aufhört zu existieren. Auf keinen Fall kann der gebundene Wert ohne eine Neuzuweisung geändert werden.

Wird der Wert eines Terms mit einer Variable ermittelt, ist die Variable durch den gebundenen Wert, genauer durch den zugewiesenen Repräsentanten des Werts, zu ersetzen. Die Terme der Variablen werden „kopiert". Man spricht auch von einer Wertesemantik. Voraussetzung für die Berechnung eines Terms ist, daß sämtliche Variablen gebunden sind. Eine Verletzung dieser Voraussetzung führt zu einem dynamischen Fehler bei der Ausführung des SDL-Programms.

Das Konzept der Zuweisung ist nicht nur auf Variablen beschränkt. Auch bei der Bindung eines aktuellen Parameters an einen formal definierten Parameter bei

- dem Ruf eines Operators oder einer Prozedur,
- der Ausgabe von Signalen und
- der Erzeugung eines Prozesses

treten im erweiterten Sinne Zuweisungen auf.

Effizienzproblem: Aus programmtechnischer Sicht wird bei einer Zuweisung der errechnete Wert in den Speicher der Variablen geschrieben. Das ist bei komplexen Datenstrukturen, z.B. sehr langen Zeichenketten, ein nicht zu verachtender Laufzeitaufwand. Das Laufzeitverhalten läßt sich verbessern, wenn durch entsprechende Optimierungsverfahren bei einer Zuweisung nur Teile des Variablenspeichers modifiziert werden oder auch nur ein Verweis auf den errechneten Wert gespeichert wird. Allerdings ist der Nachweis, daß die SDL-Semantik nicht verletzt wurde, oft problematisch bzw. man akzeptiert sogar Abweichungen von der SDL-Semantik. Außerdem haben die Optimierungsalgorithmen nur eine beschränkte Sicht auf das SDL-Programm, so daß nicht alle möglichen Optimierungen erkannt werden.

Diese Situation ist für eine automatische Codegenerierung aus SDL sehr unbefriedigend.

2.2.5 Teilbereichstypen

SDL bietet ein einfaches Konzept für die Bildung von Teilbereichstypen, die *syntype*-Konstruktion. Die Werte des Teilbereichstyps können aufgezählt oder mit den Ordnungsrelationen bestimmt werden.

Beispiel: **syntype** Natural = Integer **constants** >= 0 **endsyntype**;

Das Transformationsmodell der statischen Semantik ersetzt jede Verwendung eines *syntype*-Bezeichners durch den Bezeichner der zugrundeliegenden Sorte. Demzufolge ist die Definition eines Teilbereichstyps keine neue Sorte. Bei jeder Zuweisung an eine Sorte, die eigentlich auf einer Teilbereichsdefinition basiert, wird der zuzuweisende Wert mit dem angegebenen Teilbereich verglichen. Diese *syntype*-Tests sind Bestandteil der dynamischen Semantik von SDL. Liegt der Wert außerhalb des Teilbereichs, so ist das weitere Verhalten des Systems undefiniert.

Problem mit Mehrdeutigkeiten bei Operatorsignaturen: Bei Operatorsignaturen führt die Ersetzung der *syntype*-Bezeichner in wenigen Fällen zu unerwarteten Mehrdeutigkeiten. Beispielsweise sind beide Signaturen

operators
op: Integer -> Integer;
op: Natural -> Natural;

identisch zu

op : Integer -> Integer;

d.h. es liegt eine Mehrfachdefinition (statischer Fehler) vor.

Problem bei verschachtelten *syntype*-Konstruktionen: Bei verschachtelten *syntype*-Konstruktionen gilt auf Grund der Ersetzungsregel für *syntype*-Bezeichner nur die jeweils letzte Einschränkung des Teilbereichs. Das Beispiel

syntype Intervall = Natural **constants** <10 **endsyntype**;

ist äquivalent zum Teilbereichstyp

syntype Intervall = Integer **constants** <10 **endsyntype**;

aber nicht wie erwartet zu

syntype Intervall = Integer **constants** 0:9 **endsyntype**;

Problem mit unterschiedlichen Implementierungen: SDL-Werkzeuge können den *syntype*-Test ignorieren oder sehr spezifisch reagieren. SDL selbst bietet keine Sprachmittel, derartige Fehler abzufangen.

2.2.6 Transformationsbasierte Ausdrücke

Um die Lesbarkeit von Ausdrücken einer SDL-Spezifikation zu verbessern, unterstützt die Sprachdefinition Präfix- und Infixoperatoren, Feld- und Indexzugriffe sowie strukturelle Initialisierungen. Für diese Notationen ist in der statischen Semantik von SDL eine Transformation auf Operatoranwendungen definiert.

Zur Beschreibung der Syntaxelemente wird auf die im Z.100 verwendete Notation für Syntaxregeln zurückgegriffen. Die syntaktische Darstellung eines Ausdrucks sei <expression>, die eines SDL-Namens <name> und die einer Variablen <variable>.

Präfixoperatoren: Ein Ausdruck der Form

{ **not** | **-** } <expression>

wird in die Operatoranwendung

{ **"not"** | **"-"** } (<expression>)

transformiert.

Infixoperatoren: Ein Ausdruck der Form

<left expression> <infix operator> <right expression>

wird in die Operatoranwendung

"<infix operator>**"** (<left expression>, <right expression>)

transformiert. Infixoperatoren sind

<infix operator> ::=
=> | **or** | **xor** | **and** | **in** | **/=** | **=** | **>** | **<** | **<=** | **>=** | **+** | **-** | **/** | ***** | **//** | **mod** | **rem**.

Feld- und Indexzugriffe: Man muß zwischen einem lesenden und schreibenden Zugriff unterscheiden. Lesende Zugriffe sind Ausdrücke der Form

<expression> **!** <name>
<index expression> **(**<expression> { **,**<expression> }* **)**

Diese werden in Operatoranwendungen der Form

<name>**Extract!** **(** <expression> **)**
Extract! **(** <index expression>**,** <expression> { **,**<expression> }* **)**

transformiert, wobei „<name>**Extract**!" und „**Extract!**" jeweils Namen sind.

Schreibende Zugriffe sind spezielle Zuweisungen

<variable> **!** <name> **:=** <expression>
<variable> **(**<expression> { **,**<expression> }* **)** **:=** <right expression>**.**

Diese werden in Zuweisungen der Form

<variable>**:=**<name>**Modify!** **(** <variable> **,** <expression> **)**
<variable>**:= Modify!** **(**<variable>**,** <expression>{**,**<expression>}*, <right expression>**)**

transformiert. Sind die Variablen wieder Feld- oder Indexzugriffe, ist die Transformation rekursiv anzuwenden. „<name>**Modify**!" und „**Modify**!" sind wieder Bezeichner von Operatoren. Die Verwendung der linksseitigen Variablen als Operatorargument nach der Transformation setzt voraus, daß diese Variable bereits initialisiert ist.

Strukturinitialisierung: Ein Ausdruck der Form

(. <expression> { **,**<expression> }* **.)**

wird in die Operatoranwendung

Make! **(** <expression> { **,**<expression> }* **)**

transformiert.

Die Verwendung des Zeichens '!' nach einem Namen bedeutet, daß dieser Namen nur innerhalb der Sortendefinition verwendet werden darf. Demzufolge kann der SDL-Nutzer beide Formen der Notation für Index- und Präfixoperatoren verwenden, jedoch für Feld-, Indexzugriffe und Strukturinitialisierungen nur die transformationsbasierte Form. Es ist durchaus möglich, für die Feld- und Indexzugriffe nur die lesenden oder schreibenden Varianten zu definieren.

Initialisierungsproblem: In Programmiersprachen können Felder oder einzelne Elemente komplexer Datenstrukturen initialisiert werden, ohne daß die Gesamtstruktur selbst initialisiert wurde. Mit Blick auf SDL als Spezifikationstechnik ist eine Gesamtinitialisierung zwar sicherer, bedeutet aber unter Umständen unnötigen Aufwand.

2.2.7 Strukturelle Beschreibungen

SDL unterstützt die Zusammenfassung von Sorten durch eine strukturähnliche Datentypbeschreibung (vgl. Z.100 Abschnitt 5.3.1.10 „*Structure sorts*“). In der statischen Semantik ist für diese Notation eine Transformation in eine SDL-Sorte definiert. Die Operatoren dieser Sorte basieren auf den Transformationsmodellen für Feldzuweisung, Feldzugriff und Initialisierung einer Datenstruktur.

Beispiel:

```
newtype SimpleStructure struct
    i  Integer;
    b  Boolean;
endnewtype;
```

Einschränkung der Sortenvererbung: Es ist syntaktisch unmöglich, die Notation für Vererbung und Strukturdefinition gleichzeitig zu verwenden. Man kann demzufolge durch Vererbung keine Felder hinzufügen.

2.2.8 Abstrakte Sorten

Eine abstrakte Sorte ist ein Spezialfall einer abstrakten Typdefinition. In SDL gibt es aus historischen Gründen zwei Arten von abstrakten Sorten mit ähnlichen Eigenschaften: Sorten mit Kontextparametern[1] und Generatoren[2].

Beispiel:

```
newtype AbstractStruct <newtype Elem; synonym undefined Elem>
    struct
        e Elem;
    default (. undefined .); /* jede Variable wird mit diesem Wert initialisiert */
endnewtype;

synonym NoValue Integer = -1;

newtype IntegerStruct inherits AbstractStruct<Integer,NoValue>
    operators all;
endnewtype;
```

Implementierungsproblem bei Kontextparametern: Die Flexibilität des SDL-Parametrisierungskonzepts ist nur mit sehr hohem Aufwand sowohl für Analyse als auch Codegenerierung implementierbar, weshalb viele SDL-Werkzeuge die Parametrisierung anfangs nur eingeschränkt oder überhaupt nicht unterstützten. Das verhindert bis heute eine breite Anwendung von abstrakten Sorten in Kombination mit Vererbung.

2.2.9 Spezifikation von Fehlern

Werte können auf der Grundlage eines Axioms, welches den speziellen Fehlerterm

Error!

1. Kontextparameter wurden als allgemeines Konzept für vererbbare Definitionen in SDL-92 eingeführt.
2. Generatoren sind als spezielles Datenkonzept seit SDL-88 Bestandteil der Sprache SDL. Alle bekannten Anwendungen des Generatorkonzepts sind durch Kontextparameter in Verbindung mit Vererbung ersetzbar. Aus Gründen der Rückwärtskompatibilität und ihrer Verwendung zur Definition der vordefinierten Daten sind sie weiterhin Bestandteil der Sprache SDL.

enthält als fehlerhaft gekennzeichnet werden. Da dieses Fehlerverhalten nicht durch die ACT ONE-Semantik abgedeckt ist, muß auf die formale Semantik, konkret den Anhang F3 zurückgegriffen werden, um dieses Konzept zu erklären.

Wird während des Ablaufs eines SDL-Programms festgestellt, daß ein zu berechnender Ausdruck Repräsentant einer derartig gekennzeichneten Termklasse so ist, dann ist das weitere Verhalten des SDL-Programms undefiniert.

Die wesentlich kompliziertere Frage ist die nach der axiomatischen Interpretation des Fehlerterms. Betrachtet man in der formalen Semantik von SDL lokale Interpretationsregeln zur Bildung von Termklassen, so gelten folgende zwei Regeln:

1. Der Fehlerterm wird bei der Termklassifizierung wie ein gewöhnlicher Term behandelt. Deshalb gibt es exakt eine Termklasse, die den Fehlerterm enthält. Diese wird auch Fehlerklasse genannt.
2. Ein Repräsentant der Fehlerklasse kann nicht an Variablen gebunden werden. Das schränkt die Verwendung von Axiomen für Repräsentanten der Fehlerklasse ein.

ACT ONE-Fehlerproblem: Wegen der Komplexität der Behandlung der Axiome und der fehlenden Werkzeugunterstützung zur Interpretation des Standards ist es nicht möglich, eine globale Aussage zur Anwendung dieser Regeln zu treffen. Es sind zwei Teilprobleme identifizierbar.

Eine einzige Fehlerklasse: Die Bindung aller Fehler an einen Wert kann bei semantisch unterschiedlichen Fehlern und ungünstig gewählten Axiomen zu unerwarteten Wertbindungen führen. Bisher sind derartige Spezifikationen jedoch nicht bekannt.

Ausschluß der Variablenbindung: Hier gibt es ein Reihenfolgeproblem. Der Ausschluß von Termen für die Variablenbindung erfolgt erst, wenn erkannt wurde, daß der Term zur Fehlerklasse gehört. Möglicherweise führt die Variablenbindung aber erst zu diesem Fehler.

Diese Fehlerproblematik ist der größte Kritikpunkt, wenn es innerhalb der ITU-T um die ACT-ONE-basierte Datensemantik von SDL geht. Es ist auch offen[1], ob die Fehlerbeschreibungen in den vordefinierten Datentypen nicht widersprüchlich sind. Praktische Realisierungen vordefinierter Daten verhalten sich bei Fehlern „wie erwartet“, da die Daten nicht streng axiomatisch, sondern intuitiv richtig implementiert werden.

2.2.10 Vordefinierte Sorten

Um zu vermeiden, daß die Definition von grundlegenden Daten stets neu und möglicherweise auch anders erfolgt, gibt es eine Zusammenfassung wichtiger Datentypen im SDL-Modul *Predefined*. Die vordefinierten Datentypen umfassen sowohl einfache Sorten wie *Integer* oder *Boolean* als auch abstrakte Sorten wie Zeichenketten (*String*), Mengen (*Powerset*) oder Assoziativspeicher (*Array*).

Eine wichtige Eigenschaft der entsprechenden Sorten ist, daß aus mathematischer Sicht unendliche Mengen auch mit dieser Abstraktion definiert sind. Beispielsweise ist die Sorte *Integer* nicht beschränkt.

Endlichkeitsproblem: Eine Implementierung der vordefinierten Daten zum Zweck der Ausführung von SDL-Beschreibungen wird sich im Normalfall an die in Programmiersprachen allgemein üblichen Beschränkungen der Wertebereiche halten. Das wird bei der Analyse von Spezifikationen oft nicht beachtet.

1. Wegen der nicht ganz klaren Fehlersemantik wird auch auf eine weitere Diskussion verzichtet.

2.3 Externe Datenbeschreibungen

SDL bietet auch die Möglichkeit, Datenbeschreibungen anderer Sprachen in eine Sortendefinition zu integrieren (vgl. Z.100 Abschnitt 5.4.6 „*External data*“). Im Unterschied zu den bisher vorgestellten Datenkonzepten besitzt dieses Konzept nur eine unvollständige Semantik.

Eine Codefragment einer anderen Sprache kann direkt in die Definition einer Sorte übernommen werden, wenn der Beginn und das Ende der externen Notation gekennzeichnet sind. Derartig gebildete Sorten werden externe Sorten genannt. Standardmäßig erfolgt die Kennzeichnung mit dem Schlüsselwortpaar **alternative/endalternative**[1] und einer Angabe zur Bezeichnung der externen Notation. Innerhalb der Kennzeichnung gelten nur die syntaktischen Regeln der externen Notation.

Beispiel:
```
newtype C_enum
    literals x, y, z;
    operators num : C_enum -> Integer;        /* beliebige Signaturen möglich */
    alternative C;          /* ab hier gilt nur C-Syntax */
        enum { x, y, z }
    endalternative;         /* ab hier gilt wieder SDL-Syntax */
endnewtype;
```

Auf Grundlage der Signatur einer externen Sorte kann man Terme bilden. Gültige Terme der Sorte *C_enum* sind *x*, *y* und *z*. Es ist aber offen, wie aus den Termen von *C_enum* Werte gebildet werden, da es keine Axiome gibt. Zusätzlich gibt es auf der Grundlage des *num*-Operators Terme von *Integer* (z.B. *num(x), num(y), num(z)*). Auch hier gilt, daß die Berechnung des Operators nicht Bestandteil der SDL-Semantik ist. Die Definition einer externen Sorte gibt also die statische Semantik der Sorte vor, läßt aber die dynamische Semantik offen.

Die Relation der externen Datenbeschreibung zu den gegebenen Signaturen ist nicht Bestandteil der SDL-Sprachdefinition. Im Beispiel ist diese Relation offensichtlich: die Namen der Literale entsprechen den Elementen der C-Aufzählung und der Operator *num* wird die in C übliche Vergabe von Nummern an die Elemente repräsentieren. Die Relation muß aber auch Eigenschaften bezüglich der Sichtbarkeit von SDL-Definition und externen Beschreibungen berücksichtigen. Betrachtet man das folgende Beispiel, werden einige Fälle deutlich:

```
newtype C_struct
    operators ...
    alternative C;              /* ab hier gilt nur C-Syntax */
        struct {
            C_enum   e_field;   /* C_enum ist nur eine Sortenname von oben */
            int      i_field;   /* Beziehung von int zur SDL-Sorte Integer unklar */
            PId      p_field;   /* Sichtbarkeit der Sorte PId unklar */
        }
    endalternative;             /* ab hier gilt wieder SDL-Syntax */
endnewtype;
```

Die Integration einer externen Notation in Sorten ist ein Indiz dafür, daß eine Relation mit SDL-Konstruktionen, die nicht direkt zu den Datentypen gehören, vom SDL-Sprachstandard nicht vorgesehen sind.

Probleme:

- Lexikalische Unterschiede zwischen SDL und der externen Sprache können zu komplizierten Regeln beim Auflösen von Bezeichnern führen. Beispielsweise unterscheidet SDL im Gegensatz zu vielen Sprachen nicht zwischen Groß- und Kleinschreibung der lexikalischen Einheiten.

1. Falls die Kennzeichnung für das Ende der externen Notation **endalternative** durch die externe Notation selbst verwendet wird, kann es durch ein beliebiges lexikalisches Wort von SDL ersetzt werden.

- Die Sichtbarkeitskonzepte der externen Notation können sehr verschieden von SDL sein.
- Die Angabe der Signaturen in externen Sorten ist notwendig, um Terme bilden zu können. Folgt man dem Transformationsmodell für SDL-Strukturen, so müßten im Beispiel *C_struct* für jedes Feld die entsprechenden Feldzugriffsoperatoren definiert werden. Zusammen mit der Initialisierung sind das 7 Operatoren, die aus der Sicht des Nutzers unleserlich und redundant sind. Das gilt für viele externe Sorten.
- Für einige externe Konstruktionen würde man auch eine Relation zu nicht datenspezifischen SDL-Konstruktionen bevorzugen. Beispielsweise könnten Kommunikationsaspekte der externen Notation auch auf SDL-Signale abgebildet werden.

2.4 Schlußfolgerung

Die herausgearbeiteten Einschränkungen und Probleme der vorangegangenen Abschnitte untermauern die Notwendigkeit, das Datenkonzept von SDL zu überarbeiten. Das ACT ONE-Konzept geeignet zu ersetzen, ist sicherlich möglich. Aber, ACT ONE hat auch unumstrittene Vorteile. Die präzise Beschreibung von Daten ist in der Vergangenheit vielfach ausgenutzt worden, obwohl eine automatische Überführung in eine Implementierung meist nicht möglich war. Das Module *Predefined* ist nur ein Beispiel für die Verwendung von ACT ONE-Konstruktionen, die letztendlich mit völlig anderen Mechanismen implementiert werden. Man weiß aber sehr genau, ob die Implementierung sich konform zur Spezifikation verhält und könnte das für kritische Anwendungen sogar beweisen. In diesem Sinne sind Beschreibungen im Stil von ACT ONE von Experten für Experten durchaus geeignet, Sprachkonstrukte semantisch zu fundieren. Diese Tatsache wird letztendlich bei SDL-Sprachverbesserungen in der vorliegenden Arbeit ausgenutzt.

Die folgenden Punkte sind ein sehr allgemeiner Katalog von Forderungen, die an Verbesserungen des Datenmodells gestellt werden können. Die konkreten Einschränkungen und Probleme der vorangegengenen Abschnitte lassen sich jeweils einem oder mehreren Punkten zuordnen. Aus der historischen Entwicklung von SDL ergibt sich zusätzlich die Randbedingung der Sprachstabilität, die sich in vielen der Punkte widerspiegelt. Verbesserungen am SDL-Datenmodell sind auf dieser Grundlage einer Bewertung zugänglich.

Anforderungskatalog für neue Datenkonzepte

1. Eine verbesserte Notation für Datentypen sollte sich harmonisch in die zentralen Strukturierungskonzepte Vererbung, Parametrisierung und Sichtbarkeit einpassen.

   Viele der momentan vorhandenen strukturellen Einschränkungen sind einfach nicht mehr akzeptabel. Dieses betrifft z.B., daß

   - algorithmische Operatoren nicht vererbt werden,
   - sich *struct*-Konstruktionen und Vererbung ausschließen oder
   - Operatoren sich nicht spät binden lassen.

2. Neue Datenkonstruktionen sollten eindeutig definiert werden, damit SDL nicht die Qualität einer formalen Sprache verliert.

   Falls die neuen Konstruktionen die bisherige Semantik nicht ändern, sollten vorhandene Schwächen zusätzlich abgestellt werden. Das betrifft beispielsweise die Probleme mit

   - den Fehlertermen,
   - der theoretischen Berechenbarkeit von Ausdrücken für Analyse und Codegenerierung,
   - der Bezeichnerauflösung.

3. Bereits vorhandene Datendefinitionen sollten ohne großen Aufwand auf die neu zu definierenden Konzepte abbildbar sein.

   Als Minimalvariante wäre eine Transformation vorhandener Sorten ohne axiomatische Operatoren denkbar. Eine Grundregel bei der Weiterentwicklung einer normierten Sprache ist, daß alte Spezifikationen wiederverwendbar sind.

4. Alle Bestandteile einer Datendefinition, die inhaltlich zu dieser Datendefinition gehören, insbesondere Operatoren, sollten in dieser Datendefinition und nur dort erfaßt werden.

   Wegen der strukturellen Einschränkungen von SDL findet man in Spezifikationen nicht zu vermeidende, schwer interpretierbare Konstruktionen zur Manipulation von Daten. Das betrifft beispielsweise

   - die Definition von leeren Sorten nur für die Bereitstellung von Operatoren für andere Sorten,
   - die Spaltung einer Datendefinition in viele Sorten, da verschachtelte Datentypdefinition nicht möglich sind oder
   - die Verwendung von Prozeduren anstelle von eigentlich besser passenden Operatoren.

5. Die Möglichkeiten zur strukturellen Beschreibung von Sorten sollten erweitert werden.

   Nimmt man sich ASN.1, eine Sprache, die standardisiert und erfolgreich mit SDL kombiniert wird, als Vorbild, dann ist klar, daß einiges zu verbessern ist. Einfache, nicht erweiterbare Strukturen und Aufzählungen (einfache Literaldefinitionen) als Datenstrukturkonzepte sind nicht mehr zeitgemäß.

6. Neue Konstruktionen sollten durch SDL-Werkzeuge implementierbar sein.

   Dieser Punkt betrifft sowohl die theoretische Implementierbarkeit der Werkzeuge als auch die praktische, d.h. letztendlich effiziente Umsetzung von Konzepten. SDL-Werkzeuge sollten

   - in der Lage sind, Fehler anzuzeigen und
   - ein akzeptables Antwortverhalten besitzen.

   Führt man SDL-Spezifikationen in einer realen Systemumgebung als Programm aus, dann wird zunehmend erwartet, daß das SDL-Laufzeitverhalten dem von handimplementierten Programmen nahe kommt. Die Effizienzprobleme bei Zuweisungen oder Initialisierungen laufen dieser Forderung entgegen.

7. Es sollte Sprachelemente geben, die Eigenschaften eines SDL-Programms als Teil einer realen Systemumgebung unterstützen.

   Die informale Einbettung anderer Sprachen mit externen Sorten löst dieses Problem nicht. Man benötigt beispielsweise Sprachelemente:

   - zur Verbesserung des Laufzeitverhaltens durch explizite Einflußnahme des SDL-Nutzers und
   - zum Datenaustausch zwischen einem SDL-Programmen und anderen, möglicherweise nicht in SDL vorliegenden Programmen.

Kapitel 3

Methoden für neue Datenkonzepte

Die im Kapitel 2 herausgearbeitete, eingeschränkte Nutzung von ACT ONE-basierten Datenbeschreibungen mit SDL-Werkzeugen führte in der Vergangenheit zu unterschiedlichen Vorgehensweisen, Daten alternativ zu beschreiben. Viele dieser Vorgehensweisen sind auf spezielle Anwendungen zugeschnitten. Sie unterscheiden sich bezüglich der syntaktischen Benutzung und der semantischen Fundierung der Datenbeschreibungen erheblich.

Dieses Kapitel hat das Ziel, eine Systematik bezüglich der Klassifizierung von Vorgehensweisen aufzustellen. Gleichzeitig werden die prinzipiellen Probleme der entsprechenden Ansätze diskutiert. Die Systematik erleichtert in späteren Kapiteln die Charakterisierung von vorhandenen und neu entwickelten Datenkonzepten erheblich. Die Motivation und die Kriterien der Klassifizierung sind in einem separaten Abschnitt den Abschnitten für die drei Grundmethoden vorangestellt.

Alle Änderungen des Datenkonzepts erfordern, sofern sie zu normieren sind, eine Aktualisierung der vorhandenen SDL-Sprachdefinition. Mit einer Einschätzung, welche Teile der Sprachdefinition betroffen wären und wie eine Aktualisierung aus technischer Sicht erfolgen kann, wird das Kapitel abgeschlossen.

3.1 Eine Systematik für neue Datenkonzepte

Für eine systematische Unterteilung von Methoden müssen Klassifizierungskriterien gefunden werden. Ein Nutzer wird zuerst auf die syntaktische Repräsentation einer Beschreibung achten, erst danach werden semantische Unterschiede wichtig. Auch ein SDL-Analysewerkzeug wird die Syntax vor der formalen Semantik von SDL bearbeiten. Deshalb wird mit der folgenden syntaktischen Grobklassifizierung (Bild 7) begonnen.

Externe Beschreibungen: Ein externes Programmfragment wird zur Beschreibung der Daten genutzt. Die Integration dieser Beschreibung in SDL soll ohne syntaktische Änderung von SDL möglich sein.

Erweiterung von SDL: Es werden Erweiterungen der Sprache SDL zugelassen, um Daten beschreiben zu können. Semantische Erweiterungen sind hier nicht ausgeschlossen.

Substitution des Datenkonzepts: Die vorhandenen Konzepte zur Beschreibung der Daten werden verworfen und durch andere Konzepte ersetzt. Diese Methode betrifft sowohl die Syntax als auch die Semantik von SDL.

Eine Verfeinerung dieser Klassifizierung und letztendlich auch eine Unterscheidung entsprechend der semantischen Fundierung werden in den nachfolgenden Abschnitten betrachtet. Einige der Unterteilungen basieren auf der Verwendung anderer Sprachen. Zur Verdeutlichung einiger Probleme mit diesen Sprachen werden die ganzen Zahlen, d.h. bezogen auf SDL die Sorte *Integer*, als Beispiel genutzt. Diese Wahl wurde auf Grund von zwei wesentlichen Eigenschaften getroffen:

- ganze Zahlen sind intuitiv verständlich und bei den verwendeten Sprachen vorhanden;

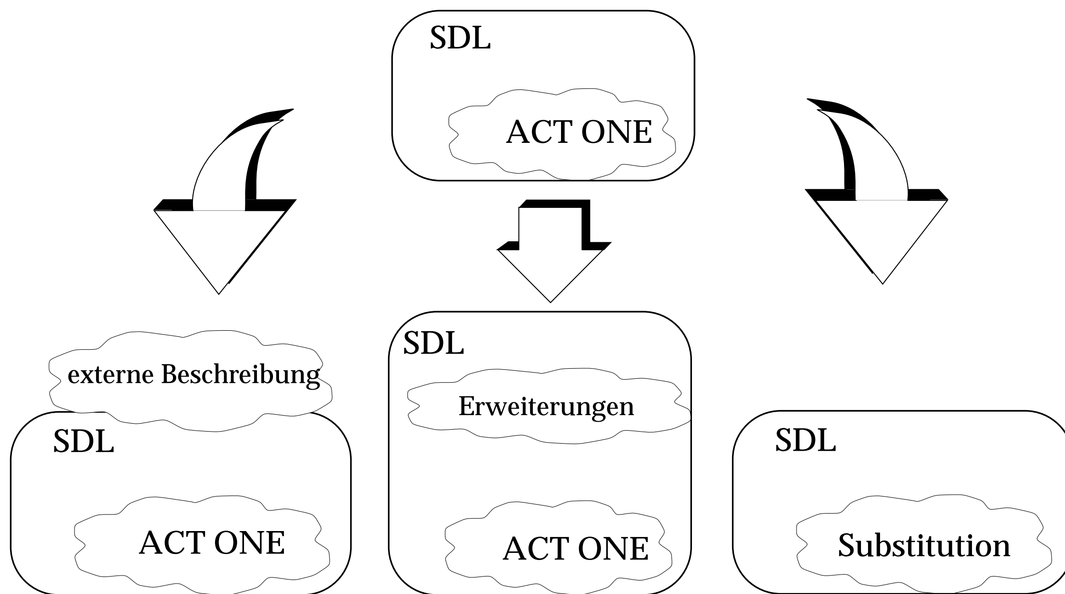

Bild 7: Veränderung der Datenkonzepte in SDL

- der Datentyp der ganzen Zahlen ist komplex genug, um die Unterschiede zwischen der abstrakten Sicht als unendliche Menge und der konkreten Implementierung in Programmiersprachen als endliche Menge zu verdeutlichen.

Viele Sprachen unterscheiden Konzepte wie Kommunikation, Datenbeschreibungen oder Ablaufkontrolle nicht durch syntaktische Konstruktionen. Deshalb ist es oft nicht möglich, insbesondere dann, wenn Datenbeschreibungen basierend auf anderen Sprachen betrachtet werden, sich im Kontext von SDL nur auf die Definition von Daten zu beschränken. Der Schwerpunkt der Betrachtung wird jedoch auf den Datenkonzepten liegen.

3.2 Externe Beschreibungen

Ausgangspunkt ist ein externes Programmfragment, welches in SDL so integriert werden soll, daß die SDL-Spezifikation syntaktisch korrekt bleibt. Für diese Integration ergeben sich drei weitere syntaktische Klassifizierungen.

Annotation einer SDL-Beschreibung: Das externe Programmfragment ist in einem SDL-Kommentar integriert.

Nutzung externer Sorten: Die Integration des Programmfragments erfolgt mit Hilfe einer externen Sorte (vgl. Abschnitt 2.3 „*Externe Datenbeschreibungen*“ (S.38)).

Transformation: Die externe Beschreibung wird in syntaktisch richtiges SDL überführt.

Für die externen Programmfragmente sind durch die genutzte Sprache eine statische und oft auch eine dynamische Semantik vorgegeben. Daraus ergibt sich das Problem, in welcher Relation die statische und dynamische Semantik der externen Fragmente mit der Semantik von SDL steht. Da nach einer syntaktischen Einteilung das Verhalten wichtig ist, wird die dynamische Semantik zur weiteren Unterteilung herangezogen. Die folgenden Punkte werden dabei für jede Methode, externe Fragmente in SDL zu integrieren, diskutiert:

vollständige Transformation: Das Verhalten der externen Notation wird durch ein geeignetes Transformationsmodell vollständig mit SDL erklärt.

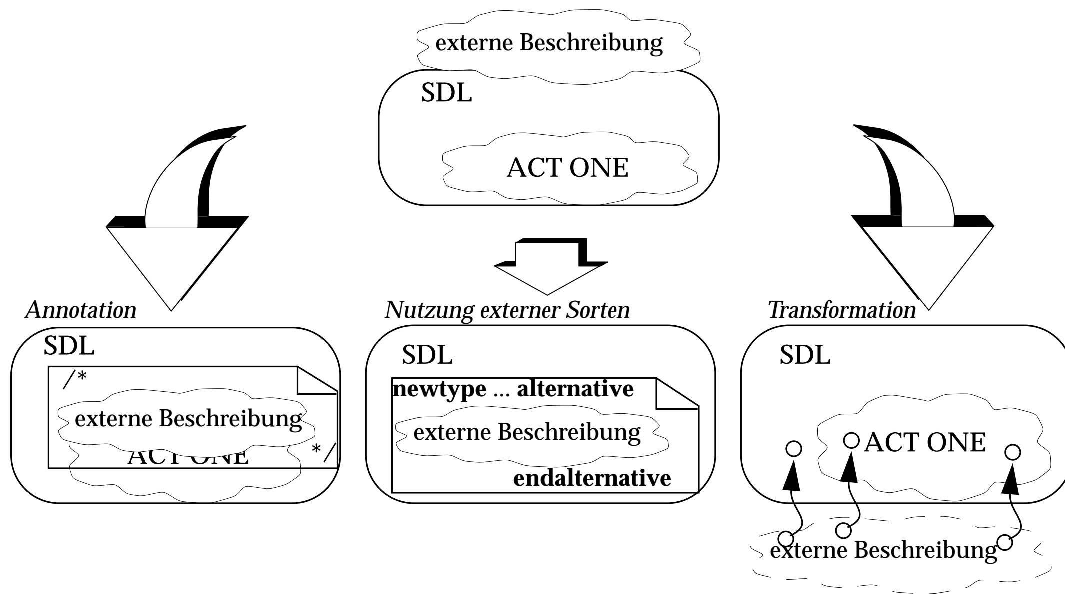

Bild 8: Integration externer Beschreibungen in SDL

Einbettung: Das Verhalten der externen Notation wird in SDL übernommen und ist nicht mit SDL erklärt.

3.2.1 Annotierte SDL-Beschreibungen

Ein Beispiel für ein annotiertes SDL-Fragment könnte die Benutzung eines C-*Integer*-Datentyps sein:

Beispiel: **dcl** i_var /*externe C-Notation: **int** */ int;

Der externe C-Datentyp *int* soll in SDL benutzbar sein.

Damit die Variablendeklaration korrekt ist, muß durch die semantische Relation zwischen C und SDL mindestens eine Sortendefinition mit dem Namen *int* bereitgestellt werden. Obwohl die SDL-Syntax nicht verändert wird, werden Werkzeuge, die diese Relation nicht kennen, statische Semantikfehler finden (im oberen Beispiel „Datentyp *int* nicht definiert“). Das läuft einer allgemeinen Verwendung derartiger Spezifikationen entgegen.

Der Umfang der Relation zwischen externem Fragment und SDL bestimmt auch, welche externe Funktionalität in SDL nutzbar ist. Beispielsweise ist es möglich, daß die C-*shift*-Operatoren "<<" und ">>" innerhalb von SDL nicht vorhanden sind.

Vollständige Transformation

Angenommen der Datentyp *int* ist definiert als

syntype int = Integer **endsyntype**;

Zusätzlich wird jede C-Bezeichnung **int** auf den *syntype*-Bezeichner *int* abgebildet. Das wäre ein vollständiges Transformationsmodell, jede Verwendung des int-Datentyps ist mit SDL semantisch fundiert. Betrachtet man jetzt die Zuweisung

task i_var := 1000000000000000000000000000;

so wird aus SDL-Sicht der richtige Wert zugewiesen. Die gleiche Zuweisung in C mit der Notation

task 'externe C-Notation: i_var = 1000000000000000000000000000;';

wird potentiell[1] mit einer Fehlermeldung „Konstante nicht im Wertebereich“ von einem C-Compiler abgewiesen werden. Es gibt also Unterschiede in der statischen Korrektheit. Beim Überlauf der Operationen "+" oder "*" treten sogar dynamische Unterschiede auf. Es liegt also ein Konflikt zwischen der C- und der SDL-Semantik vor.

Man könnte jetzt auf dem Standpunkt stehen, daß der C-Ausdruck durch die Transformation in einen SDL-Ausdruck übersetzt wird und die SDL-Semantik grundsätzlich gilt. Aber: die Ausführung eines SDL-Systems erfolgt letztendlich auf der Grundlage einer Programmiersprache. Insbesondere die annotierten Beschreibungen werden sicher entsprechend ihrer originalen Semantik genutzt. Das führt zu einem anderen Verhalten der Implementierung gegenüber beispielsweise einer simulativen, SDL-konformen Abarbeitung der Spezifikation.

Einbettung

Verändert man die Definition des Datentyps *int* in die Beschreibung gemäß Bild 9, dann wäre

```
newtype int
    literals    nameclass ('1':'9')('0':'9')5 or '0';
    operators
        "+": int, int -> int;
        "*": int, int -> int;
        /* ... und andere Operationen, die benötigt werden */
    alternative C;
        int
    endalternative;
endnewtype;
```

Bild 9: Definition der externen Sorte int

dieser Datentyp zwar im Sinne von SDL korrekt spezifiziert, jedoch wäre das Verhalten der Operatoren "+" und "*" nicht mit SDL festgelegt. Offensichtlich gibt es aber eine dynamische Semantik für diese Operatoren: genau die in C definierte Semantik. Das heißt, immer wenn entsprechend der SDL-Semantik ein Ausdruck berechnet werden soll, muß die Berechnung mit Hilfe der gegebenen C-Semantik erfolgen. In diesem Sinne ist der Begriff der Einbettung der externen Semantik gerechtfertigt.

Auch für diese Definition von *int* besteht das Problem mit der Literaldarstellung. Die demonstrierte Einschränkung auf sechs Ziffern verhindert zwar die Verwendung zu großer Literale in SDL, allerdings sind damit nicht alle Zahlen nach SDL abbildbar, die in C möglich sind. Allgemein ist es schwierig, derartige Kompromisse zu vermeiden.

Bei der Einbettung entsteht ein anderes Problem mit der Semantik als bei der Transformation. Die fehlende Zuweisungskompatibilität zwischen der vordefinierten Sorte *Integer* und der externen Sorte *int* führt zu statischen Fehlern bei „gemischten“ Ausdrücken:

task i_var := length('String der Laenge 20');

Fehler: kein Operator "length: Charstring -> int;" definiert.

1. Der Wertebereich des C-Datentyps *int* ist nicht exakt festgelegt.

Man kann natürlich die Sorte *int* um einen entsprechenden Operator erweitern, die Definition müßte in SDL-üblicher Notation, d.h. algorithmisch oder algebraisch angegeben werden. Auf diese Weise erhält man eine Sorte mit einer Kombination aus externen und internen Operatoren.

Ersetzt man alle externen C-Operatoren durch entsprechende SDL-Definitionen, ergibt sich wieder eine vollständige Transformation, die die Eigenschaften des externen Fragments möglicherweise besser widerspiegelt als die *syntype*-Konstruktion.

Aus praktischer Sicht ist es ungünstig zwei Datentypen (hier *int* und *Integer*) in einer Sprache zu unterstützen, man wird auf eine Art der Notation durchgängig verzichten oder semantisch nicht zwischen den Notationen unterscheiden. Eine entsprechende SDL-Werkzeugunterstützung kann i. Allg. nicht erwartet werden.

3.2.2 Externe Sorten

Die Beschreibung aus Bild 9 kann auch direkt in einer SDL-Spezifikation verwendet werden, um externe Datenbeschreibungen in SDL zu integrieren. Die statischen Eigenschaften der Sorte *int* sind damit festgelegt. Im Gegensatz zum vorigen Abschnitt ist diese Sortenbeschreibung jedoch nicht das Ergebnis einer semantischen Transformation, so daß die Zuordnung der SDL-Signaturen zu den externen Sprachkonstruktionen durch diese Transformation nicht gegeben ist. Mit der Definition einer derartigen Zuordnung wird das dynamische Verhalten der SDL-Operatoren auf das dynamische Verhalten der externen Konstruktionen abgebildet, d.h. es liegt analog zum vorigen Abschnitt eine Einbettung vor. Eine vollständige Transformation von externen Sorten in SDL-Sorten ist nicht sinnvoll, da in diesem Fall für das externe Fragment eine vollständige Transformation ohne dem Umweg über die externe Sorte angegeben werden kann. Die Vervollständigung der externen Sortendefinitionen mit SDL-Operatoren ist wieder denkbar.

Insgesamt ist die Anwendung externer Sorten eine semantisch verbesserte Notation externer Beschreibungen. Ein SDL-Werkzeug kann zumindest die statische Semantik einer Spezifikation testen, ohne die konkrete Abbildung der externen Konstruktionen auf SDL zu kennen. Es ist möglich, daß nicht alle Eigenschaften der externen Beschreibungen durch SDL-Konstruktionen widergespiegelt werden.

Die konzeptionellen Probleme aus Abschnitt 2.3 „*Externe Datenbeschreibungen*" (S.38) sind zu beachten und schränken die Nutzung externer Sorten ein:

- Lexikalische, syntaktische und semantische Unterschiede erschweren die Zuordnung von Namen.
- Redundante Signaturen für sich wiederholende strukturelle Beschreibungen (Signaturmuster für C-Strukturen) werden mit Sicherheit nicht von Hand spezifiziert. Das impliziert wieder eine spezielle Werkzeuglösung.

3.2.3 Transformation

Man kann natürlich die Abbildung einer externen Beschreibung nach SDL durch eine explizite, werkzeuggestützte Übersetzung nach SDL vorgeben.

Das Ergebnis dieser Transformation kann eine semantisch korrekte SDL-Beschreibung sein, d.h. die ausgeführte Transformation ist vollständig. Das hat den Vorteil, daß die SDL-Beschreibung von verschiedenen standardkonformen SDL-Werkzeugen stets gleich interpretiert wird. Damit stellt sich aber gleichzeitig die Frage nach der semantischen Relation zwischen den transformierten SDL-Fragmenten zum Original.

Es ist aber auch denkbar, daß nur eine externe Sorte oder unvollständige Beschreibungen mit Annotationen erzeugt werden. In diesem Fall wird das Verhalten der erzeugten SDL-Konstrukte wieder mit Einbettung erklärt. Die Probleme mit externen Sorten oder Annotationen aus den vorigen Unterabschnitten bleiben unverändert.

3.3 Erweiterung von SDL

Die SDL-Erweiterungen unterscheiden sich hinsichtlich zweier Hauptanwendungen.

Sprachspezifische Erweiterungen: Die Erweiterungen folgen den Konzepten einer vorhandenen Sprache. Idealerweise gibt es keine wesentlichen Unterschiede zwischen den neuen SDL-Konzepten und den Konzepten der anderen Sprache.

SDL-Erweiterungen: Es werden Sprachkonzepte speziell für SDL entwickelt und zur Sprache hinzugefügt.

Zur Einführung einer Syntax für die neuen Konzepte, gehört auch eine semantische Fundierung. Die Angabe eines geeigneten Transformationsmodells ist in vielen Fällen ausreichend. Es gibt aber auch Konzepte, bei denen die Einführung eines Transformationsmodells nicht möglich ist und neue Kernkonzepte in die SDL-Semantik aufgenommen werden müssen. Die Einführung einer Ausnahmebehandlung in [LöSc98a] ist ein Beispiel für ein derartiges Kernkonzept.

3.3.1 Sprachspezifische Erweiterungen

Sprachspezifische Erweiterungen können die Nutzung externer Beschreibungen mit SDL wesentlich vereinfachen.

Beispiel: **dcl** i_var int;

SDL wird so erweitert, daß die Sorte *int* existiert und sich semantisch wie der C-Datentyp verhält. Zusätzlich gibt es zwei Operatoren:

operators
int : Integer -> int;
Integer : int -> Integer;

die eine Konvertierung zwischen der Sorte *int* und der vordefinierten Sorte *Integer* realisieren. Das implizite Einfügen der Konvertierungsoperatoren an den aus Sicht der Zuweisungsregeln notwendigen Stellen wäre eine zusätzliche Variante der statischen Semantik. Damit wäre die Aktion

task i_var := length('String der Laenge 20‘);

aus dem Unterabschnitt „*Einbettung*” (S.44) legal.

Basieren alle Erweiterungen auf einem Transformationsmodell, so gelten alle Eigenschaften, die schon im Abschnitt 3.2.1 „*Annotierte SDL-Beschreibungen*” (S.43) besprochen wurden. Führt man ein neues Kernkonzept ein, so wird diese Definition dem Konzept der externen Sprache folgen.

Die Problematik von sprachspezifischen Erweiterungen besteht weitestgehend in der Kompatibilität zur standardisierten Sprachdefinition von SDL und den semantischen Unterschieden zu den eigentlichen externen Sprachkonzepten.

3.3.2 Sprachunabhängige Erweiterungen

Die Einführung neuer Konzepte unterliegt nur der Beschränkung, daß es keine Konflikte mit vorhandenen Sprachkonzepten gibt. Semantisch sind sowohl die Einführung neuer Kernkonzepte als auch Transformationsmodelle möglich.

Neben dem Kompatibilitätsproblem ist es ein erheblicher Aufwand, neue Konzepte zu entwickeln und diese auch formal zu beschreiben.

3.4 Substitution des Datenkonzepts

Obwohl eigentlich alle Dokumente des Sprachstandards bei einer Veränderung des Datenkonzepts betroffen sind, kann durch die Gliederung der Sprachdefinition das Datenkonzept relativ gut isoliert werden. Eine Ersetzung ist also durchaus ein realistischer Ansatz für den es prinzipiell zwei Varianten gibt.

Sprachspezifische Ersetzung: Das Datenmodell einer vorhandenen Sprache wird mit möglichst wenigen Änderungen übernommen.

Neuentwurf: SDL erhält ein auf die Sprache zugeschnittenes, neues Datenmodell.

Rückwärtskompatibilität ist eine essentielle Forderung bei der SDL-Sprachentwicklung. Deshalb muß jede Ersetzung des Datenmodells zusätzlich klären, wie vorhandene Spezifikationen mit der neuen Sprache zusammenarbeiten.

3.4.1 Sprachspezifische Ersetzung

Die Wahl einer Sprache als Vorbild für das Datenkonzept vereinfacht die Definition der Semantik erheblich, da diese in den wesentlichen Punkten vorhanden ist. Selbst Programmiersprachen sind konzeptionell stabil, einige sind sogar standardisiert, d.h. man kann davon ausgehen, daß die Semantik der geläufigen Programmiersprachen eindeutig ist. Auch bestehen Programmiersprachen hauptsächlich aus Konstruktionen, die man in SDL zur Definition von Daten benutzen kann. Orientiert man sich an einer alternativen Sprache, gibt es folgende Probleme:

- Sowohl die alternativen Sprachen als auch SDL besitzen oft ähnliche Konzepte, die nicht nur auf die Daten beschränkt sind. Betrachtet man C++, so wird das Problem deutlich:
  - C++ bietet das *template*-Konzept, SDL die Kontextparameter bzw. für Datentypen sogar Generatoren. Hier muß eine sinnvolle Vereinigung gefunden werden.
  - Das Präprozessorkonzept von C++ kann nur partiell mit SDL-Konzepten (z.B. *macro*-Notationen) nachgebildet werden. Es muß geklärt werden, ob es für den Nutzer von SDL akzeptabel ist, beispielsweise die umfangreiche *macro*-Ersetzung oder den *include*-Mechanismus von C++, nicht wiederzufinden.

  Der Neuentwurf entsprechender syntaktischer und semantischer Regeln für diese Fälle ist zwingend und sicher aufwendig.
- Die meisten Sprachen bieten nur Zusätze, die eine Ausführung von Programmen in verteilten Umgebungen ermöglichen, während in SDL Prozeß- und Kommunikationskonzepte integriert sind. Deshalb entstehen viele semantische Probleme, Datenkonzepte von Programmiersprachen in SDL zu übernehmen. Beispielsweise ist nicht intuitiv klar, wie Referenzen über SDL-Prozeßgrenzen hinaus funktionieren.
- Datentypen von Programmiersprachen sind i. Allg. endlich, um das Ablegen von Werten im Speicher zu ermöglichen. SDL kennt diese Einschränkung als abstrakte Spezifikationstechnik nicht. Deshalb verhalten sich einige Basisoperationen anders als bei Programmiersprachen. Der Operator "+" für ganze Zahlen läuft im jetzigen SDL-Datenmodell nicht über. Bei Gleitkommazahlen verschärft sich das Problem, da in SDL exakt mathematisch gerechnet wird und Programmiersprachen grundsätzlich auf eine endliche Speicherrepräsentation runden.

Auf Grund der Kompromisse, die man als Sprachentwickler wegen dieser Probleme eingehen muß, gibt es keine bekannte praktische Anwendung dieses Ersetzungsmodells.

3.4.2 Neuentwurf

Der Neuentwurf eines SDL-Datenmodells bietet die größten konzeptionellen Freiheiten. Allerdings sind Neuentwicklungen oft aufwendig, weshalb Sprachentwickler sich durchaus an vorhandenen Sprachen orientieren. In diesem Sinne ist die Unterscheidung zwischen sprachspezifischer Ersetzung und Neuentwurf fließend.

Ein Neuentwurf wird eine sehr lange Phase benötigen, ehe alle Probleme und Fehler beseitigt sind und SDL-Werkzeuge die neuen Konzepte entsprechend stabil unterstützen.

3.5 Veränderung der normativen Sprachdefinition

Für die Integration externer Datenbeschreibungen und eingeschränkt auch für Erweiterungen wird man eine Beschreibung benötigen, die die Relationen der alternativen Notation zu SDL klärt. In der Regel wird eine nicht normative Dokumentation des entsprechend genutzten SDL-Werkzeugs diese Relationen eingeschränkt und insbesondere informal beschreiben.

Die Änderung des Sprachstandards selbst ist mit einem erheblichen Aufwand verbunden,

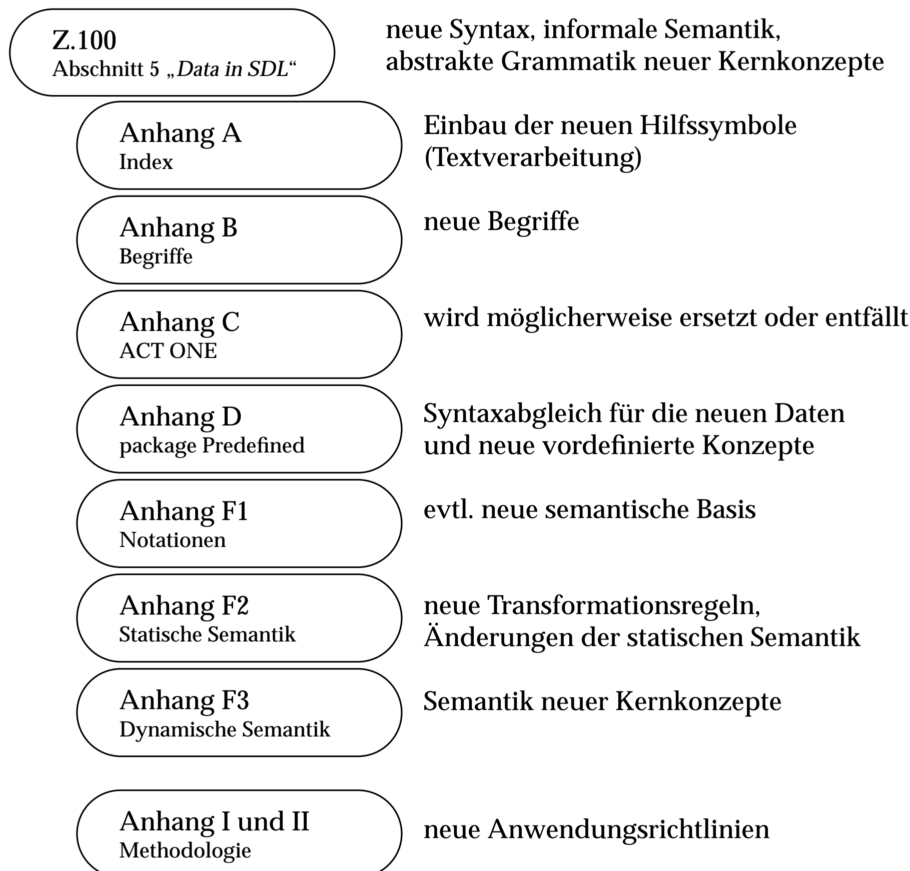

Bild 10: Notwendige Änderungen am Sprachstandard für ein neues Datenkonzept

Bild 10 listet die betroffenen Abschnitte auf. Änderungen der formalen Semantik in den F-Anhängen sind praktisch nicht möglich, da es keine elektronisch zu verarbeitenden Dokumente gibt. Die Beschreibung der Semantik neuer Konzepte muß also alternativ erfolgen. Bei der Vorstellung eines neuen Konzepts innerhalb der ITU-T werden i. Allg. zwei Dokumente verfaßt:

1. eine gute Erklärung des Konzepts und
2. wenn das Konzept akzeptiert wurde, eine systematische Änderungsliste gegenüber dem Hauptdokument Z.100.

Wegen der vielen Interessen der Sprachentwickler sollte das erklärende Dokument zu Beginn Alternativen enthalten, die in mehreren Iterationen der Entscheidungsfindung entfernt werden. Bei der Einführung von neuen Datenkonzepten in dieser Arbeit geht es ausschließlich um die gute Erklärung, die möglichst viele Detailentscheidungen offen lassen. Es besteht insbesondere kein Anspruch auf eine formale Sprachentwicklung.

Kapitel 4

Realisierte alternative Datenkonzepte

Der Einsatz von SDL zur werkzeuggestützten Implementierung von Softwarekomponenten gewinnt zunehmend an Bedeutung. Die im Kapitel 2 herausgearbeiten Mängel am Datenkonzept schränken die Nutzung von SDL als Implementierungssprache erheblich ein, weshalb verschiedenste Verbesserungsansätze von Werkzeugen bereitgestellt werden. Diese basieren oft auf der Verwendung alternativer Sprachen. Als Kandidaten für alternative Sprachen kommen sowohl Programmiersprachen als auch abstrakte Beschreibungstechniken wie ASN.1 [X.208], [X.680] oder IDL [OMG98] zum Einsatz.

In den folgenden Abschnitten werden konkrete, praktisch genutzte Verbesserungsansätze zur Integration alternativer Notationen in SDL vorgestellt und auf der Grundlage des Forderungskatalogs aus Abschnitt „*Anforderungskatalog für neue Datenkonzepte*" (S.39) bewertet. Begonnen wird mit der Integration von Programmiersprachen, die für Zielcodegenerierung aus SDL von Bedeutung sind. Die von Programmiersprachen unabhängige Kombination von SDL mit ASN.1 wird in einem separaten Abschnitt wegen der besonderen Bedeutung in der Standardisierung von SDL und der Realisierung mit SITE betrachtet. Der letzte Abschnitt dieses Kapitels umreißt die Kombination von SDL mit IDL bzw. der im Kontext von SDL von der ITU-T standardisierten Variante ODL.

4.1 Verwendung von Programmiersprachen

Die Generierung von Zielcode aus einer SDL-Spezifikation gehört zu den wichtigsten Aufgaben vieler SDL-Werkzeuge. I. Allg. wird dem Nutzer zusätzlich die Möglichkeit gegeben, eigene Funktionalität an SDL-Konstruktionen zu annotieren oder ganze Programmfragmente in der jeweiligen Zielsprache bereitzustellen. Diese alternativen Konstrukte werden bei der Generierung von Zielcode direkt übernommen. Derartig angereicherte Spezifikationen sind also nur spezifisch für die entsprechende Zielplattform bzw. -sprache nutzbar.

Die Zuordnung verschiedener technischer Realisierungen von Annotationen bzw. der Integration ganzer Programmfragmente in die allgemeinen Verbesserungsmethoden aus Kapitel 3 ist Inhalt dieses Abschnitts. Mit den bereits in Kapitel 3 herausgearbeiteten Konsequenzen für die allgemeinen Methoden, kann eine abschließende Bewertung des Einsatzes von Programmiersprachen in SDL vorgenommen werden.

4.1.1 Annotationen

Die Angabe von semantisch wichtigen Informationen für die Codegenerierung in Kommentaren, also die Annotation der Beschreibung mit Einbettungssemantik, ist bei SDL-Werkzeugen eine sehr verbreitete Methode[1]. Derartige Informationen können die gesamte, nicht mit SDL beschriebene Implementierung enthalten oder nur Schnittstellen zu externen Softwarekomponenten beschreiben.

Beispiel:

```
newtype Boolean
  literals
    /*$codegen: SDLBool::SDLTrue() */ True,
    /*$codegen: SDLBool::SDLFalse() */ False;
  operators
    /*$codegen: ASSERT #S */ assert: Boolean -> Boolean;
endnewtype Boolean;
```

Diese, in den SITE-Werkzeugen realisierte Variante, ermöglicht dem Nutzer die Verwendung von C++-Namen externer Klassen und Klassenmethoden bzw. Funktionen mit entsprechend vorgegebenen Aufrufkonventionen. So wird mit dem SDL-nach-C++-Codegenerator der SDL-Ausdruck *True* immer in den C++-Term *SDLBool::SDLTrue()* und ein Aufruf von *assert* stets in einen statischen Funktionsruf mit dem Namens *ASSERT* übersetzt.

Annotationen werden in der Regel nur für die Codegenerierung von SDL in konkrete Programmiersprachen eingesetzt.

4.1.2 Programmfragmente

Die Integration ganzer Programmfragmente in die SDL-Spezifikation ist ebenfalls verbreitet. Syntaktisch gibt es dazu mehrere gebräuchliche Varianten.

Externe Sorte: Die Verwendung externer Sorten bietet sich für alternative Notationen direkt an. Da die Abbildung der Signatur auf die alternative Notation nicht standardisiert ist, werden Annotationen oder werkzeugspezifische, implizite Abbildungsregeln verwendet.

```
newtype ExternalOperatorContainer
  operators
    /*$codegen: fib#S */ Fib: Integer-> Integer;
   alternative Cplusplus;
      int fib(int i) { return i<=1 ? 1 : fib(i-1)+fib(i-2); }
   endalternative;
endnewtype;
```

In SDL kann man mit dieser Notation den Datentyp *ExternalOperatorContainer* mit dem Operator *Fib* nutzen. Die vom SDL-nach-C++-Generator genutzte C++-Schreibweise des Operators ist aber *fib*. Die in der Sorte vorgegebene Implementierung wird in den generierten Code übernommen. Für jede Anwendung des SDL-Operators *Fib* in SDL-Ausdrücken können die statische Korrektheit getestet und entsprechender C++-Code für den Aufruf generiert werden.

Diese mit den SITE-Werkzeugen realisierte Nutzung externer Sorten mit Einbettungssemantik, ist eine Vorgehensweise, einzelne Operatoren effizient zu implementieren oder Schnittstellen zu nutzerdefinierten Konstruktionen bereitzustellen.

---

1. Annotationen sind in der Regel bei SDL-Werkzeugen nicht auf Datentypen beschränkt.

Syntaxerweiterung: Mit Hilfe syntaktischer Erweiterungen können alternative Notation direkt in SDL verwendet werden.

```
package User_extensions;
    cplusplustype
        int Fib(int i) { return i<=1 ? 1 : Fib(i-1)+Fib(i-2); }
    endcplusplustype;
endpackage;
```

Der Nutzer und natürlich auch das SDL-Werkzeug müssen das der sprachspezifischen Syntaxerweiterung zugrundeliegende Transformationsmodell kennen.

Spezifikationen mit Syntaxerweiterungen sind deutlich besser lesbar als solche mit Annotationen oder externen Sorten. Allerdings ist die Verarbeitung von Spezifikationen mit Spracherweiterungen an die Existenz spezifischer Werkzeuge gebunden, womit der Austausch von Spezifikationen unterlaufen wird.

Das obere Beispiel ist eine Abwandlung des Ansatzes zur Integration von ASN.1 in SDL aus Abschnitt 4.2.3 „*ASN.1-Einbettung mit SITE*" (S.61).

Transformation: Mit Hilfe einer Übersetzung werden aus externen Konstruktionen SDL-Definitionen mit vollständiger Information für die statische Semantikanalyse generiert (Transformation mit Einbettungssemantik). In Bezug auf Datentypen sind das beispielsweise annotierte oder externe Sorten bzw. auch Sorten mit Signaturen aber ohne Axiome, denen man die externe Herkunft nicht mehr ansieht. Vorzugsweise werden die generierten Konstruktionen in separaten Modulen zusammengefaßt.

Angenommen die Datei *UserData.h* enthält:

```
enum E { one(1), two(2) }
```

Daraus könnte folgendes SDL-Fragment generiert werden:

```
package UserData;
    newtype E
        literals one, two;
        operators num : E -> Integer;
    endnewtype;
endpackage;
```

Diese Methode, angewendet auf die Integration von C-*header*-Dateien, wurde in [HKV97] vorgestellt und ist im kommerziellen SDL-Werkzeug SDT [Tel98] implementiert.

4.1.3 Konsequenzen

Werkzeuge, die eine SDL-Spezifikation analysieren[1], ignorieren oft alternativen Beschreibungen, so daß falsche Analyseergebnisse geliefert werden. Alle Varianten mit Programmiersprachen sind semantisch problematisch, da das Verhalten der eingebetteten Konstruktionen nicht der SDL-Semantik unterliegt. Besonders fehleranfällig ist hierbei die Nutzung von Zeigerstrukturen. Werden die alternativen Codefragmente der SDL-Spezifikation verborgen, z.B. durch die Überführung externer Fragmente in syntaktisch korrekte aber nur transformierte SDL-Module, so sind Fehler bei der Interpretation der Spezifikation sehr wahrscheinlich.

Vergleicht man die Anforderungen aus Abschnitt „*Anforderungskatalog für neue Datenkonzepte*" (S.39) mit der Möglichkeit, alternative Programmfragmente in SDL zu integrieren, so ergibt sich folgendes Bild:

1. Syntax, Semantik, Nachweis von Verhaltenseigenschaften, etc.

1. *Eine verbesserte Notation für Datentypen sollte sich harmonisch in die zentralen Strukturierungskonzepte Vererbung, Parametrisierung und Sichtbarkeit einpassen.*
   Oft lassen sich Konstruktionen, die Schnittstellen zu externem Code repräsentieren, nicht weiter ausbauen. Vererbung oder die Nutzung als aktuelle Kontextparameter ist in der Regel für Definitionen, basierend auf externem Code, nicht möglich. Deshalb ist dieser Punkt der Anforderungsliste nicht erfüllt. Jedoch werden derartige Schnittstellen in der Praxis nur mit sehr geringen strukturellen Anforderungen („einfach“) in der Spezifikation verwendet, so daß dieser Nachteil nicht auffällt.
2. *Neue Datenkonstruktionen sollten eindeutig definiert werden, damit SDL nicht die Qualität einer formalen Sprache verliert.*
   SDL und in der Regel auch die jeweilige alternative Notation erfüllen diese Forderung. Problematisch ist jedoch die Kombination der Sprachen. Eine Realisierung der alternativen Sprachanbindungen durch SDL-Werkzeuge kann sehr unterschiedlich ausfallen. Oft, z.B. bei der Verwendung von C oder C++ als alternative Sprache, ist die Verwendung von Referenzen möglich, erfolgt jedoch auf eigene Gefahr. Der Nutzer muß mit dem internen Kontrollfluß der asynchronen Prozeßausführung vertraut sein, um Speicher zum richtigen Zeitpunkt freizugeben. Das zugrundeliegende Problem ist die oft nicht von den Werkzeugherstellern angegebene Semantik der Kombination beider Sprachen.
3. *Bereits vorhandene Datendefinitionen sollten ohne großen Aufwand auf die neu zu definierenden Konzepte abbildbar sein.*
   Die Möglichkeiten, vorhandene Datendefinitionen zu nutzen, wird von zusätzlichen alternativen Sprachfragmenten nicht berührt. Es gibt bei diesen Einbettungen jedoch immer die Frage nach der Relation konzeptionell ähnlicher Strukturen (das Verhältnis der Datentypen C *int* zu SDL *Integer*). Nutzt man sogar zwei verschiedene externe Beschreibungsformen, beispielsweise C++ und Java-Fragmente, mit SDL als Bindeglied, muß diese Relation klar definiert sein.
4. *Alle Bestandteile einer Datendefinition, die inhaltlich zu dieser Datendefinition gehören, insbesondere Operatoren, sollten in dieser Datendefinition und nur dort erfaßt werden.*
   Bezüglich SDL gibt es keine Änderungen. Für die Integration einer alternativer Notation könnten durchaus syntaktische Varianten von den einzelnen SDL-Werkzeugen realisiert werden, die der Forderung entgegenkommen.
5. *Die Möglichkeiten zur strukturellen Beschreibung von Sorten sollten erweitert werden.*
   Dem Nutzer stehen zusätzlich alle Möglichkeiten der alternativen Notation zur Verfügung.
6. *Neue Konstruktionen sollten durch SDL-Werkzeuge implementierbar sein.*
   Die Verwendung von Codefragmenten der Zielsprache wird von vielen SDL-Anwendern gefordert und von allen geläufigen SDL-Werkzeugen unterstützt. Die Kompatibilität der Werkzeuge bezüglich der Integration alternativer Programmiersprachen ist wegen unterschiedlicher Implementierungsstrategien nicht gegeben.
7. *Es sollte Sprachelemente geben, die Eigenschaften eines SDL-Programms als Teil einer realen Systemumgebung unterstützen.*
   Durch alternative Programmierung von Teilen der SDL-Spezifikation kann der Nutzer direkt auf die Effizienz des generierten Programmes Einfluß nehmen. Der Datenaustausch mit anderen Systemkomponenten ist mit alternativen Notationen möglich.

Trotz der Nachteile in den Punkten 1, 2, 3 und 6 kann in der Praxis auf alternative Codefragmente nicht verzichtet werden. Die Gründe dafür sind:

- Der Nutzer ist nicht in der Lage, sein Problem adäquat mit SDL zu beschreiben.
- Der generierte Code ist für die Anwendung entweder nicht effizient genug oder auch zu groß, da das hohe Abstraktionsniveau von SDL zu umfangreichen Implementierungen führt. Oft hat das SDL-Werkzeug keine Chance, Optimierungen durchzuführen, da eine spätere Verwendung der bei der Codegenerierung noch nicht genutzten Konzepte nicht ausgeschlos-

sen werden kann. Das betrifft insbesondere die Vorübersetzung von SDL-Modulen mit dem Ziel, Bibliotheken zu erstellen.

- Die Nutzung eines funktional eigenständigen Softwareprodukts über vorgegebene Schnittstellen soll aus SDL heraus ermöglicht werden. Beispielsweise sind bei Protokollen im Telekommunikationsbereich oft Datenbankzugriffe notwendig.

Durch die Bereitstellung neuer, adäquater Datenkonzepte für SDL würden viele Gründe, externe Daten zu verwenden, entfallen. Ist man trotzdem gezwungen, alternative Beschreibungen einzusetzen, sollte dieses in der Spezifikation gut erkennbar sein. Für wiederholte Anwendungen könnte man sogar semantisch fundierte Schnittstellen als Ergänzung des Sprachstandards definieren. Denkbar wären beispielsweise normative Datenbankschnittstellen.

4.2 Integration mit ASN.1

ASN.1 ([Gora98], [Schr94], [Ste93]) wurde im OSI-Kontext [X.200] für die abstrakte Beschreibung und architekturunabhängige Übertragung von Daten in offenen, verteilten Systemen entwickelt und standardisiert [X.208]. Wie SDL ist auch ASN.1 ist eine lebendige Sprache. Weitere Entwicklungen erfolgten in einem separaten Standard [X.680]. Beide Sprachversionen werden praktisch verwendet. Die Integration von ASN.1 mit SDL ist weitestgehend X.208-basiert. Deshalb ist bei der Verwendung des Begriffes ASN.1 in dieser Arbeit stets die Sprachdefinition X.208 gemeint. Auf die neue ASN.1-Version wird im Bedarfsfall stets explizit verwiesen. Der erste Unterabschnitt gibt einen Überblick für beide Sprachversionen.

Wegen der Standardisierung von ASN.1, der Verbreitung und der Beschränkung auf die strukturelle Beschreibung von Daten ist es naheliegend, ASN.1 zur Beschreibung von Daten für SDL zu nutzen. Eine Kombination von SDL mit ASN.1 wurde 1995 von der ITU-T standardisiert [Z.105]. Aus historischen Gründen gibt es speziell für die SDL-Werkzeugumgebung SITE eine alternative Integration von ASN.1 in SDL. Beide Methoden verbessern die Möglichkeiten, Daten in SDL zu spezifizieren.

Nach der Diskussion der Grundlagen beider Integrationsmethoden werden die ASN.1-Sprachkonzepte näher beleuchtet, die bei der Integration mit SDL für SITE interessant sind aber deren Verwendung in SDL nicht standardisiert wurden. Das betrifft

- telekommunikationsspezifische ASN.1-*macro*-Anwendungen,
- Datencodierung,
- ASN.1-ANY und
- Datentyperweiterungen als Konzept der Sprachdefinition in X.680.

Die Betrachtung des Übertragungsaspekts von ASN.1 wird jedoch in das Kapitel 5 „*Datenbasierte Kommunikation*" (S.75) verschoben, da einerseits die Integration von IDL im Abschnitt 4.3 „*Kombination mit IDL/ODL*" (S.71) eine ähnliche Diskussion erfordert und andererseits die Kommunikation weit über den Aspekt von Datendefinitionen hinausgeht.

Der ASN.1-Abschnitt wird mit einer Einschätzung des Nutzens der ASN.1-Integration für die SDL-Sprachentwicklung abgeschlossen.

4.2.1 ASN.1 im Überblick

ASN.1 ist eine Sprache zur Beschreibung von Datentypen und Datenwerten. Es gibt jedoch keine Konzepte, Werte zu manipulieren, insbesondere steht kein Operatorkonzept zur Verfügung. Die angesprochene Übertragung von Werten basiert auf ebenfalls standardisierten Codierungskonzepten, die Werte auf eine Folge von Bytes und umgekehrt abbilden.

Die Stärke dieser Sprache ist es, daß sich ihre Konzepte mit wenigen Einschränkungen in Implementierungssprachen abbilden lassen. Eine Vielzahl von sowohl kommerziellen als auch frei verfügbaren Werkzeugen unterstützt solche Abbildungen. Schnittstellen zur Manipulation von Daten, insbesondere deren Codierung, werden jedoch in den entsprechenden Programmiersprachen durch die ASN.1-Werkzeuge sehr unterschiedlich bereitgestellt.

Eine ASN.1-Beschreibung ist ein Modul mit Import- und Exportmechanismen für andere Module, das eine nicht weiter strukturierbare Menge von Datentyp-, Wert- oder Syntaxdefinitionen in Form von Zuweisungen enthält. Alle mit ASN.1 beschriebenen Datentypen lassen sich mit der Vorschrift BER [X.209] codieren.

Datentypzuweisung: Durch eine Datentypzuweisung wird einem Datentyp ein Name zugeordnet. Obwohl dieses intuitiv eine Datentypdefinition ist, wird der Begriff „Datentypzuweisung“ wegen der englischen Vorgabe „*type assignment*“ in der ASN.1-Sprachdefinition weiter verwendet. Datentypen sind entweder primitiv oder konstruiert.

primitiver Datentyp: Datentypen, die nicht auf der Basis anderer Datentypen konstruiert werden, sind primitive Datentypen. Dazu zählen beispielsweise BOOLEAN oder INTEGER, aber auch Aufzählungen (ENUMERATED).

konstruierter Datentyp: Konstruierte Datentypen, z.B. CHOICE-, SET-, SEQUENCE-Konstruktionen, setzen sich aus anderen Datentypen zusammen. ASN.1 besitzt eine Vielzahl von Konstruktionsmechanismen, die weit über die Konstruktionsmöglichkeiten von üblichen Programmiersprachen hinausgehen.

Datentypkonstruktionen können verschachtelt werden.

Teilbereichstyp: Vorhandenen Datentypen lassen sich mit komplexen Notationen für Intervall- und Mengenbildung zu Teilbereichstypen einschränken. Bei strukturierten Beschreibungen sind sogar Teilbereichseigenschaften einzelner Komponenten veränderbar.

Wertzuweisung: Durch eine Wertzuweisung wird einem Wert eines mit anzugebenem Datentyps ein Name zugeordnet. Der Wert muß kompatibel zum Datentyp sein. Das entspricht der Definition einer getypten Konstanten.

Wert: Den primitiven Datentypen sind in der Sprachdefinition literale Werte, die durch Zahlen, Namen, Zeichenketten oder Schlüsselwörter beschrieben werden, zugeordnet. Für komplizierte Datentypkonstruktionen gibt es Notationen zur Zusammenfassung mehrerer Werte in einem {}-Paar. Ist die Zuordnung von Werten zu bestimmten Feldern nicht eindeutig, so können den Werten auch die Feldnamen vorangestellt werden. Da ASN.1 kein Operatorkonzept besitzt, sind selbst einfache Berechnungen nicht möglich.

Kompatibilität von Werten zu Datentypen: Die vordefinierte Zuordnung von Werten zu Datentypen ist das Hauptkriterium für die Zuweisungskompatibilität. Wenn bei der Konstruktion eines Datentyps nicht die Art der strukturellen Wertbeschreibung geändert wird, beispielsweise bei Datentypkennzeichnervergabe oder der Bildung von Teilbereichstypen, dann bleibt diese Kompatibilität erhalten.

Werte, die auf der Basis eines Teilbereichstyps zugewiesen werden, sollten natürlich im angegebenen Bereich liegen.

Datentypkennzeichner: Für die standardisierte Codierung von Werten sind alle Datentypen mit vordefinierten Datentypkennzeichnern (*tag*-Konstruktionen) ausgestattet. Bei der Bildung neuer Datentypen durch den ASN.1-Nutzer können auch die Typkennzeichner neu festgelegt werden. Um insbesondere eine eindeutige Decodierung von Werten zu gewährleisten, gibt es für strukturierte Datentypen Regeln, die im wesentlichen die Verwendung eindeutiger Datentypkennzeichner fordern.

Datentyp ANY: Der Datentyp ANY ist ein universeller Platzhalter für eine beliebige Datentypbeschreibung. ANY-Werte können nur in Kombination mit einem konkreten Datentyp angegeben werden, z.B. INTEGER:7.

Syntaxzuweisungen: Mit Hilfe einer *macro*-Notation können neue syntaktische Datentypkonstruktoren und Wertnotationen eingeführt werden. Diese werden durch eine ebenfalls spezifizierbare semantische Transformation auf vorhandene Datentyp- und Wertnotationen abgebildet. Das entspricht einer nutzerdefinierten Syntax- und Semantikerweiterung.

Diese Erweiterungen haben drei wesentliche Probleme:

- die Spezifikation einer konfliktfreien Syntax kann nicht garantiert werden,
- Analysewerkzeuge realisieren auf keinen Fall eine unbeschränkt erweiterbare Syntax,
- es wurden in praktischen ASN.1-Spezifikationen Interpretationen von *macro*-Konstruktionen vorausgesetzt, die nach der semantischen Transformation nicht mehr nachvollziehbar sind.

Aus diesem Grund gibt es nur ASN.1-Werkzeuge, die fest vorgegebene *macro*-Notationen verarbeiten können. Die neue ASN.1-Version unterstützt das *macro*-Konzept in dieser Form nicht mehr.

Namen: In ASN.1 wird die Groß- und Kleinschreibweise beachtet. Die Namen unterteilen sich, abhängig von ihrer Schreibweise, in Typ- und Wertnamen mit jeweils einem großen bzw. kleinen Anfangsbuchstaben. Typnamen können zur Definition von Datentypen oder als Modulename verwendet werden. Wertnamen sind auf Namen von Konstanten, Aufzählungselementen und Feldnamen beschränkt.

Sichtbarkeit: Die Reihenfolge der ASN.1-Zuweisungen ist unerheblich für die Sichtbarkeit von Namen. Alle Datentyp- und Wertdefinitionen des ASN.1-Moduls sowie alle importierten Definitionen sind durch Angabe des Namens referenzierbar. Feldnamen oder Aufzählungselemente sind durch den Datentypkontext in einem Wert sichtbar.

Mit dem neuen Sprachstandard X.680 stehen einige Erweiterungen zur Verfügung. Dafür wurden semantisch bedenkliche Konstruktionen, insbesondere die *macro*-Notation, entfernt. ASN.1-Spezifikationen verwenden zunehmend die neue Sprachversion.

Syntax: Die Basisnotation (Datentyp- und Wertbeschreibungen) von ASN.1 weist nur geringe Änderungen gegenüber dem Standard X.208 auf, da nur grammatikalische Unstimmigkeiten beseitigt wurden.

Datentyperweiterung (extensibility): Drei lexikalisch angegebene Punkte "..." in einer Datentypdefinition kennzeichnen diese als erweiterbar. Es gibt jedoch keinen expliziten Bezug der Erweiterung zur Basisdefinition, so daß die Verwendung des Begriffs Vererbung unangebracht ist. Eine spätere Erweiterungen des Datentyps soll für die Datenübertragung, d.h. letztendlich für die Codierung transparent sein. Die geforderte Transparenz läßt sich gut mit dem Signalkonzept von SDL erklären, wenn Signalparameter als ASN.1-Werte codiert werden:

```
asntype     -- ASN.1-Definitionen der Datentypen Arg1 und Arg2
   Arg1 ::= SEQUENCE{
      i INTEGER,
      ...        -- das kennzeichnet den Datentyp als erweiterbar
   }
   Arg2 ::= SEQUENCE {
      my-addr PId,
      pdu Arg1
   }
endasntype;

/* Signalparameter sind die ASN.1-Datentypen */
signal sig1(Arg1), sig2(Arg2);
```

Ein typisches Szenario für hierarchische Protokollspezifikationen ist das Verpacken einer Datenstruktur in einer anderen, die dann weiterverschickt wird:

```
dcl arg1 Arg1, arg2 Arg2;
/* Teile des Verahltensgraphen ... */
input sig1(arg1);
    task arg2 := (. self, arg1 .);
    output sig2(arg2);
```

Der Parameter des empfangenen Signals kann als Wert des erweiterten Datentyps

```
Arg1-version2 ::= SEQUENCE {
    i INTEGER,
    ...,
    b BOOLEAN
}
```

codiert sein. Außer der strukturellen Ähnlichkeit zu *Arg1* gibt es keine Hinweise, daß *Arg1* der Ausgangspunkt dieser Definition ist. Insbesondere muß *Arg1-version2* dem Empfänger nicht bekannt sein.

Die Decodierung von *arg1* beim Signalempfang darf keinen Fehler über den zusätzlich codierten Datenwert in der Struktur liefern und der codierte Wert des Feldes *pdu* von *sig2* darf den Wert des Felds *b* nicht verlieren. Im erweiterten Sinn ist dieses Verhalten mit der Polymorphie von Datentypen vergleichbar.

Bei ASN.1-Aufzählungen oder ASN.1-*choice*-Konstruktionen kann es durch Erweiterungen zusätzliche Elemente oder Felder geben, die unbekannt sind, z.B:

```
Version ::= ENUMERATED { one(1), two(2),...}
-- Aufzählung mit erweiterbaren Elementen
```

Informationsobjekt: Die Möglichkeit zur Definition von Informationsobjekten ersetzt das *macro*-Konzept. Auch hier zulässige Syntaxerweiterungen sind jedoch durch die Sprachdefinition so eingeschränkt, daß entsprechende Implementierungen in ASN.1-Werkzeugen möglich sein sollten. Entsprechende Implementierungen sind jedoch nicht bekannt. Das Problem der informalen Zusatzinformationen ist durch die Definition entsprechender Zugriffe gelöst (*associated table*-Konstruktionen).

verbesserte Teilbereichsbildung: Neben den vorhandenen Mengenoperationen Vereinigung und Durchschnitt stehen jetzt auch die Differenz und die Bestimmung der Komplementärmenge zur Verfügung. Man kann eine Teilbereichsbildung als erweiterbar kennzeichnen, z.B.:

```
Version-1 ::= INTEGER(1..10,...)
```

In einer späteren Version dieser Typbeschreibung kann der Teilbereich nach oben jedoch nicht nach unten erweitert werden.

Parametrisierung: Es ist möglich, abstrakte Datentypen und Werte zu spezifizieren. Das ist vergleichbar mit dem C++-*template*-Konzept oder den SDL-Kontextparametern.

Codierung: Die Abbildung von ASN.1-Werten auf einen Bytestrom mit der Codierungsvorschrift BER ist nicht eineindeutig. Für den neuen ASN.1-Standard wurden deshalb die eineindeutigen Codierungsvorschriften CER und DER - beides Einschränkungen von BER - definiert [X.690]. Eine weitere, mehr kompaktere Form der Codierung ist die Neuentwicklung PER [X.691].

4.2.2 Standardisierte Kombination mit ASN.1

Die ASN.1-spezifische Erweiterung von SDL ist eine neue Sprache mit dem Namen „*SDL in Combination with ASN.1*“ [Z.105]. Es werden die Möglichkeiten erweitert, Daten zu spezifizieren. Die Entwicklung dieser Sprachdefinition durch die ITU-T erfolgte in weniger als einem Jahr unter enormem Zeitdruck, weshalb Prioritäten gesetzt wurden.

- ❍ Nutzer beider Beschreibungstechniken sollten möglichst wenig Umstellungsprobleme haben. Insbesondere sollten SDL-Konstruktionen, die eine semantisch äquivalente ASN.1-Variante besitzen[1], frei austauschbar sein. Als Standardisierungsgremium hatte man dazu das Mandat der SDL-Werkzeughersteller, eine neue SDL-Sprachvariante zu definieren.
- ❍ Die Semantik der ASN.1-Integration sollte mit Blick auf eine spätere grundlegende Änderung des Datenmodells möglichst einfach sein. Deshalb wurde eine umfassende Lösung zur Integration beliebiger Datentypen auf der Basis externer Sorten nicht weiterverfolgt, obwohl die SDL-Sprachdefinition diese Herangehensweise explizit vorschlägt (Z.100 Abschnitt 5.4.6 „*External data*“).
- ❍ Es sollte nur eine Untermenge von ASN.1 unterstützt werden, die auch in künftigen Sprachversionen von ASN.1 vorhanden sein wird.

„*SDL in Combination with ASN.1*“ ist eine Erweiterung der SDL-Sprachdefinition, die bis auf Schlüsselwortprobleme rückwärtskompatibel ist. Bild 11 vermittelt einen syntaktischen Ein-

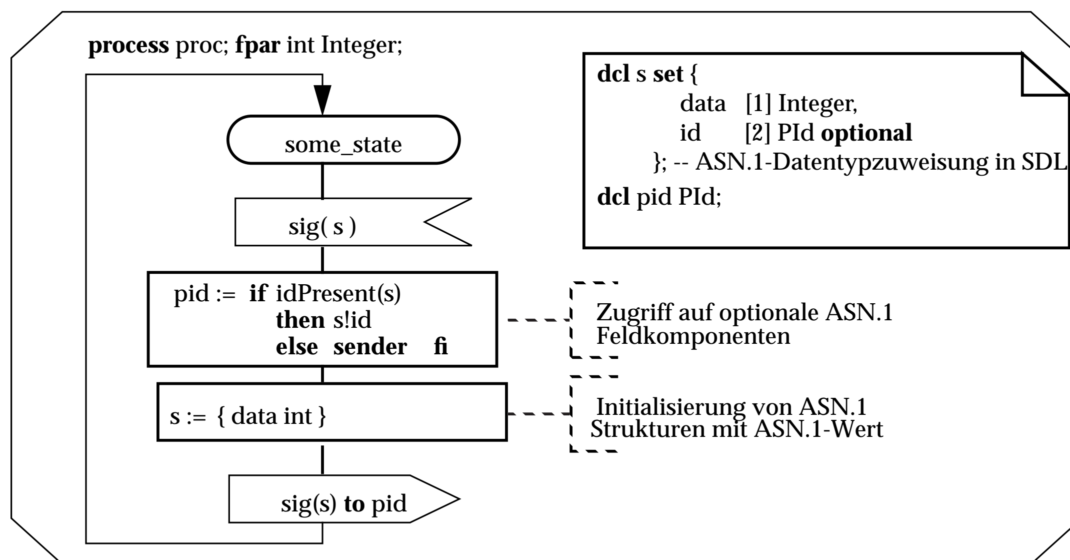

Bild 11: Beispiel einer Spezifikation mit „SDL in Combination with ASN.1“

druck. Die wesentlichen Erweiterungen sind:

- ❍ Datentypen und Konstanten können mit ASN.1-Konstruktionen definiert werden;
- ❍ ASN.1-Wertnotationen sind als Ausdruck in SDL zulässig;
- ❍ ASN.1- und SDL-Module sind nur unterschiedliche syntaktische Varianten des SDL-Modulkonzepts;

1. Beispielsweise ist die SDL-Wertnotationen *(. True, 42 .)* austauschbar mit der ASN.1-Notation *{ True, 42 }*.

❍ Anstelle von Bezeichnern, die sich auf eine SDL-Datentypdefinition beziehen, können direkt ASN.1-Datentypkonstruktionen angegeben werden. Das entspricht einer *inline*-Datentypdefinition.

Die semantische Fundierung von „*SDL in Combination with ASN.1*“ ist eine vollständige Transformation mit geringfügigen Erweiterungen der SDL-Semantik selbst. Für alle ASN.1-Konstruktionen sind in der Sprachdefinition Regeln angegeben, wie diese sich mit SDL repräsentieren lassen. Um auch kompliziertere ASN.1-Teilbereichstypen adäquat nach SDL übersetzen zu können, wurde die semantische Basis des SDL-*syntype*-Konzepts erweitert.

Insgesamt hat diese neue, sprachspezifische Erweiterung von SDL einige Schwächen.

Lexik und Syntax

Die lexikalisch syntaktische Kombination der beiden Sprachen führt zu Einschränkungen in beiden Sprachen. Eine vorliegende ASN.1-Spezifikation kann oft nicht ohne Änderung in eine SDL-Spezifikation integriert werden. Änderungen werden notwendig für:

- SDL-Schüsselwörter in ASN.1-Spezifikationen,
- die Verwendung von '-' in ASN.1-Namen,
- mehrdeutige Namen auf Grund der ignorierten Groß- und Kleinschreibung,
- nicht unterstützte X.208-Konstrukte, z.B. ASN.1-*macro*-Anwendungen und
- die Verwendung von X.680-spezifischen Konstrukten.

Semantik

Durch die semantische Transformation gibt es Konflikte bezüglich der statischen Semantik von ASN.1 und der in SDL integrierten ASN.1-Variante. Ein Beispiel für eine gültige ASN.1-Spezifikation, die in Kombination mit SDL falsch ist, ergibt sich aus der Abbildung von *integer*-Konstanten in ASN.1-*integer*-Definitionen auf SDL-*synonym*-Konstruktionen. Zwei gleiche Namen in verschiedenen ASN.1-*integer*-Definitionen erzeugen einen Namenskonflikt bei der Abbildung:

```
LastModify ::= Integer { unknown(0) }
Version ::= Integer { unknown(-1), one(1), two(2), tree(3) }
```

Mit den Transformationsregeln ergibt sich folgende SDL-Spezifikation:

```
syntype LastModify = Integer endsyntype;
syntype Version = Integer endsyntype;

synonym unknown Integer = 0;
synonym unknown Integer = -1; /* statischer Fehler */
...
```

Umgekehrt gibt es auch ASN.1-Fragmente innerhalb von SDL, die keine gültigen Beschreibungen für die Sprache ASN.1 sind. Der Standard Z.105 ignoriert beispielsweise die Codierung und damit konkret alle Regeln für Datentypkennzeichner.

Sprachkonformität

ASN.1 und SDL passen als Sprache stilistisch nicht zusammen. Auch wenn die ASN.1-Erweiterung von SDL die Abbildung von ASN.1 auf SDL erleichtert, so gibt es noch grundsätzliche Unterschiede, wie sie beispielsweise bei *inline*-Datentypdefinitionen von ASN.1 deutlich werden. Eine direkte Verwendung vorhandener ASN.1-Module erweist sich durch die oben beschriebenen lexikalischen Kompromisse bei der Kombination von SDL und ASN.1 als problematisch.

Die Sprache „*SDL in Combination with ASN.1*“ ist an die Sprachversion SDL-92 gebunden. Mit einer formalen Sichtweise auf die Sprachdefinition ist es deshalb nicht möglich, SDL-96 in Kombination mit ASN.1 zu verwenden. Die Syntaxregeln für die ASN.1-Konstruktionen beziehen sich auf die ASN.1-Version aus X.208, neue ASN.1-Konstruktionen oder auch nur Sprachkorrekturen aus dem Standard X.680 sind nicht verwendbar. Mit diesen Versionsproblemen ist der Nutzer von SDL-Werkzeugen auch praktisch konfrontiert.

Als Konsequenz auf die Veröffentlichung des SDL-2000-Sprachstandards wurde auch die ASN.1-Kombination formal überarbeitet, wobei man auf eine stärkere syntaktische Trennung beider Sprachen setzte[1]. Es bleibt abzuwarten, wie Werkzeughersteller auf SDL-2000 und in letzter Konsequenz auf die ASN.1-Kombination reagieren werden.

4.2.3 ASN.1-Einbettung mit SITE

Zu Beginn der SITE-Werkzeugentwicklung war klar, daß auf Grund der Schwierigkeiten mit der ACT ONE-Implementierung eine alternative Variante zur Beschreibung der Daten gefunden werden mußte. Die Standardisierung einer ASN.1-Kombination mit SDL wurde zu diesem Zeitpunkt noch nicht diskutiert, so daß eigene Prioritäten gesetzt wurden.

- Der SDL-Sprachstandard sollte nicht oder nur sehr gering verändert werden. Es ist für die Nutzerakzeptanz von Werkzeugen wichtig, daß der Sprachstandard eingehalten wird. Nur so können bereits vorhandene Spezifikationen wiederverwendet bzw. auch Spezifikationen mit ASN.1-Bestandteilen beschränkt von standardkonformen SDL-Werkzeugen verarbeitet werden. Da die SITE-Werkzeuge sich stets auf SDL/GR-Editoren anderer Hersteller verlassen, war dieser Punkt bei der Entwicklung zwingend.
- Mehrdeutigkeiten bei der Interpretation der ASN.1-Konstruktionen sind durch eine semantische Fundierung der Kombination mit SDL auszuschließen.
- ASN.1-Spezifikationen sollen möglichst in ihrer Gesamtheit und ohne Änderungen mit SDL kombinierbar sein. Das gestattet eine Verarbeitung der Spezifikationen durch SDL und ASN.1-Werkzeuge ohne speziellen Aufwand.
- Die zu entwickelnde Methode sollte sich auch für andere externe Sprachen verallgemeinern lassen.

Wegen eines expliziten Verwendungshinweises (Z.100 Abschnitt 5.4.6 „*External data*“) für eine ASN.1-Kombination, wurde die vorhandene Schnittstelle für externe Daten aus dem Abschnitt 2.3 „*Externe Datenbeschreibungen*” (S.38) genauer untersucht.

Da ASN.1 kein Operatorkonzept besitzt, ist die Nutzung einer einfachen Einbettungssemantik nicht ausreichend. Der Nutzer von ASN.1 in SDL erwartet natürlich einen üblichen Satz von Operatoren für die Datentypen. Deshalb müssen Operatoren nachträglich hinzugefügt werden. Für einfache Datentypen kann das in einem vordefinierten ASN.1-Modul erfolgen, bei konstruierten Daten muß der Nutzer für entsprechende Operatoren sorgen. Das ist umständlich, fehleranfällig und wegen der z. T. komplexen Signaturen für transformationsbasierte Operatoren nicht akzeptabel. Man benötigt eine Transformation, die ASN.1-Beschreibungen entsprechend expandiert. Deshalb ist die ASN.1-Integration von SITE eine Spracherweiterung mit transformationsbasierter Einbettungssemantik, die im Folgenden durch drei Schritte näher beschrieben wird.

1. Ein ASN.1-Fragment kann in einer speziellen syntaktischen Klammerung (Schlüsselwörter **asntype** und **endasntype**) an einer beliebigen Stelle, wo sonst Datentypen definiert werden können, in die SDL-Beschreibung eingeschlossen werden. Alternativ kann auch ein ASN.1-Modul direkt als SDL-Modul referenziert werden. Es besteht also eine strikte Trennung der ASN.1-Sprachkonstrukte von denen in SDL.
2. Alle ASN.1-Datentypzuweisungen werden in externe Sorten mit allen Signaturen für die aus ASN.1 ableitbaren Grundeigenschaften und zusätzlichen Operatordefinitionen überführt. Die hinzugefügten Operatoren orientieren sich an den in SDL üblichen Operatoren. Beispielsweise besitzen primitive ASN.1-Datentypen alle Operatoren der vordefinierten SDL-

---

1. Der Sprachstandard zur Kombination von SDL mit ASN.1 wurde in eine rückwärtskompatible Version und eine nur modul-basierte Kombination ohne inline-Typbeschreibungen geteilt. Die Integration kompletter ASN.1-Module hat sich in der Praxis als der eigentliche Anwendungsfall der ASN.1-Kombination herausgestellt.

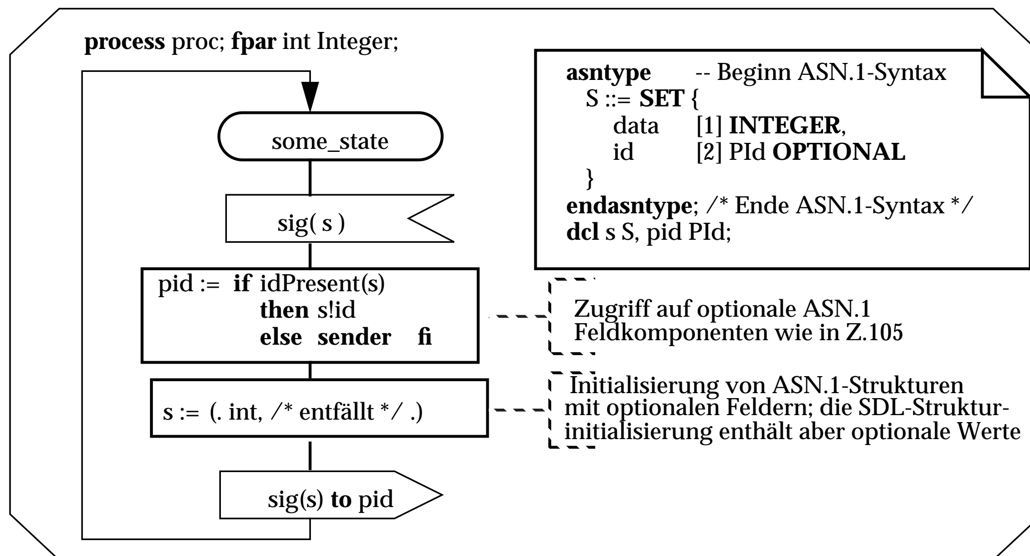

Bild 12: Beispiel für die ASN.1-Integration mit SITE

Sorten. Auch die Operatoren der strukturierten Datentypen sind so definiert[1], daß der Nutzer intuitiv mit SDL-Ausdrücken (z.B. Feld- oder Indexzugriffe) auf die ASN.1-Werte zugreifen kann.

ASN.1-Wertzuweisungen lassen sich zwar begrenzt auf Literale von externen Sorten abbilden, die aktuell genutzte Transformation in SDL-Konstantendefinitionen ist jedoch natürlicher und insbesondere konform zu Z.105.

Besteht die ASN.1-Spezifikation aus einem Modul, wird die Transformation in externe Sorten nur für exportierte Definitionen durchgeführt. Für ASN.1-Module, die in einer separaten Datei definiert sind und wie SDL-Module genutzt werden, erfolgt eine Transformation in SDL-Module. Sonst werden ASN.1-Module aus Sicht der hierarchischen Strukturierung von SDL-Definitionen ignoriert.

Verbleibende, nicht transformierte Konstrukte werden ohne Änderung in eine beliebige externe Sorte übernommen. Sie sind dann nur im Kontext der externen ASN.1-Beschreibung sichtbar.

3. Es werden Regeln definiert, wie Eigenschaften einer ASN.1-Spezifikationen durch Signaturen von externen Sorten reflektiert werden. Beispielsweise entsprechen die ASN.1-Werte *TRUE* und *FALSE* zwei Literalen. Das Verhalten dieser Signaturen wird als extern definiert vorausgesetzt. Wegen der informalen Beschreibung der Datentypen im ASN.1-Standard, sollte bei der Definition der Zuordnung auch eine formale Beschreibung der ASN.1-Konstruktionen angegeben werden. Dazu wurden in [Schr94] mathematische Mengen und Funktionen für die einzelnen ASN.1-Datentypen definiert. Alternativ kann man die jetzt verfügbare standardisierte Übersetzung ASN.1 nach SDL in [Z.105] als formale semantische Grundlage der ASN.1-Konstruktionen nutzen.

   Auch umgekehrt haben ASN.1-Beschreibungen Zugriff auf SDL-Bezeichner von Datentypen, Konstanten und Modulen.

1. Insbesondere erfolgt die Definition konform zum Standard Z.105.

Die Konflikte zwischen vordefinierten SDL-Sorten und entsprechenden primitiven ASN.1-Datentypen werden gelöst, indem die SDL-Sorten durch vordefinierte externe ASN.1-Sorten ersetzt werden. Da sowohl ASN.1 als auch SDL die unendlichen Grunddatentypen (z.B. ganze Zahlen) als mathematische Mengen betrachten, unterscheiden sich die ersetzten Datentypen kaum[1] vom jeweiligen Original. Ein Beispiel für die Verwendung von ASN.1 in SITE enthält Bild 12.

Die Verwendung einer Einbettungssemantik hat Konsequenzen:

- Die ASN.1-Bestandteile sind gültige ASN.1-Konstruktionen.
- Zusätzliche Eigenschaften der ASN.1-Konstruktionen, wie beispielsweise die Codierung von Werten, können explizit oder implizit von der SDL-Spezifikation genutzt werden.
- Die syntaktische Trennung von SDL und ASN.1 gestattet eine standardkonforme Verwendung beider Sprachen. Werkzeugtechnisch könnte man entsprechende Analysewerkzeuge sogar getrennt implementieren, das den Austausch bei verschiedenen Sprachversionen gestatten würde.

Diese Methode der ASN.1-Integration wird bis heute in den SITE-Werkzeugen verwendet. Mit der Veröffentlichung der Arbeiten in [FiSc93] begann man innerhalb der ITU-T über die konkrete Realisierung einer Kombination von SDL mit ASN.1 nachzudenken. Wegen der Prioritäten in der ITU-T wurden die funktionalen Aspekte, wie die Definition von Operatoren, zum größten Teil übernommen aber syntaktisch ein anderer Weg eingeschlagen. Um den Unterschied zur standardisierten Sprachdefinition gering zu halten, wurden im nachhinein auch semantische Differenzen der SITE-Werkzeuge zum Standard bereinigt. SDL/PR-Spezifikationen, die nur ASN.1-Datentyp- und Wertkonstruktionen benutzen, können wegen der funktional identischen Operatordefinitionen automatisch in eine standardgerechte Kombination von SDL mit ASN.1 überführt werden.

Aber auch diese Integrationsvariante von ASN.1 in SDL hat Schwächen:

Lexik und Syntax

Eine strikte Trennung der ASN.1-Bestandteile von SDL garantiert die uneingeschränkte Nutzung von ASN.1. Problematisch ist hier jedoch, daß der Nutzer alle Abbildungsregeln kennen und insbesondere für Ausdrücke anwenden muß. Über den Standard Z.105 hinausgehende Abbildungsregeln sind als werkzeugspezifische Lösungen nicht portabel. Das betrifft beispielsweise ASN.1-*macro*-Anwendungen, Sprachkonstruktionen der neuen ASN.1-Sprachdefinition X.680 oder die automatische Konfliktbeseitigung bei ASN.1-Namen und SDL-Schlüsselwörtern.

Für optionale Felder und ASN.1-*choice*-Werte gibt es keine adäquaten SDL-Ausdrucksnotationen. Entweder umschreibt der Nutzer diese Werte mit komplizierten Zuweisungen oder die Syntax von SDL muß, beispielsweise, wie in Bild 12 bei der Zuweisung an *s* demonstriert, erweitert werden. Die aktuellen SITE-Werkzeuge unterstützen als Z.105 konforme Syntaxerweiterung zu SDL auch ASN.1-Strukturwerte.

Semantik:

Die Einbettung der grundlegenden Eigenschaften der ASN.1-Datentypen ist für eine formale Sprache problematisch, da die Konformität der externen Semantikdefinition zu den semantischen Eigenschaften von Sorten nicht gesichert ist.

Die drei Schritte zur Einbettung von ASN.1 sind wesentlich komplizierter als die Übersetzung von ASN.1 nach SDL wie sie beispielsweise für die semantische von *„SDL in Combination with ASN.1“* genutzt wird.

Sprachkonformität:

Die Verwendung von SDL-gerechten Ausdrücken ist für einen ASN.1-Nutzer ungewöhnlich. Allerdings ist ein Nutzer von SDL der typische Anwender der Kombination beider Sprachen.

---

1. Der ASN.1-Datentyp REAL kennt beispielsweise zwei zusätzliche Literale PLUS-INFINITY und MINUS-INFINITY, die die neue vordefinierte Sorte auch besitzt.

SDL-konforme Werkzeuge können die SITE-gemäße Einbettung von ASN.1 nicht verarbeiten. Angepaßt für spezifische Editoren wurden Lösungen entwickelt, dem Editor die nicht standardkonformen Erweiterungen zu verbergen (z.B. durch spezielle Annotation für *include*-Mechanismen).

4.2.4 ASN.1-*macro*-Anwendungen

In Abschnitt 4.2.1 „*ASN.1 im Überblick*” (S.55) wurde bereits auf die Problematik der *macro*-basierten Syntaxerweiterungen hingewiesen. Es gibt im Protokollbereich zwei *macro*-Definitionen, die sehr oft angewendet werden: *ERROR* und *OPERATION*. Im definierenden Standard [X.219] wird auch ein grundlegendes Protokollverhalten, basierend auf Instanzen dieser *macro*-Definitionen erklärt, welches durch ergänzende Standards zu spezialisieren ist. Eine Spezialisierung ist die Definition der INAP-Operationen [EN301]. Genau diese Spezialisierung wird auch von den SITE-Werkzeugen für eine spezielle Variante[1] der SDL-nach-C++-Übersetzung unterstützt. Die Unterabschnitte beschreiben die Prinzipien dieser erweiterten ASN.1-Unterstützung für die konkreten *macro*-Definitionen. Alternative Prinzipien sind möglich.

ERROR

Die standardisierte Definition der ERROR-*macro*-Konstruktion ist in Bild 13 angegeben. Eine

```
ERROR MACRO ::= BEGIN
    TYPE NOTATION ::= Parameter
    VALUE NOTATION ::= value (VALUE CHOICE {
        localValue    INTEGER,
        globalValue   OBJECT IDENTIFIER})
    Parameter    ::= "PARAMETER" NamedType | empty
    NamedType    ::= identifier type | type
END
```

Bild 13: ASN.1-macro-Definition ERROR

Typ- und Wertnotation muß sich an die Syntaxregeln der *macro*-Definition halten.

Beispiel:

```
IO-Type ::= ERROR PARAMETER Reason
io IO-Type ::= localValue 42
Reason ::= ENUMERATED { timeout(1), unknown(2) }
```

Entsprechend der ASN.1-Sprachdefinition müssen sich *macro*-Instanzen durch eine gegebene Abbildungsregel (in Bild 13 die Konstruktion: VALUE CHOICE ...) in ASN.1-Typ- bzw. Wertnotationen transformieren lassen. Angewendet auf das Beispiel ergibt sich:

```
IO-Type ::= CHOICE {
    localValue    INTEGER,
    globalValue   OBJECT IDENTIFIER
}
io IO-Type ::= localValue 42
Reason ::= ENUMERATED { timeout(1), unknown(2) }
```

1. In einem Projekt mit der Siemens AG Berlin wurde die Implementierung einer INAP-Kommunikationsschnittstelle zwischen SDL und einem kommerziellen SS7-Protokollstack über TCAP (*Transaction Capabilities Application Part*) realisiert.

Würde ein ASN.1-Werkzeug diese Transformation realisieren, so ginge die Verbindung zwischen dem Datentyp *Reason* und *IO-Type* verloren. Die transformierte Datentypbeschreibung für *IO-Type* enthält keinen Hinweis auf den ursprünglich in der Spezifikation enthaltenen Parameter *Reason*. Deshalb ist es eine gute Strategie, daß ASN.1-Werkzeuge die spezielle *macro*-Notation *ERROR* kennen.

Aus der Sicht des definierenden Standards X.219 beschreiben Instanzen dieser *macro*-Definition Fehler, die in einem System auftreten können. Die Werte der Auswahl (CHOICE) identifizieren den jeweiligen Fehlertyp. Die Art des Fehlers und die entsprechenden Parameter können von einem Protokoll, das auf diesen Datenbeschreibungen basiert, aber durch den ASN.1-Sprachstandard nicht definiert wird, geeignet übertragen werden. Wegen dieses Übertragungsaspekts bietet sich für die Instanzen eine zusätzliche Bindung der Datenbeschreibungen an das SDL-Signalkonzept an, sofern man die Freiheit hat, ASN.1 auf beliebige SDL-Konstruktionen abzubilden. Stehen nur Sorten mit den entsprechenden Signaturen zur Verfügung, hat man die Grenze der externen Einbettung aus Abschnitt 2.3 „*Externe Datenbeschreibungen*" (S.38) erreicht - Signale sind als Signatur natürlich nicht zulässig.

SITE realisiert eine Abbildung der Wertinstanzen einer *ERROR*-Definitionen auf SDL-Signale. Ein derartiges Signal wird implizit in der, die Wertnotation enthaltenden, Struktureinheit definiert und hat den Namen des definierten Werts ergänzt um *_ERROR*. Das erste Argument ist eine protokollspezifische Datenstruktur, die den vordefinierten Namen *TCAPErrArg* trägt, das zweite Argument ist der informale Datentypparameter der *macro*-Anwendung, sofern er vorhanden ist. Aus dem oberen Beispiel entsteht also implizit die folgende Signaldefinition:

signal io_ERROR(TCAPErrArg, Reason);

Das erste Argument vom Datentyp *TCAPErrArg* erscheint unmotiviert, wird aber durch praktische Erfahrungen mit realen Kommunikationsbibliotheken gestützt. Dieser Parameter enthält Informationen des genutzten Übertragungsprotokolls[1] für die *macro*-Notationen und sollte für konkrete Implementierungen offen sein, z.B. indem er vom Werkzeughersteller im SDL-Modul *Predefined* definiert wird. Der Datentyp *IO-Type* und die Konstante *io* können so genutzt werden, wie es in der Transformation unter dem Beispiel angegeben ist.

OPERATION

Die *macro*-Konstruktion *OPERATION* hat als Ziel, die Signatur einer Operation mit ihren Argumenten, dem Ergebnis, möglichen Fehlersituationen und von der Operation selbst gerufene Operationen zu beschreiben. Alle Datentyp- und Wertangaben der Operatorsignatur sind entsprechend der Definition in Bild 14 aus der Sicht von ASN.1 informal. Jede Anwendung der *macro*-Konstruktion wird wieder auf den ASN.1-Datentyp

```
CHOICE {
    localValue   INTEGER,
    globalValue  OBJECT IDENTIFIER
}
```

abgebildet. Die Abbildung einer Wertinstanz dieser *macro*-Definition erfolgt auf zwei SDL-Signale. Die Namen und die dazugehörigen Parameter werden wie folgt gebildet:

Name der Instanz: Die Parameter dieses Signals bestehen aus dem Datentyp *TCAPOpArg* und dem Argumenttyp, sofern vorhanden.

Name der Instanz + *_RESULT*: Die Parameter dieses Signals bestehen aus dem Datentyp *TCAP-ResArg* und dem Ergebnistyp, sofern vorhanden.

---

1. In der konkreten SITE-Realisierung enthält diese Datenstruktur Informationen zur TCAP-Dialog-basierten Adressierung (ein Dialog besteht aus einem Signalspiel, ein Prozeß kann durchaus mehrere Dialoge bedienen) und wählbare Übertragungsmodi, die in SDL mit Signalversendung nicht direkt angegeben werden können.

```
OPERATION MACRO ::= BEGIN
    TYPE NOTATION     ::= Argument Result Errors LinkedOperations
    VALUE NOTATION ::= value (VALUE CHOICE {
        localValue    INTEGER,
        globalValue   OBJECT IDENTIFIER})
    Argument              ::= "ARGUMENT" NamedType
                          |   empty
    Result                ::= "RESULT" ResultType | empty
    ResultType            ::= NamedType | empty
    Errors                ::= "ERRORS" "{" ErrorNames "}" | empty
    LinkedOperations      ::= "LINKED" "{" LinkedOperationNames "}" | empty
    ErrorNames            ::= ErrorList | empty
    ErrorList             ::= Error | ErrorList "," Error
    Error                 ::= value (ERROR)
                          |   type - - shall reference an error type if no error
                                   - - value is specified
    LinkedOperationNames ::= OperationList | empty
    OperationList         ::= Operation | OperationList "," Operation
    Operation             ::= value (OPERATION)
                          |   type - - shall reference an operation type if no operation
                                   - - value is specified
    NamedType             ::= identifier type | type
END
```

Bild 14: ASN.1-macro-Definition OPERATION

Die beiden Datentypen *TCAPOpArg* und *TCAPResArg* müssen vordefiniert sein. Die Angabe der Fehlersituationen (ERRORS ...) und der Unteroperationen (LINKED ...) werden aus Sicht der SDL-Spezifikation ignoriert. Allerdings kann ein SDL-Werkzeug die Informationen verarbeiten und eine erweiterte Ablaufkontrolle realisieren. Alternativ ließen sich auch korrekte Kontrollflußelemente generieren, diese wären jedoch sehr spezifisch für die jeweilige Anwendung.

Die entsprechenden Signale für die Fehlersituationen werden dort bereitgestellt, wo die *macro*-Instanzen definiert sind. Wenn die angegebenen Datentypen mit Namen versehen sind, kann der Nutzer darauf achten, daß die entsprechenden Variablen zum Empfang der Signale in der SDL-Spezifikation diese Namen haben. Das ist jedoch nicht zwingend erforderlich.

Beispiel:

```
RPC ::= OPERATION
    ARGUMENT   INTEGER
    RESULT     BOOLEAN
    ERRORS     { io } -- io definiert in Unterabschnitt „ERROR" (S.64)

is-positive RPC ::= localValue 7
```

Folgende SDL-Signale werden implizit definiert:

```
signal
    is_positive(TCAPOpArg,Integer),
    is_positive_RESULT(TCAPResArg,Boolean);
```

Das SDL-Signal *io_ERROR* ist bereits definiert. *RPC* und *is-positive* sind wieder als Datentyp bzw. Wert für SDL sichtbar.

4.2.5 Codierung von ASN.1-Daten

Die Unterstützung für die Codierung von Werten ist für die Kommunikation eines SDL-Systems mit seiner Umgebung hilfreich. Dieser Abschnitt untersucht lediglich die Grundkonzepte, wie Codierung von ASN.1-Werten semantisch in SDL integriert werden kann. Die Möglichkeiten zur Kommunikation werden im Kapitel 5 „*Datenbasierte Kommunikation*" (S.75) detailliert untersucht.

Voraussetzung für die Codierung der Daten ist, daß ASN.1-Datentypkennzeichner entsprechend der ASN.1-Sprachdefinition interpretiert werden und insbesondere auch eine fehlerhafte Benutzung angezeigt wird. Das steht im direkten Gegensatz zur Z.105-Sprachdefinition und kann bei einigen Z.105-konformen Spezifikationen Änderungen erfordern.

SDL-Datenwerte sind mit dem ASN.1-Codierungsschema nicht codierbar. Folgt man dem SITE-Einbettungsmodell für ASN.1, so sind bereits die wichtigsten vordefinierten Sorten ASN.1-Datentypen und damit deren Werte prinzipiell doch codierbar. Alternativ kann man ein Schema definieren das festlegt, wie einfache Sorten in ASN.1 repräsentiert werden. Das Bild 15 gibt eine

SDL-Sorte	ASN.1-Datentyp
Boolean	BOOLEAN
Integer	INTEGER
Charstring	IA5String
Real	REAL
PId	OBJECT IDENTIFIER
Duration	IMPLICIT PRIVATE [2] REAL
Time	IMPLICIT PRIVATE [1] REAL

Bild 15: Schema zur Abbildung von vordefinierten SDL-Sorten auf ASN.1

Variante dieses Schemas an. Für komplexe SDL-Strukturen sind regelbasierte Ergänzungen möglich. Beispielsweise ist bei einer *array*-Definition zu spezifizieren, wie Index und Wert als ASN.1-Datentyp strukturiert werden:

```
MyArray ::= SEQUENCE {
    defaultValue  [1] WertTyp OPTIONAL,
    arrayField    [2] SET OF SEQUENCE {
        index  [3] IndexTyp,
        wert   [4] WertTyp
    } DEFAULT { }
}
```

Derartige Festlegungen sind nicht standardisiert und deshalb stets werkzeugspezifisch.

In den SITE-Werkzeugen sind die nicht standardkonformen semantischen Tests für Typkennzeichner deaktivierbar. Der explizite Zugriff von SDL auf die Codierungsfunktionalität oder die Typkennzeichner mit entsprechenden Operatoren ist eine Implementierungsoption für SDL-Werkzeuge. Die SITE-Codegenerierung unterstützt mit speziellen Bibliotheken nur explizite Codierung von Werten nach BER.

4.2.6 ASN.1-ANY

Die Verwendung von ANY in einer ASN.1-Spezifikation kann mit verschiedenen Zielstellungen verbunden sein. In Kombination mit SDL ergeben sich jedoch neue Möglichkeiten, den Datentyp ANY in einer Spezifikation zu interpretieren.

1. In einer Spezifikation soll ein universeller Platzhalter für eine Datenbeschreibung eingefügt werden, die noch nicht bekannt ist. Als Platzhalter wird ANY genutzt. Es ist nicht klar, mit welchem Sprachkonzept (sowohl in ASN.1 als auch in der Kombination von SDL mit ASN.1) dieser Platzhalter später ersetzt werden soll, i. Allg. wird die komplette Spezifikation durch eine neue Version ersetzt. Im ASN.1-Standard wird empfohlen, diese einfache ANY-Nutzung nicht zu verwenden.
   In SDL kann ein universeller Platzhalter für eine Sorte durch einen Sortenkontextparameter wesentlich genauer als mit dem Datentyp ANY beschrieben werden. Deshalb wird auch für die Kombination von SDL mit ASN.1 ein konzeptioneller Ausbau dieser Interpretation von ANY nicht weiter verfolgt.

2. Die konkrete Datentypdefinition ist für die Anwendung nicht von Interesse, d.h. es gibt in der Spezifikation keine Zuweisung von Werten zwischen ANY und einem anderen Datentyp. Beispielsweise sind in Protokollschichten oft die Spezifikationen von Datenstrukturen anderer Protokollschichten uninteressant, obwohl sie übertragen werden. Das heißt insbesondere, daß auf mehrfache Codierung für die physische Übertragung der Werte verzichtet werden kann. Die Verwendung des Datentyps ANY für derartige Datenstrukturen ist damit ein syntaktisches Mittel für effiziente Codegenerierung aus SDL.

3. Der Datentyp, der sich hinter ANY verbirgt, hängt von einem anderen Wert ab (erweiterte ANY-Konstruktion "ANY DEFINED BY ..."). In ASN.1 fehlt jedoch die Möglichkeit, die Bindung zwischen Datentyp und Wert zu beschreiben. Deshalb ist eine entsprechende Realisierung der erweiterten ANY-Konstruktion durch ASN.1-Compiler sehr unterschiedlich.
   Wenn in SDL eine Konvertierung von ANY in beliebige Datentypen zur Verfügung steht, z.B. die Codierungsfunktionalität, kann die fehlende formale Beschreibung mit SDL spezifiziert werden.

```
asntype
    Seq ::= SEQUENCE {
            what        INTEGER,
            component   ANY DEFINED BY what
    }
endasntype;

dcl seq Seq,
    integer_var Integer,
    boolean_var Boolean;
...
decision seq!what;
    (0):    task integer_var := DECODE_INTEGER(seq!component);
    (1):    task boolean_var := DECODE_BOOLEAN(seq!component);
    else:   task 'Fehler: unbekannte PDU';
enddecision;
```

   Zur Belegung des Felds *component* muß auch eine Konvertierung nach *ANY* zur Verfügung stehen, z.B.

```
task seq := (. 0, ENCODE_INTEGER(42) .);
```

Die Konvertierung von *ANY* in andere Datentypen und umgekehrt kann auch als unsicheres, polymorphes Konzept[1] für SDL definiert werden.

Gemäß dem Standard Z.105 ist der ASN.1-Datentyp *ANY* eine Sorte *Any_type* mit den Operatoren "=" und "/=". Formal ist diese Sorte leer, d.h. die Konstruktion von Werten ist nicht möglich. Das kommt der oben charakterisierten Interpretationsmöglichkeit 2 nahe. ASN.1-Werte können nur aus der SDL-Umgebung in das System gelangen und sind nicht interpretierbar.

Die SITE-Werkzeuge unterstützen abhängig von der verwendeten SDL-Bibliothek sowohl die Optimierung von Datenübertragungen entsprechend der Variante 2 als auch die Verwendung von ANY als unsicheres polymorphes Konzept wie in Variante 3 charakterisiert.

4.2.7 Realisierung von Datentyperweiterungen (*extensibility*)

Die syntaktische Notation für Datentyperweiterungen ist rückwärtskompatibel zum X.208-Standard. Deshalb kann eine X.208-konforme Analyse problemlos erweitert werden.

Bei Datentyperweiterungen für Aufzählungstypen gibt es möglicherweise Werte, die keine Literale besitzen. Da auf Grund der Codierung der Zahlenwert des Aufzählungselements bekannt ist, können die vordefinierten Vergleichsoperatoren und der Operator *Num* gültige Ergebnisse liefern (vgl. Z.105 Abschnitt 4.2.5 „*Enumerated*"). Für andere Operatoren sollte die Datentyperweiterung ignoriert werden.

Beispiel:

```
asntype
    Version ::= ENUMERATED { one(1), two(2),...} -- mit erweiterbaren Elementen
andasntype;
signal s(Version);
dcl v Version;
/* weitere Teile des Verhaltensgraphen */
input s(v);
    decision v;
        (one) : task 'Aktion für Version 1';
        (two): task 'Aktion für Version 2';
        else :  task 'Fehler: unbekannte Version';
    enddecision;
```

Für Struktur- und Auswahlkonstruktionen (SEQUENCE, CHOICE) betreffen Datentyperweiterungen aus semantischer Sicht nur die Codierung. Möglicherweise benötigt man in einer SDL-Spezifikation den Zugriff auf unbekannte Erweiterungen, um die zusätzlichen Datenwerte in andere Protokolldatenstrukturen (Signale) geeignet einzubauen. Das kann durch zusätzliche Operationen oder Feldzugriffe erreicht werden. Angenommen, die ASN.1-Beschreibung wäre

```
Arg1 ::= SEQUENCE{
    i INTEGER,
    ...
}.
```

Als Zugriffsvarianten für die unbekannten Werte ergäben sich drei semantische Alternativen:

1. Die Konvertierung von ANY in einen Datentyp kann zu einem dynamischen Fehler führen. Ein konkretes SDL-Werkzeug kann ein tolerantes Verhalten bei Konvertierungsoperatoren besitzen, um beispielsweise polymorphe Zuweisungen zu erlauben.

```
operators
    ellipsisExtract!   : Arg1 ->   Octet_String;
    ellipsisExtract!   : Arg1 ->   Asn_Any;
    ellipsisExtract!   : Arg1 ->   anomymous_Arg1_Typ;
```

Abhängig vom Ergebnis der Operation gibt es verschiedene technische Interpretationen:

- *OCTET STRING*, um die codierten Werte zu repräsentieren (diese Variante ist in den SITE-Werkzeugen realisiert),
- *ANY*, als Abstraktion gegenüber der Codierung,
- ein anonymer Datentyp, um Zuweisungen zwischen verschiedenen Strukturen zu unterbinden.

Ein analoges Schema kann für Auswahlkonstruktionen angewendet werden.

4.2.8 Zusammenfassung

Vergleicht man die Anforderungen aus Abschnitt *„Anforderungskatalog für neue Datenkonzepte"* (S.39) mit den Methoden der Kombination von SDL und ASN.1, so werden einige Mängel deutlich:

1. *Eine verbesserte Notation für Datentypen sollte sich harmonisch in die zentralen Strukturierungskonzepte Vererbung, Parametrisierung und Sichtbarkeit einpassen.*
   Die syntaktische Kombination von ASN.1-Datentypedefinitionen mit den SDL-üblichen Vererbungs- und Parametrisierungsnotationen ist nicht möglich. Vererbung und Parametrisierung ist nur mit Sorten möglich, die allerdings auf Grundlage vorhandener ASN.1-Datentypdefinitionen definiert werden können, z.B.

   ```
   IntSet ::= set of Integer; -- Z.105 Syntax

   newtype MySet inherits IntSet operators all; adding
       operators delete_all : Integer,MySet -> MySet;
       ...
   endnewtype;
   ```

   ASN.1 selbst gehört nicht in die Kategorie der objektorientierten Sprachen.

2. *Neue Datenkonstruktionen sollten eindeutig definiert werden, damit SDL nicht die Qualität einer formalen Sprache verliert.*
   ASN.1-Konstruktionen lassen sich im wesentlichen nach SDL übersetzen. In diesem Sinne ist die Interpretation der ASN.1-Bestandteile einer SDL-Spezifikation eindeutig.

3. *Bereits vorhandene Datendefinitionen sollten ohne großen Aufwand auf die neu zu definierenden Konzepte abbildbar sein.*
   Die ASN.1-Integration ist eine rückwärtskompatible Erweiterung von SDL. Lexikalische oder syntaktische Probleme, z.B. die Verwendung von Bezeichnern, die in der Kombination Schlüsselwörter sind, lassen sich mit automatischen Verfahren beheben.

4. *Alle Bestandteile einer Datendefinition, die inhaltlich zu dieser Datendefinition gehören, insbesondere Operatoren, sollten in dieser Datendefinition und nur dort erfaßt werden.*
   Die vorhandene Situation für SDL-Konstruktionen bleibt durch die Einbettung der ASN.1-Konstruktionen unberührt. Innerhalb von ASN.1-Konstruktionen können keine Operatoren definiert werden. In der SDL/ASN.1-Praxis kann man die extensive Nutzung von Prozeduren anstelle von Operatoren beobachten. Folglich ist die Gesamtsituation eher schlechter als ohne ASN.1-Kombination.

5. *Die Möglichkeiten zur strukturellen Beschreibung von Sorten sollten erweitert werden.*
   Da ASN.1 direkt für die Beschreibung struktureller Daten entworfen wurde, sind nach der Kombination von SDL mit ASN.1 keine Schwächen zur strukturellen Beschreibung der Daten mehr erkennbar.

6. *Neue Konstruktionen sollten durch SDL-Werkzeuge implementierbar sein.*
   ASN.1 ist eine verbreitete Technik zur Spezifikation von Daten, die auch durch Werkzeuge, z.B. *Snacc* [SaNe93] unterstützt wird. Gegenwärtig bieten die geläufigen SDL-Werkzeuge ASN.1-Unterstützung für ausgewählte Komponenten, die z.T. nicht vollständig oder nicht konform zum Standards Z.105 ist.

7. *Es sollte Sprachelemente geben, die Eigenschaften eines SDL-Programms als Teil einer realen Systemumgebung unterstützen.*
   Die Einflußnahme des Nutzers auf die effiziente Codegenerierung aus SDL ist mit ASN.1 nicht wesentlich verbessert worden. Da durch die Verwendung von ASN.1-Datentypen sehr viele Operatoren implizit vereinbart werden, die bei der Codegenerierung letztendlich realisiert werden müssen, entstehen extrem große Codefragmente nur für Datentypen.

   Durch die standardisierte Codierung von ASN.1-Werten ist eine Kommunikation mit anderen Programmen begrenzt möglich. Dieser Aspekt wird noch einmal im Kapitel 5 aufgegriffen.

Obwohl die Kombination von SDL mit ASN.1 auch in der Praxis als erfolgreich zu bewerten ist, zeigen die Punkte 1, 4 und 7 die Mängel: es fehlen weiterhin moderne Sprachkonzepte für Daten und die automatische Transformation von SDL-Spezifikationen in effiziente Programme bleibt schwierig.

Insgesamt wurde die Kombination von SDL mit ASN.1 von den SDL-Nutzern angenommen. Der industrielle Einsatz der SITE-Werkzeuge bestätigt sowohl diese Aussage als auch die aufgezeigten Mängel.

4.3 Kombination mit IDL/ODL

Die Sprache IDL [OMG98] wurde zur Beschreibung von Schnittstellen zur Kooperation von Objekten in offenen verteilten Systemen von der OMG entwickelt. Zum Umfeld der Sprachdefinition gehören zusätzlich

- Regeln zur Abbildung von IDL in verschiedene Sprachen, z.B. C oder C++,
- Protokolle zur Bedienung der spezifizierten Schnittstellen,
- die Codierungsvorschrift CDR zur Übertragung von Datenwerten und
- Möglichkeiten, Kommunikationsendpunkte in Programmen eindeutig zu adressieren.

Von der ITU-T wurde sowohl eine IDL-Version als ITU-IDL [X.920] als auch eine telekommunikationsspezifische IDL-Erweiterung unter dem Namen ODL [Z.130] standardisiert. Basierend auf den Arbeiten von Born und Winkler [BoWi96] wurde für ODL eine standardisierte Abbildung nach SDL definiert, die in Form eines Anhangs Bestandteil der ODL-Sprachdefinition ist. Deshalb ist insbesondere ODL ein wichtiger Kandidat für eine Integration mit SDL. Da IDL eine Teilmenge von ODL ist, wird im Folgenden nur die Sprache ODL betrachtet.

Dieser Abschnitt beschränkt sich auf eine Einführung der wichtigsten ODL-Konzepte und die Grundideen der Abbildung von ODL nach SDL mit einer vorläufigen Einschätzung der Integration von ODL aus Sicht der Daten. Die eigentliche Bedeutung von ODL für Interaktion von SDL mit seiner Umgebung wird im Kapitel 5 „*Datenbasierte Kommunikation*" (S.75) herausgearbeitet.

4.3.1 ODL im Überblick

Mit ODL werden im wesentlichen Zusammensetzungen von Funktionsschnittstellen beschrieben. Schnittstellenbeschreibungen lassen sich vielfältig strukturieren, die oberste Strukturierungseinheit ist ein ODL-Modul.

Modul: Ein ODL-Modul unterstützt Importmechanismen und faßt verschiedene ODL-Strukturen zusammen. Die wichtigsten Strukturen sind Schnittstellen, Ausnahmen, Objekte, Gruppen, und Daten.

Objekt: ODL unterstützt Strukturierung basierend auf Mehrfachvererbung. Objekte sind die entsprechende Strukturierungselemente.

Gruppe: In einer Gruppendefinition werden vorhandene Definitionen zu einer funktionalen Einheit zusammengefaßt. Diese Einheit kann mit verschiedensten Eigenschaften eines realen verteilten Systems, welches die Schnittstellen einer ODL-Beschreibung realisiert, assoziiert werden. Beispielsweise kann eine Gruppe die Schnittstellen eines Programms oder einer Gruppe von Programmen auf einem Rechner beschreiben.

Schnittstelle: Eine funktionale ODL-Schnittstelle ist eine abstrakte Beschreibung, die im wesentlichen aus Funktionen, Attributen und Datentypen besteht. Speziell für den Telekommunikationsbereich gibt es auch streambasierte Schnittstellen. Hier werden die Eigenschaften des Datenstroms im Form von Attributen beschrieben.

Funktion: Funktionen werden mit Argumenten, Rückgabeparametern bzw. einem Rückgabewert und Ausnahmen spezifiziert. Hinweise zur Implementierung der Funktionen oder gar Einschränkungen der Verfügbarkeit bezüglich des Zustandes der Implementierung können nur informal angegeben werden.

Attribut: Attribute sind Variablen, die mit der Schnittstelle ansprechbar (lesen und/oder schreiben) sind.

Ausnahme: Falls unerwartete Ereignisse auftreten, können in ODL spezifizierte Ausnahmen geworfen werden, um den normalen Ablauf des Programms zu unterbrechen. Ausnahmen können Datenwerte als Parameter besitzen. Neben nutzerdefinierten Ausnahmen, gibt es zahlreiche vordefinierte Ausnahmen, die konkrete Implementierungsaspekte des zugrunde gelegten Protokolls repräsentieren.

Datentyp: Die Beschreibung der Datentypen orientiert sich stark an praktischen Implementierungen, d.h. es gibt alle üblichen primitiven Datenmengen und Datenkonstruktionen, alles ohne Vererbungskonzepte. Die primitiven Datentypen sind grundsätzlich beschränkt, um die Speicherrepräsentation zu gewährleisten. Ein Konzept zur nutzerdefinierten Manipulation von Datenwerten ist nicht vorhanden, Standardoperationen können nur zur Berechnung einfacher Konstanten genutzt werden.

Jede Schnittstelle impliziert einen besonderen Datentyp, eine Objektreferenz. Mit einem Wert dieses Datentyps, eine Objektreferenz, kann eine Schnittstelle adressiert werden.

Ein Platzhalter für Datenbeschreibungen ist der Datentyp *any*.

Datenwert: Zur Beschreibung von Konstanten oder *default*-Werten gibt es grundlegende Wertnotationen wie Zahlen, Strings oder Strukturinitialisierung.

Zuweisung von Werten: Prinzipiell müssen die Werte einer ODL-Beschreibung zur entsprechenden Datentypbeschreibung passen.

Die Implementierung einer Schnittstelle ist an Regeln gebunden. Objektreferenzen müssen so implementiert werden, daß eine Transformation in einen anderen Schnittstellentyp der Vererbungskette von Schnittstellen möglich ist.

Der Datentyp *any* ist in einer Implementierung durchaus konvertierbar.

Codierung: Wenn eine Kommunikation, z.B. durch den Ruf einer Funktion von einem Programm ausgelöst wird, dann werden die beteiligten Datenwerte im CDR-Format codiert und in einer wohldefinierten Protokolldateneinheit übertragen.

4.3.2 ODL-nach-SDL-Abbildung

Der ODL-Sprachstandard definiert eine vollständige Transformation der Sprache nach „*SDL in Combination with ASN.1*“. Da ODL keine funktionalen Beschreibungen besitzt, müssen die entstandenen SDL-Konstrukte mit dem eigentlichen Verhalten angereichert werden, z.B. durch Spezialisierung der generierten Typbeschreibungen. Die Art der Anreicherung besitzt zwei wesentliche Aspekte:

1. es sollen funktionale Aspekte der in ODL spezifizierten Schnittstelle mit einer formalen Methode angegeben werden,
2. die Ausführung des vervollständigten SDL-Programms unterstützt oder nutzt die in ODL vorgegebenen Schnittstellen, d.h. man ist in der Lage, mit dem SDL-Programm standardkonform zu kommunizieren.

Nur der zweite Aspekt ist im Rahmen dieser Arbeit wesentlich und wird im Abschnitt 5.3 „*Externe Kommunikation mit IDL/ODL*” (S.82) untersucht.

Die Übersetzungsregeln von ODL nach SDL sind komplexer Art, da

- ODL und SDL sehr verschiedene Sichtbarkeitsregeln besitzen,
- ODL-Konzepte besitzt, für die es kein direktes Äquivalent in SDL sondern nur eine umständliche Umschreibung gibt. Einige Konzepte für SDL-2000, z.B. verschachtelte Module und Interfaces, wurden unter dem Gesichtspunkt einer verbesserten Transformation von ODL nach SDL entworfen.

Mehrfachvererbung wird von der Abbildung nicht unterstützt. Es ist sehr wahrscheinlich, daß nach der Einführung von SDL-2000 die standardisierten Transformationsregeln grundlegend vereinfacht werden.

Wegen der komplexen Transformation wird ein SDL-Werkzeug mit ODL-Unterstützung diese auch praktisch realisieren, so daß der Nutzer mit der generierten SDL-Beschreibung weiterarbeitet. Diese Vorgehensweise entspricht auch der allgemein üblichen Verwendung von IDL mit Programmiersprachen.

ODL-Datentypen werden i. Allg.[1] auf ASN.1-Datentypen mit entsprechenden Teilbereichsdefinitionen abgebildet, da sonst ähnlich komplexe Sorten wie im Standard Z.105 erforderlich wären. Es gibt zwei Probleme bei der Abbildung von ODL-Datentypen nach ASN.1:

1. Die Zuweisungskompatibilität von Objektreferenzen wird durch die Abbildung auf die Sorte *PId* erreicht. Damit kann aber nicht mehr statisch getestet werden, ob der Ruf einer Funktion (in SDL das Senden eines Signals oder der Ruf einer entfernten Prozedur) an einer Schnittstelle gültig ist.
2. Der ODL-Datentyp *any* wird auf den ASN.1-Datentyp *ANY* abgebildet. Es gibt jedoch keine Konvertierung in andere Datentypen. Das schränkt die Nutzung von *ANY*-Werten in SDL erheblich ein.

Beide Problemen begründen sich in einem fehlenden polymorphen Zuweisungskonzept von SDL.

Die ODL-nach-SDL/ASN.1-Transformation ist insgesamt keine Neuerung bezüglich der SDL-Datenkonzepte, da

- die Verwendung von ASN.1 im vorigen Abschnitt bereits ausführlich diskutiert wurde und

---

1. primitive Datentypen natürlich auf vordefinierte Sorten und alle Objektreferenzen auf die Sorte *PId*

- ❍ die Probleme mit polymorphen Zuweisungen auf SDL-2000 verschoben wurden.

In diesem Sinne ist die Einschätzung der ODL-nach-SDL-Transformation analog der Einschätzung der ASN.1-Integration im Abschnitt 4.2.8 „*Zusammenfassung*” (S.70). Weitere wesentliche Vorteile ergeben sich jedoch bei der Betrachtung des Kommunikationsaspekts im nächsten Kapitel.

Kapitel 5

Datenbasierte Kommunikation

Kommunikation ist eines der Grundkonzepte von SDL, welches für die Übertragung von Daten sowohl zwischen Prozessen des Systems untereinander als auch zwischen Prozessen des Systems und der Umgebung essentiell ist.

Diese Kapitel widmet sich Problemen bei der Übertragung von Datenwerten, die sich mit der Einbettung eines SDL-Systems in eine größere Programmumgebung ergeben, deren Softwarekomponenten unter anderem durch

- möglicherweise unterschiedliche Implementierungssprachen,
- ihre verteilte Ausführung und
- die Kommunikation auf der Grundlage einer für die Programmumgebung vorgegebenen Basiskommunikation

charakterisiert sind. Untersucht werden Möglichkeiten zur Übertragung von Daten mit SDL-Signalen und mit spezifischen Spracherweiterungen. Abhängig sowohl von Detailentscheidungen bezüglich der Basiskommunikation als auch möglicher SDL-Erweiterungen ergeben sich diverse Varianten für die Kommunikation.
SDL-Erweiterungen mit Programmiersprachen werden u.a. wegen des Datenaustauschs mit andern Softwarekomponenten genutzt. Das wurde bereits in Abschnitt 4.1 „*Verwendung von Programmiersprachen*” (S.51) herausgearbeitet. Mit der Integration der standardisierten Beschreibungstechniken ASN.1 und ODL aus dem vorangegangenem Kapitel in SDL wird es nun möglich, daß ein SDL-System mit der Programmumgebung auf eine universelle Art kommunizieren kann. Man löst sich mit diesen formalen Sprachen von den Implementierungssprachen, die zur Ausführung der Spezifikation genutzt werden. Durch entsprechende Standardisierung wird auch die Variantenvielfalt bei konzeptionellen Detailentscheidungen eingeschränkt. Das erleichtert die Anpassung von Kommunikationsmechanismen an andere Softwarekomponenten, die sich an die gegebenen Standards halten, erheblich.

Die Grundlagen zu den generellen Kommunikationsmechanismen, die mit einem SDL-Programm genutzt werden können, legt der erste Abschnitt. Verschiedene Kommunikationsvarianten, die sich aus der Verwendung von ASN.1 und ODL ergeben werden in jeweils sprachspezifischen Abschnitten diskutiert. Insbesondere für ASN.1 sind alle vorgestellten Varianten in Projekten mit SITE realisiert worden.

Im letzten Abschnitt wird die unvollständige Einschätzung der Integration von ASN.1 und ODL aus dem Kapitel 4 abgeschlossen.

5.1 Externe Kommunikation allgemein

Eine SDL-Beschreibung kann grundsätzlich auf zwei Arten mit seiner Umgebung kommunizieren:

signalbasierte Kommunikation:
Signale werden vom SDL-System an die Systemumgebung gesendet oder von ihm empfangen. Diese Art der Kommunikation ermöglicht einen asynchronen Datenaustausch.

Kommunikation als Nebeneffekt von Spracherweiterungen:
Durch syntaktische aber auch semantische Spracherweiterungen wird eine Kommunikation in der Implementierung realisiert. Die einfachste Form der Erweiterung ist die Annotation mit einer Programmiersprache. In der Regel, jedoch nicht zwingend ist die so realisierte Form des Datenaustauschs synchron.

Beide Formen der Kommunikation spielen für die Einbettung von SDL-Systemen in eine Programmumgebung eine außerordentliche Rolle.

5.1.1 Signalbasierte Kommunikation

Die signalbasierte Kommunikation mit der SDL-Systemumgebung ist ein vom Standard vorgegebener Fall einer externen, asynchronen Kommunikation. Zum Senden eines Signals wird auf Grundlage einer Signalbeschreibung eine Instanz erzeugt. Liegt der Sender außerhalb eines SDL-Systems wird die Signalinstanz abhängig von der jeweiligen Implementierung zwischen dem Senden und dem Empfang im System, vorzugsweise an der Systemgrenze gebildet. An die Signalinstanzen sind aus SDL-Sicht einige Attribute gebunden.

Signalidentifikation: Eine Signalinstanz kann eindeutig einer Signalbeschreibung zugeordnet werden. Diese Eigenschaft wird benötigt, um die dem Signal zugeordnete Transition im Verhaltensgraphen zu finden.

Absender: Der *PId*-Wert des Senderprozesses ist der Absender einer Signalinstanz. Jede Signalinstanz besitzt einen gültigen Absender. Mit der impliziten Prozeßvariable **sender** kann im Verhaltensgraphen auf den Absender zugegriffen werden.

Empfänger: Der Empfänger einer Signalinstanz kann beim Senden des Signals sehr unterschiedlich angegeben werden. Die z. T. kombinierbaren Arten der Adressierung sind:

- direkte Adressierung eines Prozesses mit seinem *PId*-Wert,
- Auswahl eines beliebigen Prozesses aus einer beim Senden angegebenen Prozeßinstanzmenge,
- Benennung der Kommunikationspfade zum Empfänger beim Senden,
- indirekte Adressierung des Empfängers durch die vorhandenen Kommunikationspfade.

Sendet man eine Signalinstanz an die Systemumgebung, so ist entweder der *PId*-Wert des Empfangsprozesses bekannt oder sie gelangt durch die Kommunikationsstruktur zur SDL-Systemumgebung. Kommunikationspfade oder Prozeßinstanzmengen außerhalb des SDL-Systems sind aus Sicht der statischen Semantik nicht bekannt, so daß sich die Information zum Empfänger ab der SDL-Systemgrenze auf einen optional vorhandenen *PId*-Wert reduziert.

Sendet eine externe Komponente Signale an ein SDL-System, wäre es denkbar, daß die Kommunikationspfade bekannt sind, da die externe Komponente nicht unbedingt ein SDL-System sein muß. Diese Umgehung der SDL-Sichtbarkeitsregeln ist jedoch praktisch nicht von Bedeutung, so daß sich die Adresse auch in dieser Richtung auf optionale *PId*-Werte reduziert.

Datenparameter: In einer Signaldefinition können beliebig viele formale Datenparameter angegeben werden. Beim Senden des Signals ist die Belegung der formalen Parameter durch passende Werte möglich aber nicht zwingend. Diese Werte werden ebenfalls transportiert.

Aus der Sicht der Basiskommunikation bleiben einige wichtige Fragestellungen für die nachfolgenden Abschnitte offen, die

❍ das Format der zu übertragenden Signale und deren Parameter,

❍ die Verbindung der beteiligten Softwarekomponenten

betreffen.

Externe Signalformate

Die Basiskommunikation nutzt zur Übertragung von Nachrichten ein spezifisches Datenformat, daß als Protokolldateneinheit (PDU) bezeichnet wird. Die Attribute einer extern zu übertragenden Signalinstanz müssen eineindeutig[1] auf die PDU-Struktur abgebildet werden. Das betrifft die Signalidentifikation, die Adressen und die Datenparameter.

In der Regel wird die Basiskommunikation ein spezifisches PDU-Identifikationsmerkmal mit übertragen. Es muß also eine Abbildung zwischen PDU- und SDL-Identifikationsmerkmal geben.

Für die Definition dieser Abbildung gibt es mehrere Varianten:

Erweiterung von Signaldefinitionen: Signaldefinitionen erhalten eine syntaktische Erweiterung (z.B. einfache Annotationen) mit der eine Abbildung auf die PDU-Identifikation spezifiziert wird. Auf die konsistente Verwendung der Erweiterung in mehreren SDL-Systemen ist zu achten.

Automatische Festlegung: Entsprechend eines wohldefinierten statischen Kriteriums wird die Abbildung implizit festgelegt. Das Kriterium sichert entweder die Konsistenz der Signalbeschreibungen ausreichend oder es ergibt sich ein statischer Test für Signalbeschreibungen.

Verhandlung: Erst wenn Komponenten kommunizieren, wird die Abbildung dynamisch ausgehandelt. Die Möglichkeiten zur Definition und letztendlich die Implementierung eines Verhandlungsprotokolls sind stark von der Basiskommunikation abhängig.

Für die Adressierungsattribute muß eine Methode definiert werden, wie *PId*-Werte auf die Adressierungsmethoden der Basiskommunikation abgebildet werden. In der Regel bildet eine konkrete Implementierung des *PId*-Datentyps einen Container für die Plattformadresse.

Die Datenwerte werden mit Codierungsverfahren an das Format der PDU angepaßt. Sowohl für ASN.1 als auch ODL gibt es standardisierte Codierungsvorschriften.

Verbindungen

Aus SDL-Sicht sind Verbindungen Kommunikationspfade, die mit den Kanälen zur Systemumgebung verbunden sind, aber nicht der SDL-Semantik unterliegen. Auf der Basis vorhandener Kommunikationspfade können Signalinstanzen ohne Empfängerangabe an potentielle Empfänger gesendet werden. Das ist besonders für den initialen Verbindungsaufbau zwischen dem SDL-System und einer externen Komponente wichtig.

Aus Sicht der Basiskommunikation stellen die Kommunikationspfade Zugangspunkte für physikalische Verbindungen dar. Die Zugangspunkte können in unterschiedlichen Relationen zu einem SDL-System stehen.

❍ Alle Endpunkte von Kanälen zur Systemumgebung werden zu einem Zugangspunkt zusammengefaßt.

❍ Die Kanäle selbst repräsentieren die Zugangspunkte.

❍ Jeder Kanal unterteilt sich in mehrere Zugangspunkte. Kriterium der Unterteilung werden Signale oder Gruppen von Signalen sein. In diesem Sinne werden Verbindungen basierend auf Signalen festgelegt.

---

1. Für den Fall, daß ein SDL-System nur Signale sendet, kann die Bedingung auch gelockert werden. Die eindeutige Konstruktion einer Signalinstanz ist nur für den Empfang eines Signals von der Systemumgebung von Bedeutung.

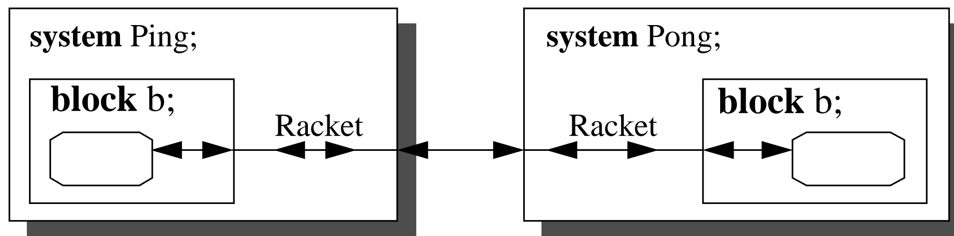

Bild 16: Konfiguration von Verbindungen basierend auf Kanalnamen

Über einen Zugangspunkt können beliebig viele Verbindungen zwischen externen Komponenten und dem SDL-System aufgesetzt werden. Diese Verbindungen können statisch konfiguriert sein oder auf Grundlage eines Regelwerks auch dynamisch aufgebaut werden. Für die dynamische Konfiguration bieten sich beispielsweise die Registrierung von Namen der SDL-Definitionen mit einer eindeutigen Beziehung zu Zugangspunkten bei einem Namensdienst an. In Bild 16 gibt es zwei SDL-Systeme, die auf der Basis gleicher Kanalnamen dynamisch und letztendlich automatisch verbunden werden. Ein nicht in SDL spezifiziertes Programm muß nur den Namen des entsprechenden Kanals kennen, um eine Verbindung zum SDL-System herstellen zu können.

Nach der Betrachtung der Varianten für die Aushandlung der Kommunikationspartner, steht die Frage nach dem dynamischen Verhalten der Verbindung. Aus Sicht des SDL-Systems gibt es zwei Unterscheidungen.

sichere Verbindung: Signalinstanzen werden an den Zugangspunkten gespeichert, bis ein Weitertransport möglich ist. Eine Verzögerung kann eintreten, wenn Kommunikationspartner

- noch nicht gestartet wurden (Initialisierung) oder
- ausgefallen sind.

Eine Implementierung sollte den möglichen Überlauf des Zwischenspeichers beachten.

Auch bei sicheren Verbindungen können Signale verworfen werden, wenn kein Empfänger vorhanden ist oder eine SDL-Transition im aktuellen Zustand das Signal nicht behandelt.

unsichere Verbindung: Signalinstanzen werden verworfen, sobald ein Zugangspunkt nicht mehr in der Lage ist, eine Signalinstanz weiterzutransportieren. Das SDL-System sollte so spezifiziert sein, daß verlorene Signale keine Prozesse blockieren.

Die technischen Attribute „verbindungsorientiert“ bzw. „verbindungslos“ werden in diesem Zusammenhang nicht verwendet, da die physikalische Kommunikation für sowohl sichere als auch unsichere Kommunikation auf beide Arten realisiert sein kann.

5.1.2 Spracherweiterungen

Grundsätzlich realisiert eine Codegenerierung aus SDL für eine spezifische Softwareplattform bereits die Anbindung an die Basiskommunikation. Allerdings sind die Abbildungsregeln einer automatischen Codegenerierung uniform, d.h. ein SDL-Konzept, z.B. das Signalkonzept, wird stets auf ein Plattformkonzept abgebildet. Oft wird jedoch eine nutzergesteuerte Abbildung benötigt. Man möchte mit Signalen sowohl eine Datenbank als auch andere SDL-Komponenten erreichen, obwohl die Plattform dafür zwei unterschiedliche Kommunikationskonzepte zur Verfügung stellt. Mit einer geeigneten SDL-Erweiterung kann eine automatische Codegenerierung die notwendige Anbindung an die Plattform realisieren.

Für die externe Kommunikation werden in der Regel Annotationen mit Programmiersprachen an Datentypen, Aktionen des Verhaltensgraphen oder Signalen genutzt. Hier gilt es, eine konzeptionelle Besonderheit zu beachten: wartet eine extern kommunizierende Instanz auf eine Antwort, so muß die dynamische Semantik des SDL-Systems beachtet werden. Es wäre beispielsweise schlecht, wenn ein vom externen Code geöffnetes Eingabefenster das System blokkiert bis der Nutzer die erwartete Eingabe abgeschlossen hat. Diese informalen und softwarespezifischen Erweiterungen werden hier nicht weiter betrachtet.

Man kann jedoch auch SDL-Erweiterungen mit formale Sprachen zur Beschreibung von Schnittstellen nutzen. Insbesondere die im Kontext von SDL standardisierte Sprache ODL ist ein Kandidat für eine derartige Erweiterung, da bereits eine Transformation von ODL nach SDL definiert ist. Abhängig davon, ob mit ODL die Schnittstellen für ein spezifisches SDL-System erzeugt werden oder ob einzelne SDL-Konzepte universelle ODL-Schnittstellen bereitstellen bzw. nutzen gibt es auch mit ODL verschiedene Varianten.

5.2 Externe Kommunikation mit ASN.1

Die Funktionalität, Werte auf der Grundlage von ASN.1 codieren zu können (siehe Abschnitt 4.2.5 „*Codierung von ASN.1-Daten*” (S.67)), kann vorzugsweise für Signale bei der Abbildung von Werten auf eine PDU-Konstruktion der Basiskommunikation ausgenutzt werden. Die Basiskommunikation selbst und auch das Format für die Zusammenfassung mehrerer Werte, z.B. der Attribute eines Signals, in einer PDU bleibt jedoch offen. Deshalb gibt es trotz der standardisierten Codierung unzählige Varianten, Kommunikation mit ASN.1 zu realisieren. Die SITE-spezifischen Varianten werden kurz vorgestellt.

5.2.1 Kommunikation im Projekt PLATINUM

Im Projekt PLATINUM [BGKS+95] wurden C++-Komponenten mit SDL-Programmen kombiniert. Die Programme des Gesamtsystems können auf verschiedenen Hardwareplattformen (IBM-PC mit WindowsNT oder Linux, Forceboards mit VxWorks, SUN-Workstations mit SunOS) verteilt werden. Die Basiskommunikation ist *socket*-basiert und wurde vom Industriepartner vorgegeben. Die Konfiguration der Verbindungen erfolgt dynamisch per Registrierung an einem Namensdienst.

SDL-Programme kommunizieren ausschließlich per Signal über ein sicheres Medium, wobei pro Kanal ein Zugangspunkt zur Verfügung steht. Das System selbst stellt zusätzlich einen besonderen Zugangspunkt bereit, der alle Kanäle vereinigt und ausschließlich zum Testen genutzt werden sollte. Kanäle zur SDL-Systemumgebung mit gleichem Namen werden automatisch verbunden. Die Bindung mehrerer Kommunikationspartner an einen Kanaleintrittspunkt ist möglich, die Auslieferung nicht mit *PId*-Werten adressierter Signale erfolgt stets an den sich zuletzt gebundene Partner.

Die Identifikation des PDU-Typs erfolgt auf der Basis eindeutig zu vergebenen Nummern. Diese können vom Nutzer oder automatisch den Signalen zugewiesen werden. Für die automatische Zuweisung müssen die paarweise kommunizierenden SDL-Systeme Kanäle zur Systemumgebung enthalten, die den gleichen Namen tragen und strukturell äquivalent sein. Das bedeutet insbesondere, daß

- gleiche Signalnamen verwendet werden,
- Signale mit gleichen Namen kompatible[1] Parametersignaturen besitzen und

1. Die Kompatibilität orientiert sich an der Zuweisungskompatibilität von ASN.1.

❍ die Menge der ausgehenden Signale des einen Systems mit der Menge eingehenden Signale des anderen Systems für beide Richtungen übereinstimmt.

Diese Konsistenzeigenschaften können durch eine angepaßte Semantikanalyse (vgl. Abschnitt 1.5.3 „*Semantische Analyse*” (S.16)) geprüft werden.

Sowohl die Verbindungen als auch die SDL-Prozesse pro Verbindung werden an den Zugangspunkten lokal eindeutig durchnumeriert. In Bild 17 sind in die Zugangspunkte der ein-

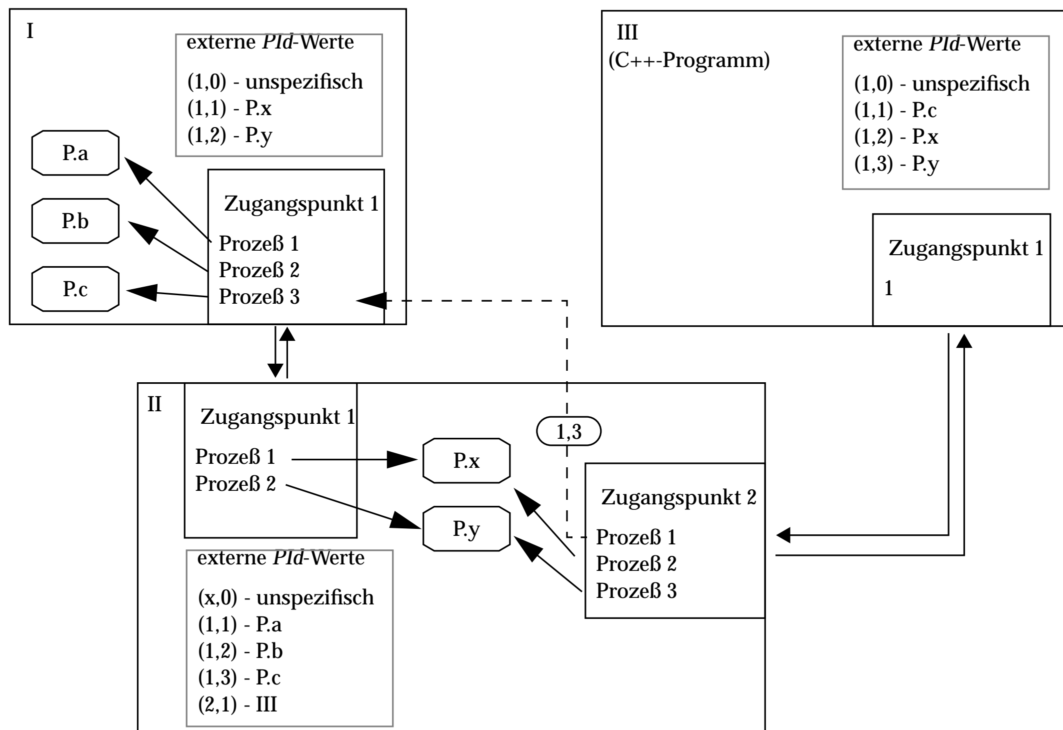

Bild 17: Adressierung in PLATINUM

zelnen Programme jeweils die lokale Nummer des Zugangs und die Prozeßnummern mit den Verweisen (Pfeile) auf die internen Prozesse eingetragen. In PDU-Konstruktionen werden nur Prozeßnummern eingetragen bzw. als Datenparameter eines Signals codiert. Das empfangene Programm ergänzt die Prozeßnummer um seine Nummer des Zugangspunkts, so daß ein nur für dieses Programm gültiges geordnetes Paar *(Zugangsnummer, Prozeßnummer)* entsteht, welches genau einen externen Prozeß adressiert. Diese Paare werden als Liste gültiger externer *PId*-Werte (die gepunkteten Kästchen) von jedem Programm gespeichert.

Wird eine PDU-Konstruktion mit einem externen *PId*-Wert adressiert, so wird vom sendenden Programm der angegebene Zugang genutzt und die Prozeßnummer als Adresse übertragen, um dem empfangenden Programm die Zuordnung zu seinem internen Prozeß zu ermöglichen. Die Prozeßnummer 0 steht für die in SDL mögliche Adressierung auf Grundlage der statischen Kommunikationswege.

Nicht in SDL spezifizierte Programme verwenden die Adressnummer entweder mit einer festen Belegung (im Bild 17 die Nummer 1) oder numerieren aufgebaute Kommunikationsbeziehungen durch.

Soll ein externer *PId*-Wert an ein drittes Programm übertragen werden, so wird am Zugangspunkt wieder eine Nummer vergeben, die nur auf das geordnete Paar verweist (Programm II Zugang 2 Prozeß 1). So kann Programm III einen Prozeß aus Programm I über den Umweg Programm II adressieren, sofern Programm II diese Weiterleitung auch unterstützt. Eine Abkürzung über eine möglicherweise existierende Verbindung zwischen Programm I und III ist mit dieser Methode nicht möglich.

5.2.2 Kommunikation im Projekt VESUV

In Projekten mit der Siemens AG (vgl. [BLS98]) unter der internen Sammelbezeichnung VESUV, werden kommerzielle Softwareprodukte - Programmkomponenten verschiedener Sprachen (C, C++, Java) - mit SDL-Programmen auf der Grundlage unterschiedlicher Kommunikationsmechanismen kombiniert. Für die unterschiedlichen Kommunikationsmechanismen stellt die Siemens AG eine einheitliche Schnittstelle bereit.

In SDL-Programmen werden zwei Kommunikationsmethoden angewendet:

1. externer Code für Operator- und Prozedurdefinitionen zur Realisierung von Datenbankzugriffen,
2. signalbasierte Kommunikation über unsichere Kanäle.

Für weitergehende Details sei auf die projektinterne Dokumentation [Schr02b] verwiesen.

Datenbankzugriffe

Nutzerdefinierte Operatordefinitionen werden hauptsächlich für Zugriffe auf Attribute von Managementobjekten[1] genutzt. Eine Weiterentwicklung ist die Speicherung des Prozeßzustands. Alle wichtigen Zustandsmerkmale eines SDL-Prozesses werden dazu auf eine ASN.1-Struktur abgebildet. Der Prozeß kann die Bildung eines Werts dieser Struktur veranlassen, der anschließend in codierter Form persistent abgespeichert wird. Diese Funktionalität wird für die Absicherung kritischer Daten in Programmen benötigt. Eine allgemeine Beschreibung dieser SITE-Erweiterung ist in [Schr02b] zu finden.

Mit diesem Ansatz als Basis wurde vom Autor eine industriell eingesetzte SITE-Variante entworfen und implementiert, die persistente SDL-Zustände mit *online-upgrade*-Möglichkeit der SDL-Spezifikationen realisiert [CLS01].

Signalbasierte Kommunikation

Die Zugangspunkte des SDL-Systems werden durch Signalgruppen definiert. Die Zuordnung der Signale zu den Zugangspunkten, der konkreten PDU-Konstruktionen für den Zugangspunkt und letztendlich auch die Konfiguration der Verbindungen erfolgt auf der Basis von Managementobjekten. Das ist im wesentlichen eine statische Zuordnung. In speziellen Fällen, z.B. wenn eine Komponente ausfällt, können diese Managementobjekte auch geeignet geändert werden, um die Kommunikation auf Ersatzprogramme umzuleiten. Signale selbst besitzen eindeutige Nummern, die in Syntaxerweiterungen zugeordnet werden. Die Konsistenz wird durch Auslagerung der Signaldefinitionen in von allen Kommunikationspartnern genutzte Module erreicht.

Jedem adressierbaren Prozeß ist auch ein global eindeutiges Managementobjekt zugeordnet, woraus sich die Abbildung eines *PId*-Werts für externe Kommunikation auf die Adresse des Managementobjekts ergibt.

5.2.3 Basiskommunikation mit CORBA

Laufzeitsysteme für SDL, die codierte ASN.1-Werte an andere Programmkomponenten übertragen, sind meist firmenspezifisch (s. Realisierungen der vorangegangenen Unterabschnitte) und verlangen spezifische Lösungen zur Anbindung der jeweils vorhanden Basiskommunikation.

1. Managementobjekte repräsentieren Konfigurationen oder Zustände von Programmen die mit einer geeigneten Schnittstelle gelesen oder geändert werden können.

CORBA bietet eine normierte Kommunikationsplattform, die als Basiskommunikation für SDL-Bibliotheken genutzt werden kann. Diese Bibliotheken sind, insbesondere auch für SDL-Spezifikationen mit ASN.1-Anteilen, nutzbar.

Unter Verwendung einer konkreten CORBA-Implementierung [Cho96] wurde von Bieling [Bie97] eine SDL-Bibliothek für die SITE-Werkzeuge entwickelt, die signalbasierte, externe Kommunikation unterstützt.

Die Zugangspunkte des SDL-Programms sind die externen Kanäle und das System als Vereinigung aller Kanäle. Ein Zugangspunkt ist eine IDL-Schnittstelle, an der ein Signal in das System eingespeist werden kann. Kommunikationsfehler werden ignoriert, in diesem Sinne sind die Verbindungen unsicher. Der vollständige Pfadname des Signals wird als Identifikationsmerkmal genutzt.

Signalparameter werden auf der Grundlage der ASN.1-Codierung als String an die CORBA-Basiskommunikation übergeben. Diese Strings werden dann bei externer Kommunikation noch einmal mit dem CDR-Verfahren codiert. Optimaler wäre eine direkte Unterstützung der CDR-Codierung für ASN.1-Daten oder eine Verwendung von IDL-Daten in SDL.

Die Adressierungsparameter des Signals sind Objektreferenzen für Schnittstellen, die den Transport eines Signals an den adressierten Prozeß gewährleisten. Je nach Effizienzanforderung der konkreten Anwendung kann die Implementierung der Schnittstellen den Transportpfad entsprechend der SDL-Semantik beachten oder den Signalpuffer des Prozesses direkt manipulieren.

Entsprechend der CORBA-Technologie kann sich ein Programm an einen Kanal wenden, sobald es die Objektreferenz des Kanals kennt. Für die automatische Verbindung von SDL-Systemen untereinander werden alle von einem Kanal empfangbaren Signale per Namensdienst registriert, so daß initiale Signale ohne *PId*-Adresse einen bei Mehrdeutigkeit zufällig ausgewählten, empfangsbereiten Kanal in einem anderen System finden. Die Verwendung der vollständigen Pfadnamen für die Signale impliziert die gemeinsame Verwendung von Modulen, womit die Konsistenz der Signalbeschreibungen ausreichend gewährleistet wird.

5.3 Externe Kommunikation mit IDL/ODL

IDL und die telekommunikationsspezifische Variante ODL sind Schnittstellenbeschreibungssprachen für CORBA-Plattformen. Eine CORBA-Plattform kann natürlich als Basis für eine Implementierung einer SDL-Bibliothek genutzt werden. Diese Nutzung ist unabhängig davon, ob ODL-Spezifikationen in SDL nutzbar sind. Im Abschnitt 5.2.3 „*Basiskommunikation mit CORBA*" (S.81) wurde sogar eine Kombination von SDL mit ASN.1 mit einer Laufzeitbibliothek auf CORBA-Basis realisiert. Es liegt für ein CORBA-basiertes SDL-Laufzeitsystem auf der Hand, daß ODL-Schnittstellen von SDL-Konstrukten realisiert werden. Obwohl diese Schnittstellen natürlich spezifisch für die Implementierung der Laufzeitbibliothek sind, liegen sie als Beschreibung sprachunabhängig vor und können mit einer normierten CORBA-Plattform von beliebigen Programmkomponenten genutzt werden. Dieser Ansatz ist bereits wesentlich universeller als die Bedienung einer proprietären Bibliotheksschnittstelle. Da nur die SDL-Bibliothek und nicht die SDL-Spezifikation CORBA-basiert ist, wird diese Variante als eine spezifische Implementierungsvariante nicht weiter verfolgt.

Die Kombination von SDL mit ODL gestattet die Entwicklung CORBA-basierter Spezifikationen. Dieser Ansatz entspricht exakt dem für Programmiersprachen übliche Weg, IDL-Schnittstellenbeschreibungen normiert in die Programmiersprachen zu übersetzen. Es gibt eine normierte Abbildung von ODL nach SDL. Andere Varianten der Kombination von SDL mit IDL/ODL werden auf Grund dieser vorhandenen Normierung nicht betrachtet.

5.3.1 Standardisierte Transformation von ODL nach SDL

Mit der Vorgabe einer ODL-Beschreibung ergibt sich implizit und insbesondere normiert eine Basiskommunikation für die definierten Schnittstellen, sofern eine CORBA-Plattform als Basis genutzt wird. Nach der normierten Abbildung der Schnittstellen auf SDL entsteht auch ein Zusammenhang zwischen den generierten SDL-Konstrukten und der Basiskommunikation. Es stellt sich nur die Frage, ob das SDL-System die Schnittstellen implementiert und anbietet oder voraussetzt, daß eine externe Komponente diese Schnittstelle bereitstellt. Die Regel, daß durch Verhaltensbeschreibungen ergänzte SDL-Konstrukte vom SDL-Programm implementiert werden, erscheint logisch. Es gibt jedoch bisher keine standardisierte Festlegung dafür. Nicht signalbasierte Zielkonstruktionen der ODL-nach-SDL-Abbildung, die externe *server*-Strukturen repräsentieren, müssen entsprechend als semantische Einbettung verstanden werden. Die Einbettung repräsentiert die externe Implementierung der ODL-Schnittstelle.

Erfolgt die Kommunikation signalbasiert mit der SDL-Systemumgebung, dann muß der externe Kommunikationspfad als unsicher angesehen werden. Allerdings gibt es bei Kommunikationsfehlern vordefinierte Ausnahmen, d.h. mit dem gegenwärtigen Abbildungsmodell wird der Sender mit einem Signal über den Verlust informiert.

Die Beschreibung der externen Kommunikation erfolgt ausschließlich mit SDL-Konstrukten, die aus ODL entstanden sind. Da ODL Kommunikationsfunktionalität unabhängig von einer konkreten Programmiersprache bereitstellt, ist man auf diese Art völlig von der konkreten Codegenerierung entkoppelt. Diesem wesentlichen Vorteil steht der Nachteil der komplexen und teilweise problematischen Abbildung von ODL nach *„SDL in Combination with ASN.1“* gegenüber. Wenn die ODL-nach-SDL-Abbildung mit SDL-2000 verbessert werden kann, dann besitzt SDL eine einfache, implementierungsunabhängige standardisierte Kommunikationsschnittstelle.

Das kommerzielle Werkzeug SDT [Tel98] unterstützt eine ODL-Integration in SDL entsprechend dieser Methodik. An einer durchgängigen Unterstützung von ODL für die SDL-nach-C++-Codegenerierung wird im Rahmen der SITE-Weiterentwicklung gearbeitet.

5.4 Zusammenfassung

Die Integration von gebräuchlichen Beschreibungstechniken und insbesondere die Standardisierung der Integration eröffnet eine Reihe von Möglichkeiten, mit einem SDL-Programm zu kommunizieren. Die praktischen Möglichkeiten wurden in einem gemeinsamen Konferenzbeitrag mit dem Projektpartner Siemens AG [KLS99] dem SDL-Fachpublikum auf der Basis eigener Implementierungen erörtert.

Die Vor- und Nachteile der Kombination von SDL mit ASN.1 und ODL wurden aus Sicht der Datentypen bereits im Kapitel 4 „*Realisierte alternative Datenkonzepte*“ (S.51) diskutiert. Offen blieb eine genauere Einschätzung des letzten Kriteriums für beide Sprachen:

Es sollte Sprachelemente geben, die Eigenschaften eines SDL-Programms als Teil einer realen Systemumgebung unterstützen.

In der heutigen Zeit gibt es viele Softwareprodukte, z.B. kommerzielle Protokollstacks, die BER-codierte Daten verarbeiten. Leider ist sowohl die Anbindung an diese Produkte als auch die entsprechende SDL-Abbildung für die Codierung immer noch werkzeugspezifisch. Ursache ist das Fehlen von standardisierten Kommunikationsprotokollen für ASN.1. Sofern ASN.1 mit möglicherweise besseren Codierungsformaten eine bedeutende Basis für Telekommunikationsprotokolle bleibt, wird auch die Kombination von SDL mit ASN.1 unabhängig von den Sprachversionen nicht an Bedeutung verlieren. Es wäre gut, wenn man bei der Sprachentwicklung insbesondere auf eine Vereinheitlichung der Codierungsschnittstellen achtet.

Die Nutzung von ODL und SDL, insbesondere in Zukunft mit einer einfachen, standardisierten Abbildung von ODL nach SDL-2000, ist die bisher durchgängigste Standardlösung mit einem SDL-Programm universell zu kommunizieren. Das ist ein wesentliches Argument für die Standardkonformität. Leider müssen auch einige Nachteile angeführt werden:

- An einen nicht nur akademischen Einsatz ist erst mit der Verfügbarkeit von SDL-2000 zu denken. Bei der rasanten Softwareentwicklung stellt sich die Frage nach der Aktualität einer CORBA-SDL-2000-Plattform. Es ist sogar fraglich, ob CORBA-basierte Plattformen in der Telekommunikation eine wesentliche Rolle spielen werden.
- Die Abbildung von ODL nach SDL setzt voraus, daß SDL-Spezifikationen aus generierten SDL-Fragmenten neu entwickelt werden. Das ist nicht der Normalfall, da SDL-Spezifikationen in der Regel als Standard vorliegen und nur angepaßt und verbessert werden. Offensichtlich ist der umgekehrte Weg, aus vorhandenen SDL-Spezifikationen ODL-Schnittstellenbeschreibungen abzuleiten ebenfalls wichtig. Das ist aber wieder werkzeugspezifisch.

Kapitel 6

Neue objektorientierte Datenkonzepte

Das Datenkonzept von SDL wird i. Allg. als nicht objektorientiertes Konzept aufgefaßt, obwohl es die Vererbung unterstützt und Bestandteil einer anerkannt objektorientierten Sprache ist. Ursache ist das Fehlen wichtiger Konzepte wie Referenzen, Polymorphie und späte Bindung.

Dieses Kapitel widmet sich nun gerade der Bereitstellung dieser bislang fehlendeden Konzepte in SDL. Zunächst werden einleitend die dafür notwendigen Begriffe und Konzepte fixiert, um zwei grundlegende Entwicklungsrichtungen der Sprache SDL betrachten zu können. Die erste und für SDL traditionelle Richtung neue Sprachelemente einzuführen, ist eine Spracherweiterung mit semantischer Transformation auf vorhandene Kernkonzepte. Alternativ ist auch der radikale Neuentwurf des SDL-Datenkonzepts möglich.

Angeregt durch die Aufforderung der ITU-T, Vorschläge für die SDL-Sprachentwicklung zu erarbeiten, entstanden 1998 z.T. mit prototypischen SITE-Implementierungen unterlegte Vorschläge, die dem Transformationsprinzip folgten. Eine auch mit Blick auf die weitere Entwicklung von SDL-2000 systematisierte Betrachtung dieser Vorschläge ist Inhalt des Abschnitts 6.2 „*Transformationsbasierte Datenerweiterungen*" (S.91).

Die ITU-T-Sprachentwickler entschieden sich für die radikale Lösung, das Datenkonzept zu ersetzen. Dieses neue Datenkonzept von SDL-2000 wird im Abschnitt 6.3 „*Substitution des Datenkonzepts durch ITU-T*" (S.120) erläutert.

Da es keine allgemein akzeptierte Liste von Anforderungen für neue objektorientierte Sprachelemente gibt, ist ein systematischer Vergleich der Spracherweiterungen bezogen auf die objektorientierten Eigenschaften der beschriebenen Varianten unmöglich. Für Einzelkonzepte sind natürlich Ähnlichkeiten vorhanden, auf die in der jeweiligen Darstellung konkret verwiesen wird. Eine alternative Sichtweise auf vergleichbare Konzepte verhilft zu Erkenntnissen, die bei isolierter Betrachtung unmöglich sind. In der abschließenden Zusammenfassung werden die Verbesserungsvarianten anhand der allgemeinen Kriterien zur Verbesserung des Datenkonzepts aus Abschnitt „*Anforderungskatalog für neue Datenkonzepte*" (S.39) einer Bewertung unterzogen.

6.1 Neue Begriffe im Kontext von SDL-Daten

Ziel dieses Abschnitts ist es, nicht durch die SDL-Sprachdefinition vorgegebene objektorientierte Begriffe im Kontext der SDL-Daten für die weitere Nutzung in dieser Arbeit zu fixieren. Bei den Begriffsdefinitionen werden gegebenenfalls Relationen zu vorhandenen SDL-Sprachkonzepten hergestellt. Die Darstellung orientiert sich an Computersprachen bzw. der Literatur (z.B. [FiAh96]). Zur Untermauerung der Konzepte gibt es eine Reihe von Beispielen in konkreten Sprachen oder, falls möglich, auch in SDL.

In einem separaten Unterabschnitt wird das in SDL-2000 verfügbare Ausnahmekonzept vorgestellt. Auf dieses, vom Autor mitentwickelte Sprachkonzept [LöSc98a] wird zurückgegriffen, um Reaktionen für auftretende Fehler einfach zu beschreiben. Die Alternative wäre eine sehr spezifische Fehlerbehandlung, die mit der Einführung von Ausnahmen bereits wesentlich allgemeiner gefaßt wurde.

Objekt: Als Objekt wird eine Zusammenfassung konkreter Werte (Attribute) im Speicher bezeichnet. Der Begriff Instanz wird als Synonym für Objekt genutzt.

Attribut: Der Begriff Attribut wird anstelle des Begriffs Wert verwendet, wenn dieser Bestandteil eines Objekts ist.

Attribute werden in vielen Computersprachen in Datenattribute und Zugangspunkte für Funktionen (Tabellen für diese Eintrittspunkte sind oft nur ein implizites Attribut) unterteilt. Einige Sprachen fassen sogar Funktionen als Objekte auf. Beispielsweise unterstützt C++ Referenzen auf Methoden. Das hat den Vorteil, daß eine Funktion wie ein Wert nutzbar ist. In SDL ist das bislang für Operationen und Prozeduren weder möglich noch in allen vorgestellten Spracherweiterungen vorgesehen. Deshalb kann man den Begriff Attribut als Synonym für Datenwert nutzen.

Attributbeschreibung: Die Definition eines Attributes ist die Attributbeschreibung.

Für eine Funktionsbeschreibungen wird auch der Begriff „Signatur“ und für Datenwerte der Begriff „Feld“ verwendet. Oft wird auch Attribut synonym für Attributbeschreibung genutzt.

Objektbeschreibung: Die Spezifikation eines Objekts mit seinen Attributbeschreibungen (Felder und Funktionenssignaturen) ist die Objektbeschreibung. Der Begriff „Typ“ wird hier nicht verwendet, da es ein allgemeines SDL-Konzept ist.

Die Integration von Daten und Funktionen ist ein wichtiges Strukturierungs- und Abstraktionskonzept von Computersprachen. Meist kann man Vererbungs- und Parametrisierungskonzepte für Objektbeschreibungen nutzen. Es gibt sogar Sprachen (z.B. Python) die eine Objektbeschreibung selbst als Objekt darstellen können.

Eine SDL-Sortendefinition ist demnach eine Objektbeschreibung.

Methode: Eine Methode ist eine Funktion zur Manipulation von Werten, die einer Objektbeschreibung zugeordnet ist. Es ist nicht ausgeschlossen, daß eine Methode auch globale Programmzustände manipuliert. Eine derartige Methode verursacht Nebeneffekte.

Operator: Für SDL ist der Begriff des Operators bereits von der ACT ONE-Semantik vorgegeben und im Abschnitt 2.1.2 „*Signaturen und Sorten*“ (S.22) erläutert worden.

In einer Zuweisung kann man mit Ergebnissen von Operatoren Werte von Variablen verändern. Innerhalb einer Operation kann nicht auf den globalen Programmzustand zugegriffen werden. Demzufolge ist ein SDL-Operator eine Methode, die keine Nebeneffekte verursacht. Zusätzlich ist das Ergebnis eines Operators durch seine Argumente fest vorgegeben.

Strukturelle Einordnung von Objekten: Objekte werden in vielen Computersprachen ausschließlich dem Datenkonzept zugeordnet.

Der Begriff des Objekts ist auch außerhalb des Datenkonzepts anwendbar. Das Prozeßkonzept von SDL ist ein derartiges Beispiel. Datenattribute sind die Werte der expliziten und impliziten lokalen Variablen, die Methoden entsprechen den Transitionen des Prozeßgraphen. Eine Methode wird gerufen, indem an die Prozeßinstanz ein Signal gesendet wird, um die Ausführung einer Transition zu erwirken. Entfernte Prozeduren kommen der eigentlichen Vorstellung von Methoden sehr nahe, semantisch werden die Prozedurrufe jedoch wieder auf Signale und implizite Transitionen abgebildet.

Falls die konzeptionelle Einordnung eines Objekts mehrdeutig ist, wird ein Konzeptpräfix genutzt, z.B. Prozeßobjekt, Datenobjekt.

Referenz: Mit einer Referenz erhält man indirekt (vermittelt über die Referenz) Zugriff auf die Attribute eines Objekts. Gibt es mehrere Besitzer einer Referenz auf ein und dasselbe Objekt, kann es bei gleichzeitigem Zugriff zu Konfliktsituationen kommen.

Für SDL-Prozeßinstanzen gibt es Referenzen: die *PId*-Werte. Unter Verwendung dieser Referenzen lassen sich gezielt Signale an Prozeßinstanzen senden. Die Verarbeitung mehrerer Signale erfolgt auch bei gleichzeitigem Eintreffen stets nacheinander. Die Beschreibung von Prozessen gehört nicht zum Datenkonzept, weshalb *PId*-Werte keine Referenzen auf Datenobjekte sind. Beschränkt man sich nur auf das SDL-Datenkonzept, gibt es kein Konstrukt für indirekte Zugriffe auf Werte per Referenz.

Strukturelle Einordnung von Referenzen: Referenzen sind in allen bekannten Sprachen den Datenwerten gleichgestellt. Man kann sie an (Referenz-) Variablen zuweisen, als (Referenz-) Parameter übergeben und sie unterliegen Zuweisungsbeschränkungen. Das Kriterium für derartige Einschränkungen sind meistens Vererbungsbeziehungen referenzierter Objektdefinitionen. Es gibt aber auch wesentlich allgemeinere Regeln.

Die Datentypdefinition einer Referenz sei in Anlehnung an SDL als Referenzsorte bezeichnet. Die Beschreibung des Objekts, auf das eine Referenz verweist, ist der dynamischer Typ der Referenz. In den meisten Sprachen gibt es auch eine eindeutige Zuordnung einer Referenzbeschreibung zu einer Objektbeschreibung. Diese ist der statische Typ einer Referenz. Fehlt diese Zuordnung, spricht man von einer ungetypten Referenz. Statischer und dynamischer Typ einer Referenz stehen in einer Relation, die von den Zuweisungsregeln der Sprache vorgegeben wird. Das führt zum Konzept der Polymorphie und der Typsicherheit.

Die SDL-Sorte *PId* ist bezüglich der Prozeßdefinitionen eine ungetypte Referenzsorte. Zwei *PId*-Werte können unterschiedlich definierte Prozeßinstanzen referenzieren. Es gibt in SDL keine Klassifizierung der *PId*-Spezifikation bezüglich unterschiedlich definierter Prozesse.

Polymorphie: Eine Zuweisung (Ziel := Quelle) wird als polymorph bezeichnet, wenn sie folgende zwei Merkmale besitzt:

- die Quelle hat eine andere statische Datentypbeschreibung als das Ziel,
- der zugewiesene Wert enthält, möglicherweise verdeckt, alle Informationen (Attribute, Bindung an die Typbeschreibung) der Quelle.

Für Referenzen ist diese Art der Zuweisung auf eine natürliche Art vorgegeben und so in den Computersprachen realisiert. Für Datenwerte, die keine Referenzen repräsentieren, werden polymorphe Zuweisungen i. Allg. nicht unterstützt. Im Abschnitt *„Vollständig Polymorph"* (S.114) wird jedoch eine derartige Variante für SDL diskutiert.

Betrachtet man *PId*-Werte als Referenzen für Prozeßobjekte, so sind *PId*-Zuweisungen polymorph bezüglich der Prozeßdefinitionen. Der *PId*-Wert jeder Prozeßinstanz läßt sich an eine *PId*-Variable zuweisen, wobei die Bindung an die Prozeßinstanz bei der Zuweisungen erhalten bleibt. Bezüglich des Datenkonzepts gibt es keine polymorphen Zuweisungen, da die Definition aller *PId*-Werte zur Sorte *PId* gehört. Insgesamt gibt es in SDL wegen des strengen Zuweisungskonzepts von SDL (die statischen Typbeschreibungen müssen bei Zuweisungen übereinstimmen) bezüglich der Datenbeschreibungen keine Polymorphie.

Bindung von Methoden: Für eine Methode kann es mehrere alternative Beschreibungen der auszuführenden Funktionalität geben. Die Auswahl kann abhängig von festgelegten Sprachkriterien mit der Spezifikation des Aufrufs (statisch) oder erst zum Zeitpunkt des Aufrufs (dynamisch, später) entschieden werden. Das sei als statische bzw. späte Bindung einer Methode bezeichnet.

Die statische Bindung von Methoden über die Signatur (Überladung) ist üblich und erfolgt auch oft in Kombination mit später Bindung.

Ein oft genutztes Auswahlkriterium für späte Bindung ist die Nutzung der Methodenbeschreibung des dynamischen Typs eines ausgezeichneten Referenzparameters. Man spricht auch vom Aufruf der Methode an dem Objekt, auf das der Parameter verweist. Das setzt natürlich ein polymorphes Zuweisungskonzept voraus.

SDL-Operatoren sind spezielle Methoden. Hier ist nur statische Bindung (Überladung) möglich. Bei einer SDL-Prozedur, gesehen als Methode eines Prozesses, ist es wesentlich komplizierter. Ein normaler Prozedurruf wird in SDL statisch an eine sichtbare Prozedurdefinition gebunden, wobei keine Überladung unterstützt wird. Es gibt jedoch zusätzlich das Konzept der entfernten Prozeduren. Ruft man eine entfernte Prozedur mit einem *PId*-Wert, dann ist erst zur Laufzeit entscheidbar, welche Prozeßinstanz den Ruf bearbeitet. Die Aktionen der für diesen Prozeß sichtbaren Prozedurdefinition werden ausgeführt. Der entfernte Prozedurruf wird also mit dem *PId*-Wert zur Laufzeit spät gebunden.

Daneben gibt es aber auch das SDL-Konzept „Virtualität“, insbesondere für Prozedurdefinitionen.

Virtualität: Das Schlüsselwort **virtual** wird in Programmiersprachen mit sehr unterschiedlichen Bedeutungen unterlegt. So gibt es in C++ mit **virtual** gekennzeichnete Methoden, die im Gegensatz zu ungekennzeichneten Methoden spät gebunden werden. In Anlehnung an die allgemeine Sprechweise wird diese Semantik im Kontext von Methoden fortlaufend angewendet. Auch C++-Klassen können mit **virtual** gekennzeichnet sein. Dieses beeinflußt das Speicherlayout von Klassen mit Mehrfachvererbung.

In SDL gilt, daß eine (Basis-) Definition, die mit **virtual** gekennzeichnet ist, „später“ durch eine Redefinition ersetzt werden kann. Alle Instanzkonstruktionen, die auf Grundlage der Basisdefinition erzeugt werden (z.B. Prozedurrufe oder Prozeßinstanzmengenbildung), beziehen sich dann rückwirkend auf die Ersetzung. Der Begriff „später“ muß weiter präzisiert werden:

a. Eine Redefinition darf nur in der Ableitung der umgebenden Struktureinheit erfolgen. Damit ist Virtualität eng an das Vererbungskonzept gekoppelt und nur innerhalb von Typen zulässig.
b. Die Ersetzung erfolgt während der Analyse der Ableitung (statisch). Nach der Analyse einer vollständigen SDL-Spezifikation steht fest, auf welche virtuelle Typdefinition sich eine Instanzkonstruktion bezieht.

Zusätzlich gilt, daß die Redefinition eine Ableitung der ursprünglichen Definition ist. Angewendet auf Prozeduren ergibt sich eine statische Bindung von Prozedurrufen trotz „virtueller“ Definitionen. Das ist für viele SDL-Nutzer mit der allgemein üblichen Vorstellung zu virtuellen Methoden eine überraschende Erkenntnis. Das Konzept der späten Bindung entfernter Prozeduren auf der Basis einer Empfänger-*Pid* ist mit virtuellen Prozedurdefinitionen formal kombinierbar. Derartige SDL-Konstruktionen richtig anzuwenden und zu durchschauen ist ein gewisses Problem für die Nutzer.

Typsicherheit: Typsicherheit ist ein Konzept zur Klassifizierung einer Sprache bezüglich auftretender Fehler, die sich aus der Verletzung der Zuweisungsregeln ergeben. Die Einhaltung der Regeln wird einerseits von der Semantikanalyse bezüglich der statischen Typbeschreibungen der beteiligten Seiten geprüft. Andererseits ist es bei polymorphen Zuweisungskonzepten möglich, daß die beteiligten dynamischen Typen nicht mit den statischen Typbeschreibungen übereinstimmen und es dynamisch zur Verletzung der Zuweisungsregeln kommen kann. Das wird als dynamischer Zuweisungsfehler bezeichnet.

Sprachen, die sicherstellen, daß statisch korrekte Zuweisungen auch während der Laufzeit keine Fehler erzeugen, seien als statisch typsicher bezeichnet. Wegen des strengen Zuweisungskonzepts und fehlender Polymorphie ist SDL statisch typsicher bezüglich der Datentypen. Das ist für Spezifikationstechniken eine wünschenswerte Eigenschaft.

Für alle Sprachen mit polymorphen Zuweisungen gibt es keine Garantie zur fehlerfreien Ausführung von Zuweisungen. Es kann aber Sprachkonzepte geben, die eine definierte Reaktion bei allen regelwidrigen Zuweisungen ermöglichen. Das sei als dynamisch typsicher bezeichnet. Beispielsweise wird in Java bei dynamischen Zuweisungsfehlern stets eine Ausnahme geworfen.

Sprachen, deren Reaktion auf dynamische Zuweisungsfehler undefiniert ist, sind nicht typsicher. C und C++ sind populäre Vertreter solcher Sprachen. Nach fehlerhaften Zuweisungen stürzen Programme auf einigen Systemen ab, auf anderen funktionieren sie.

In SDL kann man die Typsicherheit von *PId*-Referenzen auf Prozeßinstanzen untersuchen. Formal wird es nach Feststellung statischer Korrektheit auch keine dynamischen Zuweisungsfehler oder undefiniertes Verhalten für *PId*-Werte geben, SDL ist also auch an dieser Stelle statisch typsicher.

Laufzeittypinformation: Ein Attribut, daß die Bestimmung des dynamischen Typs eines Objekts zuläßt, nennt man eine Laufzeittypinformation, oder besser bekannt unter der Abkürzung RTTI (*run time type information*). In einigen Sprachen ist diese Information einem Datentyp zugeordnet, so daß ein Typvergleich auf einen Wertevergleich abgebildet werden kann.

Eine Laufzeittypinformation ist die Basis der dynamischen Typsicherheit.

Speicherverwaltung: Zur Implementierung eines beliebigen Datenkonzepts benötigt man für die Repräsentation von Werten Speicher. Da dieser auf realen Computern begrenzt ist, muß man die Wiederverwendung von Speicherbereichen organisieren. Das ist die Aufgabe einer Speicherverwaltung.

Ein Speicherbereich kann wiederverwendet werden, wenn vom Programmablauf sichergestellt wird, daß auf enthaltene Werte nicht mehr zugegriffen wird. Das stellt in Sprachen mit Referenzen ein schwieriges Problem dar. Man weiß i. Allg. nicht, ob ein Wert im Speicher vom Programm noch referenziert wird. Computersprachen sind bezüglich der Lösung dieses Problems katalogisierbar:

Explizite Speicherverwaltung: Das Programm garantiert explizit, daß das ein Wert nicht mehr genutzt wird. Mit einem speziellen Konstrukt „löscht" man den Wert bzw. dessen Referenz. Diese Variante führt zu dynamischen Speicherfehlern wie Mehrfachfreigabe oder Zugriff nach Freigabe. Diese Fehler können mit entsprechendem (Laufzeit-) Aufwand dynamisch erkannt werden. C++ ist eine Sprache mit expliziter Speicherverwaltung.

Implizite Speicherverwaltung: Das Laufzeitsystem des Programms organisiert implizit eine Buchführung für Referenzen, die zur automatischen Erkennung ungenutzter Speicherbereiche genutzt wird. Die schnelle und vollständige Freigabe von Speicherbereichen läuft der Programmeffizienz entgegen. Deshalb sind automatische Verfahren für laufzeiteffiziente Anwendungen nicht unproblematisch. Java ist ein bekannter Vertreter der Sprachen mit ausschließlich impliziter Speicherverwaltung.

Die entgültige, möglicherweise zeitlich unbestimmte Freigabe eines Objekts kann eine Aktion triggern (Aufruf einer Methode). Diese Aktion nennt man allgemein Finalisierung, die entsprechenden Methoden zur Finalisierung meist Destrukturen.

In SDL als Spezifikationstechnik ist es üblich, einen unendlich großen Speicher vorauszusetzen. So werden für Prozeßinstanzen stets neue *PId*-Werte vergeben. Sendet man ein Signal an einen beendeten Prozeß, so wird dieses verworfen und nicht zufällig einer anderen Prozeßinstanz zugestellt. Das kann aber auch durch eine Implementierung mit endlichem Speicher erreicht werden, d.h. Speicherverwaltung ist ein Implementierungsdetail von SDL-Werkzeugen. Es gibt kein Finalisierungskonzept für dynamische SDL-Elemente.

6.1.1 Ausnahmebehandlung (SDL-2000)

Ausnahmen bilden ein neues Kontrollflußkonzept für SDL. Kommt es während eines Zustandsübergangs explizit oder implizit zur Erzeugung einer Ausnahme, so wird die Ausführung der Aktionen an dieser Stelle abgebrochen und eine Behandlungsroutine gesucht. Ausnahmen innerhalb einer Prozedur oder Operators, die nicht intern abgefangen werden, erzeugen eine Ausnahme beim Ruf der Prozedur bzw. des Operators. Diese Weiterleitung von Ausnahmen umfaßt auch entfernte Prozeduren: die Ausnahme wird sowohl an den *server*-Prozeß als auch den entfernten Rufer weitergeleitet. Wenn die Suche nach einer Ausnahmebehandlung den Service oder Prozeß erreicht und keine Behandlung gefunden wurde, ist das weitere Verhalten des SDL-Programms undefiniert.

Die Definition einer Ausnahme erhält einen Namen und optional eine Parameterliste. Ausnahmen können auch als Kontextparameter vereinbart werden. Instanzen kann man mit einer *raise*-Konstruktion (vergleichbar mit einer Signalausgabe) erzeugen (werfen) und mit *handle*-Konstruktionen (vergleichbar mit einem Signalempfang) abfangen. Im Gegensatz zur Signalausgabe ist die Beschreibung des Zustandsübergangs nach einer expliziten Ausnahmeerzeugung beendet, da die Ausnahmebehandlung den Zustandsübergang fortsetzt. Innerhalb einer Ausnahmebehandlung kann mit der Konstruktion

raise -;

die gerade behandelte Instanz mit allen Parametern noch einmal geworfen werden. Mehrere *handle*-Konstruktionen bilden zusammengefaßt eine Ausnahmebehandlung. Die Definition einer Ausnahmebehandlung folgt den Prinzipien der Syntax und der statischen Semantik einer Zustandsdefinition, nur daß andere Schlüsselwörter verwendet werden (**exceptionhandler** - **state** bzw. **handle** - **input**). Insbesondere gibt es virtuelle *handle*-Konstruktionen, und Notationen mit '*', um mehrere Ausnahmebehandlungen oder Ausnahmen zusammenzufassen. Die Aktivierung der Ausnahmebehandlung kann mehrstufig durch Assoziierung mit sechs Lokalitätsstufen eines Verhaltensgraphen erfolgen.

1. Verhaltensgraph (inklusive der Vererbungsstruktur)
2. Grundzustand (alle Zustandsübergänge über die gesamte Vererbungsstruktur)
3. alle Aktionen eines Zustandsübergangs bzw. der start-Transition
4. Ausnahmebehandlung (alle *handle*-Konstruktionen über die gesamte Vererbungsstruktur)
5. alle Aktionen zum Abfangen einer speziellen Ausnahme
6. eine Aktion während eines Zustandsübergangs oder einer Ausnahmebehandlung

SDL-2000 wird weitere Lokalitätsstufen für algorithmische Notationen und verschachtelte Zustände unterstützen. Die Suche nach einer Ausnahmebehandlung erfolgt mit abnehmender Lokalität in den vorgegebenen Lokalitätsstufen. Die Assoziation, die zur Ausnahmebehandlung führt, wird während der Behandlung deaktiviert.

Assoziationen an Zuständen und Ausnahmehandlungen gelten auch in Ableitungen von Basistypen und umgekehrt. Bild 18 demonstriert die grundlegenden Ausnahmekonzepte in grafischer Form.

Vordefinierte Ausnahmen

Gegenwärtig kann es während der Ausführung eines Zustandsübergangs zu Situationen kommen, in denen das weitere Verhalten des SDL-Programms undefiniert ist. Dazu gehört der Zugriff auf nicht initialisierte Variablen oder die Bewertungen des Fehlerterms. Wird bei derartigen Situationen eine vordefinierte Ausnahme geworfen, so ist eine Beschreibung der Reaktion mit SDL und damit eine definierte Fortsetzung des SDL-Programms möglich. Findet sich keine Behandlung, ist das weitere Systemverhalten wiederum undefiniert.

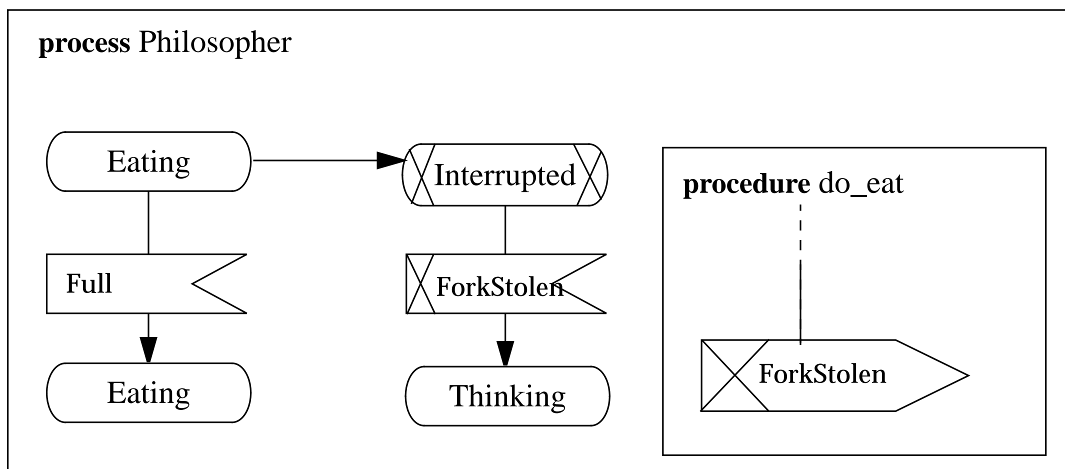

Bild 18: Beispiel einer Spezifikation mit Ausnahmen

Neben diesen SDL-spezifischen vordefinierten Ausnahmen sind auch werkzeugspezifische Ausnahmen denkbar, die sich aus dem Einsatz eines SDL-Programms in einer Laufzeitumgebung ergeben, z.B. für überfüllte Empfangspuffer von Prozessen.

In SDL-2000 gibt es für nicht vermeidbare, dynamische Fehler vordefinierte Ausnahmen. In dieser Arbeit wird eine vordefinierte Ausname *InvalidReference* für die Erweiterung von SDL um ein Referenzkonzept genutzt.

Ausnahmen kombiniert mit ACT ONE

Ausnahmen sind nicht Bestandteil des ACT ONE-Modells. Es gibt sie nur an den Schnittstellen zur Sprache SDL:

- Nur in algorithmischen Operatoren kann das Werfen einer Ausnahme spezifiziert werden. Für Axiome gibt es keine adäquate Syntax. Die Auswertung algorithmischer Operatoren unterliegt jedoch einer prozeduralen Semantik.
- Das Feststellen, ob eine Variable an einen Wert gebunden ist, erfolgt vor der algebraischen Bestimmung des Terms.

Das Ausnahmekonzept ist also nur für Ausdrücke, jedoch nicht für Terme anwendbar. Folglich gibt es ein Problem, wenn Ausnahmen für Fehlerterme zu werfen sind. Ein Beispiel wäre die Reaktion auf einen falschen Indexzugriff. Ein Umbau von axiomatisch beschriebenen Zugriffsoperationen auf algorithmische ist denkbar.

6.2 Transformationsbasierte Datenerweiterungen

Im Folgenden werden zwei konkrete Erweiterungsvarianten von SDL mit semantischer Transformation auf vorhandene SDL-Konzepte untersucht. Aus der Sicht des Anwenders integriert die erste Variante lediglich ein Referenzmodell in das vorhandene Datenkonzept. Sie wird als Sammelobjektverwaltung bezeichnet. Die zweite Variante, die Einzelobjektverwaltung, wird neben den vorhandenen Datendefinitionen eine alternative Methode zur Spezifikation von Datenobjekten mit entsprechenden Vererbungs- und Referenzkonzepten bereitstellen. Der zentrale Gedanke hinter beiden Varianten ist die Nutzung von Referenzen in SDL.

Die Auswahl dieser Erweiterungsvarianten ist nicht zufällig. Sie ergibt sich direkt aus der systematischen Betrachtung der SDL-Konzepte mit denen sich Referenzkonzepte für Daten mit entsprechendem Aufwand nachbauen lassen. Dieser modellbasierte Start einer SDL-Weiterentwicklung ergibt sich aus der SDL-Historie. Es gab keine allgemein akzeptierten, konkreten Konzeptvorgaben bezüglich der Verbesserung der objektorientierten Konzepte für Daten. Deshalb wird auch in dieser Arbeit der Fokus nicht auf den Ausbau syntaktischer Feinheiten gelegt, sondern vielmehr auf die Auslotung der potentiellen Möglichkeiten zur Spracherweiterung mit semantischer Transformation.

Vor der Diskussion der beiden konkreten Erweiterungsvarianten werden zur Erleichterung des Modellverständnisses allgemeine Überlegungen angestellt, wie sich Referenzen mit SDL darstellen lassen. Die Einführung in die zwei Varianten erfolgt mit konkreten, konzeptionellen Entscheidungen. Auf diese Art können Beispiele mit willkürlich festgelegter Syntax angegeben werden. Die Flexibilität für Einzelkonzepte wird in separaten Unterabschnitten, jeweils für die Grundvarianten und in einem weiteren für beide, mit nicht systematisch orientierten Ausbauvarianten untersetzt. Die fehlende Systematik begründet sich im Fehlen von konkreten Erweiterungsvorgaben durch die SDL-Anwender und Sprachentwickler. Mit SDL-2000 sind zwar implizit derartige Vorgaben vorhanden, diese sind jedoch nach der Entwicklung der Varianten entstanden und werden auch mit vorhandenen SDL-2000-Werkzeugen kritisch zu hinterfragen sein. Konzeptionelle Bezüge in Richtung SDL-2000 sind jeweils angegeben.

Eine kurze Einschätzung und einige Bemerkungen zum historischen Verlauf der Entwicklung runden diesen Abschnitt ab.

6.2.1 Prozesse für Referenzen auf Werte

Viele objektorientierte Konzepte gehen von der Existenz eines Referenzkonzepts für Werte aus. Da genau dieses Konzept für SDL-Daten nicht zur Verfügung steht, kann man die Daten nicht so nutzen, wie man es aus anderen objektorientierten Sprachen erwarten würde. Demzufolge liegt es nahe sich zu überlegen, wie Referenzen mit vorhandenen SDL-Konzepten darstellbar sind.

Datenwerte kann man zur Ausführungszeit einer Spezifikation nur an Variablen binden. Deshalb wird man zu referenzierende Werte direkt oder geeignet verpackt in Variablen speichern. Da Services und Prozeduren lediglich eine Strukturierungsvariante für den Prozeßgraphen sind, kann man sich auf die Betrachtung von Prozeßvariablen beschränken.

Geht man davon aus, daß Werte von Referenzen über Prozeßgrenzen hinweg erreichbar sind, muß der Zugriff auf die Variablen mit SDL-Interprozeßkommunikation realisiert werden. Aus Gründen der einfachen Darstellung ist es günstig, auf ein explizites Signalspiel zu verzichten und entfernte Prozeduren als mehr abstraktes SDL-Konzept zu nutzen. Man kann prinzipiell auch alternative Kommunikationskonzepte (z.B. entfernte Variablen) wählen, die Transformation der konkreten Syntax für Referenzen wäre entsprechend zu ändern.

Eine weitere, fundamentale Annahme ist, daß jedes neue Datenkonzept für Laufzeitfehler ein wohldefiniertes Verhalten besitzt. Das im Abschnitt 6.1.1 „*Ausnahmebehandlung (SDL-2000)*" (S.90) beschriebene Konzept stand mit dem Beginn der Untersuchung zur Verfügung. Es gab sogar eine Implementierung mit den SITE-Werkzeugen, eingeschränkt auch SDL-96. Deshalb und auch mit Blick auf in SDL-2000 vorhandene Grundkonzepte wird für die Modellbildung auf das Ausnahmekonzept für die Reaktion auf dynamische Fehler zurückgegriffen.

Der SDL-Anwender wird eine erweiterte SDL-Syntax für Referenzen, bzw. etwas allgemeiner für Objekte verwenden können. Die erweiterte SDL-Semantik gibt Transformationsregeln für die Referenzkonzepte vor. In der Sprechweise von SDL sind die Referenzen dann ein Modell. Dieses Modell basiert auf Prozessen mit Variablen für die Werte der Objekte und entfernten Prozeduren für den Zugriff. Bild 19 verdeutlicht diesen semantischen Ansatz. Die Verwaltung der Objekte mit Prozessen führt zu zwei alternativen Modellansätzen:

```
dcl   iObj IntegerObjectType, /* Spracherweiterung: Referenztyp für Integer */
      i Integer;
...
task  iObj   := create(42),   /* Spracherweiterung: Objekterzeugung */
      i      := ^iRef;        /* Spracherweiterung: Objektzugriff */
```

Nutzersicht

prinzipielle Modellsicht

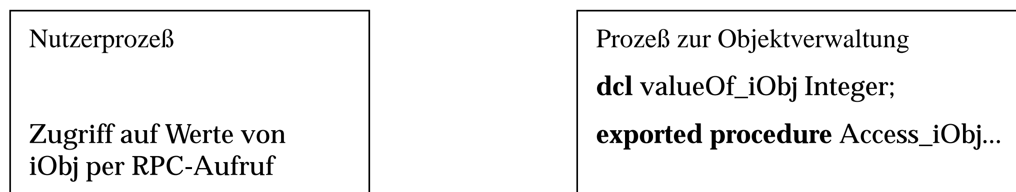

Bild 19: Referenzen mit Prozeßvariablen

Sammelobjektverwaltung: Die Werte aller erzeugten Datenobjekte werden von einer zentralisierten Prozeßinstanz gesammelt und verwaltet. Diese Instanz wird als Speicherprozeß bezeichnet. Konsequenterweise muß für den Zugriff auf Werte von Objekten ein Identitätsmerkmal der internen Speichervariable genutzt werden.

Einzelobjektverwaltung: Für jedes erzeugte Objekt ist genau eine Verwaltungsprozeßinstanz zuständig. Demzufolge ist der Zugriff auf den Wert eines Objekts über die Identität des jeweiligen Verwaltungsprozesses, also seine *PId*-Adresse möglich. Im Hinblick auf eine leichtgewichtige Implementierung, werden diese Prozeßinstanzen als Objektreferenzen bezeichnet.

Unabhängig von der Realisierung der einen oder anderen Variante sind einige prinzipielle Fragen zu beantworten. Diese klären grundsätzliche Eigenschaften des jeweiligen Referenzdatenmodells.

Konstruktion: Wie erzeugt man auf Grundlage des Referenzkonzepts neue Werte?

Zugriff: Wie erfolgt der lesende und schreibende Zugriff auf die von Referenzen verwiesenen Objekte aus der Sicht des Nutzers?

Vererbung: Welche Relation besteht zwischen Vererbung und Referenzen?

Zuweisungen: Welche Regeln gelten für Zuweisungen bei Referenzen?

Methoden: Wie passen sich die Operatoren von Datentypen in das Referenzkonzept ein und wie läßt sich späte Bindung realisieren?

Speicherverwaltung: Wie werden erzeugte Referenzen wieder freigegeben?

Die Unterschiede zwischen der Einzel- und Sammelobjektverwaltung werden anhand von Beispielen verdeutlicht, die sich auf zwei Datentypdefinition *Base* und *Sort* stützen. *Sort* ist dabei eine Ableitung von *Base*. Die konkrete, exemplarische Syntax der Datendefinition und die Verwendung der Werte unterscheidet sich in den einzelnen Varianten erheblich.

6.2.2 Sammelobjektverwaltung

Die Idee der Sammelobjektverwaltung besteht in der Realisierung eines zentralisierten Speichers für Objekte. Dabei ist es ohne Bedeutung, wie die Objektdefinitionen untereinander assoziieren. Es ist aber sinnvoll, für Objektdefinitionen, die in einer Vererbungsrelation stehen, ein jeweils separates, universell adressierbares Speicherfeld anzulegen. Mit dieser Trennung ergeben sich implizit einige Zuweisungsbeschränkungen für Referenzen, die auch von anderen objektorientierten Sprachen bekannt sind.

Der Speicherprozeß kann als systemweit eindeutige SDL-Prozeßinstanz ohne konkreten *PId*-Wert adressiert werden. Die Identifikation des Speicherplatzes muß über einen Parameter des jeweiligen Zugriffs erfolgen. Bild 20 demonstriert das Zusammenspiel des Speicherprozesses

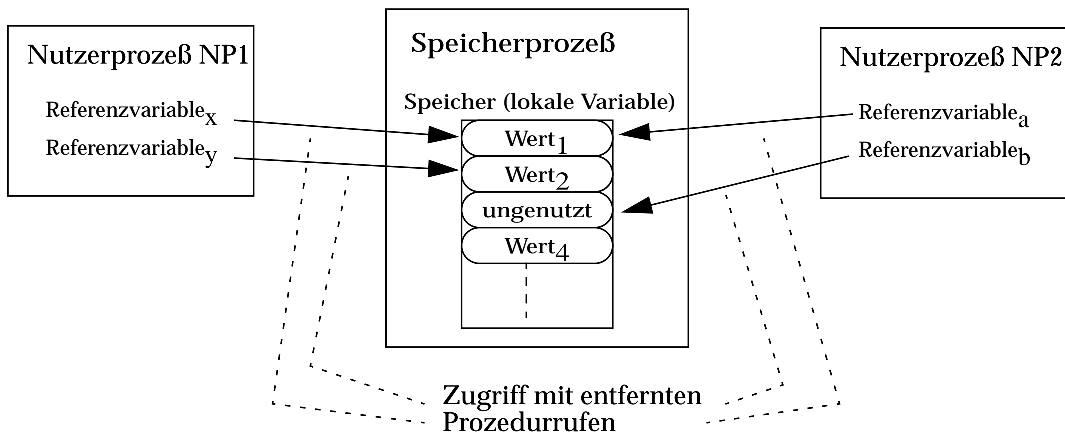

Bild 20: Referenzen im Speicherprozeß

mit zwei Nutzerprozessen, wobei typische Situationen für Referenzen dargestellt sind:

- Wert$_1$ kann von beiden Prozessen NP1 und NP2 manipuliert werden,
- Wert$_2$ nur vom Prozeß NP1, da Prozeß NP2 die Referenz nicht kennt,
- es gibt eine ungenutzte Speicherstelle, deren möglicher Zugriff von Prozeß NP2 zu einem dynamischen Fehler führt, und
- der Wert$_4$ wird von keinem Prozeß referenziert, er kann gelöscht werden, wenn in der Gesamtspezifikation keine Referenz auf diesen Wert mehr existiert.

Die Spezifikation des Speicherprozesses ergibt sich durch die semantische Transformation der konkreten Syntax aller Datenbeschreibungen des SDL-Systems. Für den SDL-Nutzer ist der Speicherprozeß nicht sichtbar, er kann mit geeigneten Syntaxerweiterungen einfach Referenzen verwenden. Um die Beziehung einer Sorte zu seiner Referenzsorte darzustellen, wird der Name der Referenzsorte aus dem Namen der Sorte und dem Präfix "^"[1] gebildet.

Beispiel:

newtype Base...	**newtype** Sort **inherits** Base;
literals 1, ...;	**operators all**; **adding literals** 2,....:
endnewtype;	**endnewtype**;

1. "^" ist ein gültiges Zeichen für SDL-Namen. Für die konkrete Syntax sind andere Varianten zur Kennzeichnung von Referenzen denkbar.

```
process NP1;
    dcl x ^Base, y ^Sort;
    dcl b Base, s Sort;
    ...
    task x := new Sort(0), y := new Sort(42);
    task 'Übergabe von x an NP2';
    ...
    task s := ^y;
endprocess;

process NP2;
    dcl a ^Base, b ^Sort;
    ...
    task 'Empfang von x aus NP1 mit a';
    task b := new Sort(^a);
    ...
endprocess;
```

O.B.d.A. kann man die Definition des Speicherprozesses auf zwei Datendefinitionen mit Vererbungsbeziehung (*Base* und *Sort*) einschränken. Für jede weitere Vererbungshierarchie gibt es lediglich ein weiteres Speicherfeld, neue Datendefinitionen innerhalb der Hierarchie implizieren einen weiteren Satz von Schnittstellen, die im Folgenden spezifiziert werden. Da eine konkrete SDL-Spezifikation des Speicherprozesses eingeschränkt auf zwei Objektdefinitionen immer noch umfangreich ist, werden vorab die konkreten Entwurfsideen vorgestellt:

❍ Es ist wegen des strengen SDL-Zuweisungskonzepts nicht möglich, Werte verschiedener Sorten direkt in einer universellen Speicherstelle abzulegen. Deshalb muß für die Speicherstelle eine Sorte definiert werden, die aus allen beteiligten Sorten eine Vereinigung bildet. Das läßt sich in ASN.1 adäquat mit einer CHOICE-Konstruktion beschreiben. Die algebraische Umschreibung dieses Konzepts ist in [Z.105] als eine komplexe Transformation angegeben. Um die Spezifikation nicht mit diesen Zusatzkonzepten zu belasten, wird die ASN.1-Notation beibehalten, obwohl das Ziel eigentlich eine Transformation auf SDL-Konzepte ist.

❍ Der Datentyp des universellen Speicherfelds ist eine SDL-*array*-Konstruktion, deren Index die Referenz auf den entsprechenden Wert (die CHOICE-Konstruktion) repräsentiert. Für jede Sorte *Sort* gibt es einen Bezeichner ^*Sort*, der auf die Definition der Indexsorte verweist. Damit ist ^*Sort* die Referenzsorte von *Sort*.

❍ Der Speicher ist eine Prozeßvariable des *array*-Datentyps, die mit leeren Einträgen (*empty*) initialisiert wird.

❍ Für den Zugriff auf die Speichervariable gibt es pro Datendefinition, hier exemplarisch für *Base* einen Satz von entfernten Prozeduren.

```
remote procedure Base_MAKE;
    fpar in Base;
    returns ^Base;

remote procedure Base_EXTRACT;
    fpar in ^Base;
    returns Base;
    raise InvalidReference;

remote procedure Base_MODIFY;
    fpar in ^Base, in Base;
    raise InvalidReference;

remote procedure Base_VERIFY;
    fpar in ^Base; returns Boolean;
```

Da gemäß der SDL-Sprachdefinition die Signatur von Prozeduren nicht zur Unterscheidung von Prozeduren genutzt werden kann (keine Überladung von Prozeduren), erhalten die Prozedurnamen einen eindeutigen Präfix, hier den Namen der Sorte[1]. Das Löschen von Objekten ist unabhängig vom dynamischen Typ des Objekts möglich. Deshalb kann diese Schnittstelle universell, d.h. ohne Angabe des Sortennamens für alle Sorten der Vererbungshierarchie bereitgestellt werden[2].

```
remote procedure DELETE;
    fpar in ^Base;
    /* Signatur setzt Kompatibilität von ^Base zu ^Sort voraus */
```

Die für dynamische Fehler zuständige Ausnahme muß im Modul *Predefined* zusätzlich bereitgestellt werden:

```
exception InvalidReference;
```

- Ein Objekt wird erzeugt, indem ein freier Speichereintrag ermittelt wird und ein gegebener Wert nach der Konvertierung in ein Speicherelement an diese Stelle kopiert wird. Der Index auf den ermittelten Eintrag wird die neue Referenz. Demzufolge werden Referenzen auf vorhandene Werte[3] von diesem Modell nicht unterstützt.
- Ein Objekt kann per *Array*- Indexzugriff gelesen als auch neu gesetzt werden, wobei jeweils eine lesende und eine schreibende entfernte Prozedur als Schnittstelle zur lokalen Speichervariablen eingesetzt wird.
- Ein Objekt bzw. der gespeicherte Wert kann durch einen Leereintrag (*empty)* gelöscht werden. Alle Versuche, *empty* per Referenz zu lesen oder zu manipulieren, werden einen dynamischen Fehler erzeugen.

Spezifikation der Transformationsziele für *Base* und *Sort*

Im Folgenden werden die Transformationsziele, d.h. die entsprechenden SDL-Konstrukte, für die Sorten *Base* und *Sort* in drei Schritten angegeben.

1. Definition der implizit eingeführten Datentypen:
   Für jede Sorte gibt es implizit eine global sichtbare Referenzsorte. Außerdem muß der Datentyp des Speichers geeignet definiert werden.
2. Aufbau des Speicherprozesses:
   Auf Grundlage der nun vorhandenen Datendefinitionen kann der Speicherprozeß spezifiziert werden. Die exportierten Prozeduren, die Schnittstelle für Referenzen, bleiben an dieser Stelle noch offen.
3. Definition der exportierten Prozeduren:
   Es werden die konkreten Realisierungen der exportierten Prozeduren angegeben.

Schritt 1: Referenzen sind letztendlich Werte der Indexsorte der *Array*-Definition des Speichers. Es muß also eine Sortendefinition für Referenzen geben. Alle Referenzsorten haben eine gemeinsame, im Modul *Predefined* vordefinierte Basissorte *REFERENCE*, die ähnlich wie die vordefinierte Sorte *PId* unendlich viele Werte mit Hilfe einer Nachfolgeroperation erzeugen kann.

---

1. Eigentlich muß der voll qualifizierte Name als Präfix genutzt werden, darauf wird aber an dieser Stelle aus Gründen der Lesbarkeit verzichtet.
2. Formal muß man die DELETE-Prozeduren pro Vererbungshierarchie alternativ benennen. Auch auf diese Feinheit der Transformation wird aus Gründen der Lesbarkeit verzichtet.
3. z.B. Werte, die an Variablen, Felder von Strukturen oder Indexelemente gebunden sind

```
newtype REFERENCE
    literals Null;
    operators New : REFERENCE -> REFERENCE;
    axioms
        for all r in REFERENCE (New(r) /= Null;);
        for all r1,r2 in REFERENCE (New(r1) = New(r2)  ==  r1 = r2;);
endnewtype;
```

Eine konkrete Referenzsorte für eine beliebige in der Spezifikationen definierte Sorte wird nach folgenden Transformationsregeln erzeugt.

a. Ist eine Sorte *Base* ohne Nutzung von Vererbung definiert, so ist *^Base* eine Ableitung von *REFERENCE*:

```
newtype ^Base inherits REFERENCE
    operators all; adding default Null;
endnewtype;
```

Referenzen von Sorten ohne Vererbungsbeziehung sind verschiedene Sortendefinitionen und damit nicht zuweisungskompatibel.

b. Enthält die Definition der Sorte *Sort* eine Vererbungskonstruktion mit *Base* als Basissorte, dann ist *^Sort* definiert als:

```
syntype ^Sort = ^Base
endsyntype;
```

Gemäß dem Modell für SDL-*syntype*-Konstruktionen sind *^Sort* und *^Base* oder verallgemeinert alle Referenzen von Sorten mit Vererbungsbeziehung uneingeschränkt zuweisungskompatibel.

Die Definition der Elemente des Speichers für alle beteiligten Sorten erfolgt mit ASN.1:

```
ELEM ::= choice {
    empty       NULL,
    elemBase    Base,
    elemSort    Sort
}
```

Schließlich muß noch der Speicher selbst definiert werden:

```
/*  Zur besseren Lesbarkeit wird der Name ADDR als
    syntype-Name für alle Referenzen eingeführt.
*/
syntype ADDR = ^Base /* = ^Sort */ endsyntype;

newtype STORAGE Array(ADDR, ELEM);
endnewtype STORAGE;
```

Die Spezifikation von *ELEM* und *STORAGE* kann in den Speicherprozeß verschoben werden, es gibt keinen Grund für eine globale Sichtbarkeit.

Schritt 2: Nach der Erklärung aller Datentypen wird der Speicherprozeß beschrieben. Die Definition des Speicherprozesses ist typbasiert. Von dieser Typdefinition wird eine Prozeßinstanzmenge mit einem initialen Prozeß - dem eigentlichen Speicherprozeß - gebildet.

```
process type StorageProcessType;

    /* Der Speicher wird mit der Variante empty global initialisiert. */
    dcl     storage     STORAGE     := (. empty:NULL .);

    /* Diese Hilfsvariable zeigt immer auf die letzte belegte Speicherstelle. */
    dcl     free_ref  ADDR  := Null;
```

```
/* Speicherverwaltung: es gibt unbegrenzt neuen Speicher. */
procedure NEW;
    returns ADDR;
    start;
        task free_ref := New(free_ref); /* Operator New von REFERENCE */
        return free_ref;
endprocedure;

/* Schnittstelle von Base */
exported procedure Base_MAKE  referenced;
exported procedure Base_EXTRACT  referenced;
exported procedure Base_MODIFY  referenced;
exported procedure Base_VERIFY referenced;

/* Schnittstelle von Sort */
exported procedure Sort_MAKE  referenced;
exported procedure Sort_EXTRACT  referenced;
exported procedure Sort_MODIFY  referenced;
exported procedure Sort_VERIFY referenced;

exported procedure DELETE referenced;

/*  Der Prozeß erlaubt den Zugriff auf den lokalen Speicher ausschließlich durch
    entfernte Prozeduren. Dynamische Fehler (Ausnahmen) werden an den zugreifenden
    Prozeß weitergeleitet, die zusätzliche Propagierung dieser Ausnahmen an den
    lokalen Prozeßgraphen wird ignoriert.
*/
start;
    nextstate running;

state running; onexception Ignore;

exceptionhandler Ignore;
    handle *; nextstate -;
endprocess type;

process StorageProcess(1,1) : StorageProcessType;
```

Schritt 3: Schließlich kann man sich dem Zugriff auf den lokalen Speicher von außen zuwenden und die **referenced**-Prozeduren nachreichen. Die Implementierung dieser Prozeduren ist für die Sorten *Base* und *Sort* jeweils angegeben.

Implementierung von ..._MAKE: Die Spezifikation zur Erzeugung einer Referenz ist für verschiedene Sorten auf Prozeduren mit verschiedenen Signaturen, die sich aber inhaltlich nicht unterscheiden, aufgeteilt:

```
exported procedure Base_MAKE;
    fpar in b Base; returns ^Base;
    start;
        task storage(call NEW)!elemBase := b;
        return free_ref;
endprocedure;

exported procedure Sort_MAKE;
    fpar in s Sort; returns ^Sort;
    start;
        task storage(call NEW)!elemSort := s;
        return free_ref;
endprocedure;
```

Implementierung von ..._EXTRACT: Die Definitionen für den Zugriff auf den Wert sind ebenfalls einfach

```
exported procedure Base_EXTRACT;
    fpar in ref ^Base; returns Base; raises InvalidReference;
    start;
        decision elemBasePresent(storage(ref));
            (True): return storage(ref)!elemBase;
            (False): raise InvalidReference;
        enddecision;
endprocedure;

exported procedure Sort_EXTRACT;
    fpar in ref ^Sort; returns Sort; raises InvalidReference;
    start;
        decision elemSortPresent(storage(ref));
            (True): return storage(ref)!elemSort;
            (False): raise InvalidReference;
        enddecision;
endprocedure;
```

Der Speicherzugriff testet, ob die angeforderte Variante im Speicher abgelegt wurde. Ist die aktuelle Variante *empty*, dann wurde mit einer ungültigen Referenz zugegriffen. Es kann auch vorkommen, daß die im Speicher verfügbare Variante nicht mit der geforderten Variante übereinstimmt. Das ist ein dynamischer Datentypfehler. Beide Fehlerarten werden hier mit einer wohldefinierten Ausnahme behandelt, andere Modellvarianten sind denkbar.

Implementierung von ..._MODIFY: Der referenzierte Wert kann auch modifiziert werden. Mindestens der Zugriff auf eine ungültige Referenz sollte abgefangen werden:

```
exported procedure Base_MODIFY;
    fpar in ref ^Base, in b Base; raises InvalidReference;
    start;
        decision storage(ref)!present;
            (empty): raise InvalidReference;
            else : task storage(ref)!elemBase := b; return;
        enddecision;
endprocedure;

exported procedure Sort_MODIFY;
    fpar in ref ^Sort, in s Sort; raises InvalidReference;
    start;
        decision storage(ref)!present;
            (empty): raise InvalidReference;
            else : task storage(ref)!Sort := s; return;
        enddecision;
endprocedure;
```

Implementierung von ..._VERIFY: Eine Form der Datentypabfrage wird durch die VERIFY-Prozeduren realisiert, es sind auch Modellvarianten denkbar, die Aussagen zu Vererbungsrelationen treffen.

```
exported procedure Base_VERIFY;
    fpar in ref ^Base; returns Boolean;
    start;
        return elemBasePresent(storage(ref));
endprocedure;
```

```
exported procedure Sort_VERIFY;
    fpar in ref ^Sort; returns Boolean;
    start;
        return elemSortPresent(storage(ref));
endprocedure;
```

Diese Prozeduren realisieren eine in der SDL-Spezifikation nutzbare Laufzeittypinformation.

Implementierung von DELETE:

```
exported procedure DELETE;
    fpar in ref ^Base;
    start;
        task storage(ref)!empty := NULL; return;
endprocedure;
```

Diese Variante erlaubt mehrfaches Löschen des Werts einer Referenz.

Obwohl die Sammelobjektverwaltung relativ einfach aufgebaut ist, sind einige Punkte bei der Formulierung formaler Transformationsregeln zu beachten:

- Die globale Nutzung von Referenzen setzt voraus, daß der Speicherprozeß global erreichbar ist und viele seiner Datendefinitionen sogar global sichtbar sind. In tieferen Hierarchiestufen definierte Daten, sind für den Speicherprozeß eigentlich nicht sichtbar. Formal muß man also alle Datendefinitionen auf die Definitionsstufe der Modelldefinitionen ziehen. Das ist für Modellbetrachtungen kein prinzipielles Problem.
- Möglicherweise muß die Definition der entsprechend notwendigen Prozeßinstanzmenge an eine Stelle verschoben werden, an der sie auch zulässig ist. Beispielsweise werden Datentypen oft in Modulen definiert.
- Namen, konstruiert nach dieser Variante, sind nicht in allen Fällen eindeutig, da möglicherweise gleiche Namen von hierarchisch definierbaren Datenstrukturen auf einem Sichtbarkeitsniveau verwendet werden. Man muß eigentlich voll qualifizierte Namen zur Bildung der neuen Namen nutzen.

Mit Hilfe der gegebenen Definitionen lassen sich jetzt geläufige objektorientierte Konzepte nachbilden. Exemplarisch vorgegebene Syntaxkonstrukte sind geeignet auf die gegebenen Schnittstellen des Speicherprozesses abzubilden. Die Beispiele orientierten sich an den Definitionen des Nutzerprozesses NP1 auf Seite 94.

Konstruktion

Für die Konstruktion einer Referenz sollte eine adäquate Syntax genutzt werden, z.B.

```
task    x := new Base(b);
```

Die semantische Transformation erzeugt die Aktion:

```
task    x := call Base_MAKE(b);
```

Die implizite Prozeßvariable **sender** sollte wieder den ursprünglichen Wert nach der Ausführung der *call*-Aktionen enthalten. Diese Grundvariante sieht keine nutzerdefinierte Konstruktornotation vor.

Es ist eine sehr wichtige Eigenschaft dieses Modells, daß die Referenzen stets auf Kopien der übergebenen Werte verweisen. Man kann auf diese Art keine Referenzen auf lokale Variablen erzeugen. Das ist eine sinnvolle Einschränkung, um Probleme mit der Lebensdauer temporärer Objekte (z.B. lokale Variablen von Prozeduren) zu umgehen.

Zugriff auf Werte von Referenzen

Der formale Zugriff auf den Wert einer Referenz wäre

task s:= **call** Sort_EXTRACT(y);
call Sort_MODIFY(y,s);

Mit einem neuen Präfixoperator "^", dessen Vorrang dem des "not"-Operators entspricht, kann der Zugriff „nutzerfreundlich“ gestaltet werden. Für die Sorte *Sort* ist die Signatur des Operators

operators "^": ^Sort -> Sort;

Für die Modifikation des Werts einer Referenz muß die Möglichkeit für den Zugriff auf den Wert einer Referenzvariablen geschaffen werden, z.B. mit

<reference variable> ::=
    ^ <variable>

als einer weiteren syntaktischen Alternative für linksseitige Variablennotationen.

Beispiel: **task** s:= ^y;
task ^y := s;

Da durch diese Syntaxtransformation der Ruf der entfernten Prozedur verborgen wird, sollte bei allen Zugriffen die implizite Prozeßvariable **sender** wieder auf dem Wert vor dem Prozedurruf zurückgesetzt werden.

Vererbung

Das vorhandene Vererbungskonzept für Sorten wird wie bisher genutzt. Demzufolge werden alle in Abschnitt 2.2.2 „*Vererbung*” (S.29) aufgeführten Mängel übernommen. Strukturelle und algorithmische Beschreibungsmittel für Sorten müßten über diese konkrete Variante hinaus überarbeitet werden, um die Mängel zu beheben.

Zuweisungen

Da die Referenzen als Modell eingeführt wurden, ist die Zuweisungssemantik von Referenzen durch die aktuelle Semantik von SDL wohldefiniert: die Referenzen einer Vererbungshierarchie sind als *syntype*-Konstruktionen zuweisungskompatibel. Der Versuch, einen Wert aus einer Referenz zu extrahieren, der nicht mit der statischen Typbeschreibung korrespondiert, impliziert eine SDL-Ausnahme, d.h. das Modell ist dynamisch typsicher.

Methoden

Ein Beispiel für eine Methode *print* mit später Bindung eines Arguments kann wie folgt aufgeschrieben werden:

```
newtype Base
    literals foo;
    methods print : virtual ^Base -> Charstring;
    method print; fpar ref ^Base; returns Charstring;
        start; return '1';
    endmethod;
endnewtype;
```

```
newtype Sort inherits Base
    operators all;
    adding literals bar;
    methods print : redefined ^Sort -> Charstring;
    method print; fpar ref ^Sort; returns Charstring;
        start;
            decision ^ref;
                (foo): return 'ONE';
                (bar): return 'TWO';
            enddecision;
    endmethod;
endnewtype;
```

Innerhalb der Methodendefinition wird dereferenziert, d.h. vom Standpunkt des Modells wird eine entfernte Prozedur aufgerufen. Daraus folgt, daß das SDL-Operatorkonzept als Modell für Methoden nicht nutzbar ist. Es gilt mit der gleichen Begründung sogar allgemein, daß Referenzen in SDL-Operatordefinitionen nicht nutzbar sind.

Eine Alternative zu Operatoren sind Prozeduren. Die Prozedur *print* in Bild 21 realisiert ein

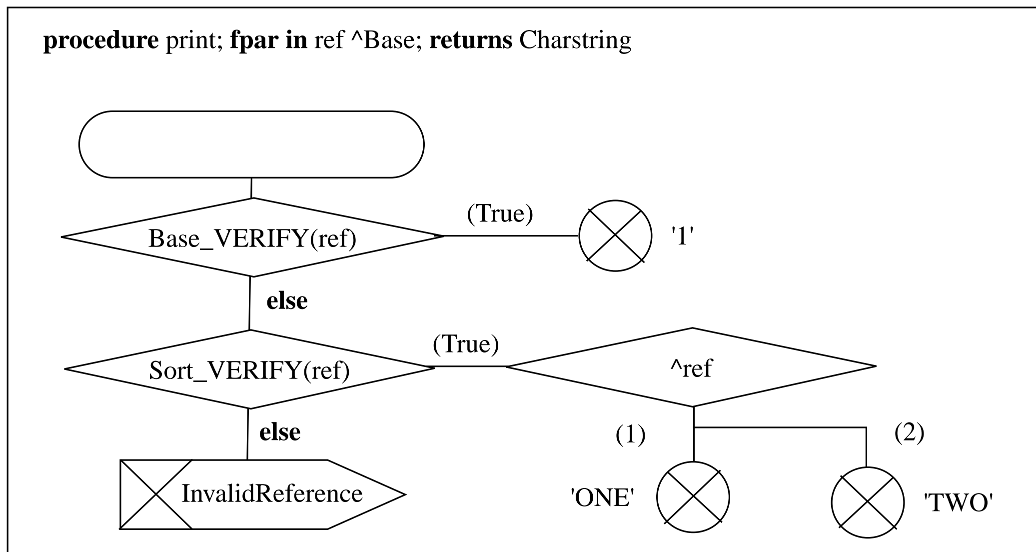

Bild 21: Prozedur print

mögliches Verhalten einschließlich der späten Bindung. Man transformiert also jede Methodenredefinition in eine *decision*-Konstruktion mit entsprechender Laufzeittypabfrage. Methoden ohne virtuelle Elemente werden durch einfache Prozeduren realisiert. Dieses Transformationsmodell hat folgende Konsequenzen:

- Es ist nicht möglich, Methoden zu überladen, da SDL-Prozeduren nicht überladen werden dürfen.
- Die Änderung von Methodensignaturen ist nur für den virtuellen Referenzparameter zulässig. Man kann auch eine Syntax mit implizitem virtuellen Parameter, analog zu C++ vorgeben.
- Die Bindung ist exakt. Gibt es keine Redefinition der Methode für eine Vererbungsstufe, wird die Ausnahme *InvalidReference* beim Aufruf der Methode an einem Objekt dieser Stufe geworfen.

Speicherverwaltung

Wenn eine Referenz erzeugt wurde, so ist der Zugriff auf den an diese Referenz gebundenen Term so lange möglich, wie diese Referenz oder eine beliebige Kopie der Referenz existiert oder die *delete*-Methode explizit gerufen wurde. Das entspricht einer expliziten Speicherverwaltung.

6.2.3 Varianten der Sammelobjektverwaltung

Das angegebene konkrete Transformationsmodell für die Sammelobjektverwaltung ist eine Grundvariante. Jetzt werden einige voneinander unabhängige Aspekte der Modelldefinition noch einmal genauer diskutiert, mit dem Ziel das Modell in verschiedene Varianten aufzufächern und partielle Verbesserungsmöglichkeiten aufzuzeigen.

Verteilung der Speicherprozesse

Man ist keinesfalls gezwungen, alle Referenzen einer Systemspezifikation in einem Speicherprozeß zu verwalten. Die Verlagerung der Vererbungshierarchien auf einzelne Prozesse ist strukturell möglich aber nicht spannend. Betrachtet man jedoch eine Aufspaltung der Speicherprozesse für statische SDL-Strukturen (z.B. Blöcke), dann kann man durchaus von einer verteilten Speicherverwaltung auf SDL-Niveau sprechen. Entsprechende Erweiterungen für Referenzmigration über Blockgrenzen sind denkbar. Gibt es mehrere Speicherprozesse, müssen sie auch in der Spezifikation durch geeignete Sprachkonstrukte unterscheidbar sein.

Beispiel: Die Anforderung des Speichers aus der Speicherverwaltung eines vorhandenen Blocks b könnte durch Qualifizierung explizit angegeben werden:

```
task y:= <block b>new Sort(1);
```

Konstruktoren

Durch entsprechende Syntaxtransformation könnte der SDL-Nutzer bei der Definition von Datentypen auch eine Konstruktornotation erhalten. Diese kann geeignet in den Verhaltensgraphen der *MAKE*-Prozedur übernommen werden. Es ist eine Frage der Transformation der konkreten Syntax auf das Modell, ob und wann im Vererbungsfall der Konstruktor der Basisklasse gerufen wird und wie Überladung von Konstruktoren funktioniert.

Beispiel:

```
newtype Base...
    literals 1, ...;
    methods
        new : ^Base; /* alternativer Konstruktor ohne Argument */
    method new; fpar _this ^Base;
        start; task ^_this := 1; return;
    endmethod;
endnewtype;
```

Veränderung der Zuweisungsregeln

Prinzipiell sind die Referenzen einer Vererbungshierarchie uneingeschränkt zuweisungskompatibel. Das ist für eine Spezifikationstechnik mit strengem Datentypkonzept unbefriedigend, es sollten zusätzliche Einschränkungen definiert werden. Deshalb wurde im Grundmodell bereits eine Laufzeittypinformation eingebaut (VERIFY), die bei Bedarf auch auf die Erkennung der Vererbungshierarchie erweitert werden kann. Die Auswertung der Laufzeittypinformation erfordert in jedem Fall einen Zugriff auf einen Prozeß, d.h. formal findet ein Signalaustausch statt. Jetzt gibt es zwei Varianten, diese Informationen zu nutzen:

1. Man definiert ein Transformationsmodell, daß jede Zuweisung in einen Test entsprechend eines Regelsatzes für Referenzzuweisungen und die Zuweisung selbst aufspaltet.
2. Man erweitert die Semantik der impliziten *syntype*-Tests um die Möglichkeit, nicht konstante Terme für notwendige Referenztests zuzulassen.

Beide Varianten haben das Problem, daß das SDL-Modell für Signale eine implizite Synchronisation mit anderen Zugriffen auf den Referenzprozeß definiert. Arbeitet der Referenzprozeß beim Eintreffen des Signals Transitionen ab, blockiert der Test. Dieses Problem wird im Abschnitt „*Zugriffssynchronisation*" (S.118) noch einmal aufgegriffen. Außerdem kann sich der Wert der Referenz zwischen dem Test und dem Zugriff ändern (...MODIFY kann in dieser Modellvariante den Typ des Werts einer Referenz ändern), wenn andere Prozesse auf diese Referenz zugreifen können.

Folgende, für andere objektorientierte Sprachen geläufige Einschränkungen für die Zuweisung von Referenzen könnten definiert werden:

1. Referenzen einer Basissorte oder Basisklasse können nur an Referenzen der Ableitung zugewiesen werden, wenn das Objekt der Referenz mindestens ein Objekt der Ableitung ist.
2. Referenzen verschiedener Vererbungszweige können nicht gegeneinander zugewiesen werden. Diese Einschränkung kann sogar statisch getestet werden.

Falls ein dynamischer Test eine fehlerhafte Referenzzuweisung entdeckt, könnte eine vordefinierte Ausnahme geworfen werden oder eine leere Referenz (Nil) zugewiesen werden.

Die Zuweisung zwischen Referenzen und Werten ist formal nicht möglich, da diese unterschiedliche Datentypen repräsentieren. Man muß stets die geeigneten Transformationsprozeduren (das Objekt dereferenzieren oder aus einem Wert konstruieren) rufen. Das ist durchaus im Sinne des statischen Datentypkonzepts von SDL. Transformationsbasierte Erweiterungen für derartige Zuweisungen sind denkbar.

Globale Variablen

Da es einen zentralen Speicherprozeß gibt, kann dieser Prozeß auch zur Definition statischer Sortenvariablen analog den statischen Klassenvariablen in C++ genutzt werden. Beispielsweise kann für die Notation

```
newtype Sort
    static out name Charstring := 'Sort';
endnewtype;
```

im Speicherprozeß implizit eine Variable *name* der Sorte *Charstring* angelegt werden. Der Zugriff erfolgt über exportierte Prozeduren, z.B.

```
exported procedure nameExtract; fpar ^Sort; returns Charstring;
    start; return name;
endprocedure;
```

Alle notwendigen Definitionen werden durch Transformation bereitgestellt. Wird das Schlüsselwort **out** weggelassen oder durch **in/out** ersetzt, so wird auch implizit der schreibende Zugriff mit *nameModify* definiert. Der eigentlich notwendige Prozedurruf in Aktionen kann durch eine beliebige Syntax ersetzt werden, z.B.

```
task string_variable := Sort!name;
```

Referenzen auf Referenzen

Es ist denkbar, daß das Schema zur Definition der Referenzsorten auch rekursiv auf die transformierte Spezifikation angewendet wird. Das gestattet beliebige Indirektheit auf Werte, wie sie beispielsweise von Zeigern aus C++ bekannt sind.

Speicherverwaltung

Die Grundvariante gibt dem SDL-Nutzer die Möglichkeit, Objekte zu löschen, d.h. es gibt eine explizite Speicherverwaltung. In einer Spezifikationstechnik ist jedoch eine implizite Speicherverwaltung anstrebenswert. Mit Blick auf die SDL-Semantik muß man nur die *delete*-Prozedur des Speicherprozesses streichen und einen unendlich großen Speicher voraussetzen. Dann wird die Speicherverwaltung ein Implementierungsdetail des genutzten SDL-Werkzeugs.

Man kann durch die Nutzung der SDL-Virtualität sogar eine Schnittstelle für die formale Redefinition dieser Methode vorsehen. Angenommen die Modellvariante gibt die folgende Signatur vor:

```
virtual exported procedure DELETE;
    fpar in ref REFERENCE;
    start virtual;
        task storage(ref)!empty := NULL; return;
endprocedure;
```

Dann wäre eine Redefinition im Modul *Predefined* durchaus eine adäquate Schnittstelle für werkzeugspezifische Einschränkungen, z.B. bei der Klärung, welche Reaktion die Nutzung der *delete*-Konstruktion nach sich zieht:

```
redefined exported procedure DELETE;
    fpar in ref REFERENCE; raise OperationNotImplemented;
    start redefined;
        task 'using JAVA memory management';
        raise OperationNotImplemented;
endprocedure;
```

Nun wird aber SDL auch genutzt, um möglichst effiziente Programme basierend auf C oder C++ automatisch zu generieren. Diese Sprachen besitzen eine explizite Speicherverwaltung. Wenn Spezifikationen mit dem Hintergrund einer derartigen Zielsprachenabbildung entwickelt werden, so wird der Entwickler die Speicherproblematik kennen und beachten. Es ist auch möglich, verschiedene Freispeicherverwaltungen, abhängig von Entwicklungsstand der Spezifikation, zu verwenden, ähnlich wie es bei der Programmverifikation in realen Sprachen praktiziert wird. Allerdings sind ungültige Referenzen Fehlerquellen, die man in einer Spezifikationstechnik eigentlich nicht erwarten würde. Neben der expliziten Variante, Objekte zu löschen, gibt es eine Reihe denkbarer Varianten, von denen einige vorgestellt werden.

Lokalitätsprinzip

Möglicherweise soll der Zugriff auf den Term einer Referenz auf bestimmte Struktureinheiten eingeschränkt werden. Das kann einfach durch das Abspeichern einer Kontextinformation (z.B. den Hinweis wer die Referenz erzeugt hat) und einer entsprechenden Abfrage beim Zugriff erreicht werden. Die Datendefinition aus „*Spezifikation der Transformationsziele für Base und Sort*” (S.96) könnte für prozeßlokale Referenzen wie folgt geändert werden:

```
ELEM ::= sequence {
    owner PId,
    elem  choice {
        empty       NULL,
        elemBase    Base,
        elemSort    Sort
    }
}
```

Die Implementierung der Zugriffsprozeduren ist entsprechend anzupassen. Die Zugehörigkeit einer Referenz kann bei entsprechenden Schnittstellen sogar zu einem anderen Prozeß transferiert werden. Das entspricht einem ITU-T-intern diskutierten Vorschlag, ein *owned pointer*-Konzept für SDL einzuführen.

Es wäre neben der lokalen Sichtbarkeit auch denkbar, daß Referenzen zerstört werden, wenn der erzeugende Prozeß gestoppt wird. Formal müßte dazu jeder Prozeß eine Liste seiner Referenzen verwalten, die vor Ausführung der *stop*-Aktion geeignet abgearbeitet wird. Alternativ kann man auch den Speicher- und Nutzerprozeß zusammenlegen. Das wäre eine sehr einfache implizite Speicherverwaltung, die für lange laufende Prozesse problematisch ist.

Prozeßlokale[1] Referenzen sind der Standardfall für SDL-2000, da viele Synchronisationsprobleme beim Zugriff mehrerer Prozesse auf Referenzen in dieser Variante wegfallen.

Maximalprinzip

Analog der Einschränkung von Prozeßinstanzmengen könnte die Anzahl der möglichen Referenzen pro Sorte eingeschränkt werden. Das kann sogar durch Redefinition der *new/delete*-Prozeduren des Speicherprozesses formal spezifiziert werden. Damit können Werkzeuge den benötigten Speicherplatz ermitteln und eine besonders effiziente Speicherzuweisung für bestimmte Sorten realisieren. Allerdings ist keine objektorientierte Sprache bekannt, die eine derartige Einschränkung zuläßt. Diese Variante könnte bei der Codegenerierung für Plattformen mit sehr begrenzten Ressourcen (*embedded systems*) zum Einsatz kommen.

Referenzzählung

Wenn jede Referenz bei einer Kopie oder beim Überschreiben einen internen Zähler modifiziert, spricht man von Referenzzählung. Ist der Zählerstand 0, kann der Speicher freigegeben werden. Auf der Grundlage der *syntype*-Tests bei allen expliziten oder impliziten Zuweisungen könnte die Referenzzählung in das semantische Modell von SDL eingebaut werden. Es gibt aber zwei wesentliche Probleme mit diesem Modell:

- Die Freigabe des Zählers von Referenzen in verschachtelten Strukturen, die auf normalen Sortendefinitionen beruhen, erfordert einen erheblichen Aufwand bei der Definition der entsprechenden Transformationsregeln.
- Es lassen sich zyklische Strukturen definieren, bei denen der Algorithmus für die verschachtelte Freigabe der Strukturen versagt. Das ist ein bekanntes Problem der Referenzzählung. Ist das Laufzeitsystem nicht in der Lage, mit weiterem Verwaltungsaufwand derartige Zyklen zu durchbrechen, muß der Nutzer die Zyklen durch Zuweisung der *null*-Referenz explizit beseitigen.

Alternative Datentypkonstruktionen

Der Nutzer von *„SDL in Combination with ASN.1“* kennt neben den gewöhnlichen Strukturdefinitionen auch Aufzählungen und Auswahlkonstruktionen sowie andere nützliche Zusatzkonstruktionen für Daten, z.B. optionale Felder. Da es bereits eine Semantikdefinition für die ASN.1-Kombination gibt, könnte man ohne größeren Aufwand gute Sprachkonzepte mit zu SDL passender Syntax in die Sprache importieren.

6.2.4 Einzelobjektverwaltung

Eine Prozeßinstanz verwaltet genau einen Datenbereich, der aus mehreren Werten bestehen kann. Dieser Datenbereich, gebildet durch lokale Prozeßvariablen, wird ausschließlich über den *PId*-Wert des Prozesses mit einer prozeduralen Schnittstelle referenziert, d.h. der Datentyp für Referenzen ist die Sorte *PId*. Jede Prozeßinstanz repräsentiert demzufolge ein eigenständiges Objekt mit entsprechenden Attributen. Für strukturell verschiedene Datenbereiche gibt es auch unterschiedliche Prozeßdefinitionen, die als Objektbeschreibungen zu verstehen sind. Deshalb ist der Begriff des Referenzobjekts für eine Verwaltungsinstanz durchaus passend.

---

1. Genauer: Referenzen sind lokal zu Agenten.

Im Bild 22 erhalten Nutzerprozesse den Zugriff auf die Attribute der Referenzobjekte nur über die *PId*-Werte der Referenzobjekte. Zugriffe ohne *PId*-Werte würden entsprechend der

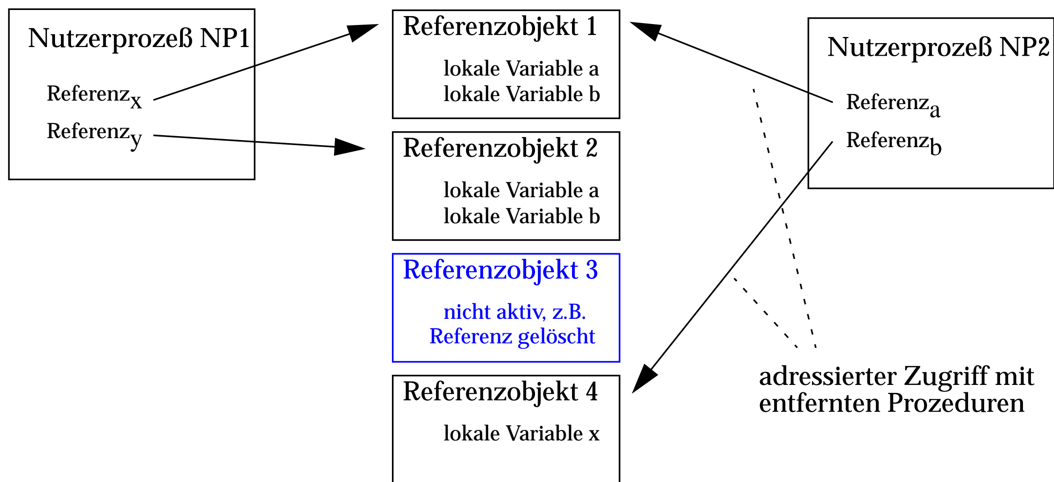

Bild 22: Referenzen mit Referenzobjekten

SDL-Semantik zufällig von einem für den Zugriff geeigneten Referenzobjekt beantwortet werden und sollten dem SDL-Nutzer durch entsprechende Syntax nicht ermöglicht werden. Aus der Sicht des Nutzers muß man Objektbeschreibungen von bisherigen Datendefinitionen unterscheiden und im Gegensatz zur Sammelobjektverwaltung auch explizit angeben. Typischerweise denkt man bei Vererbung an Strukturen, für Einzelobjektverwaltung kein Problem, da Vererbung als Bestandteil der Transformationsregeln neu zu klären ist:

Beispiel:

```
object type Base struct                object type Sort inherits Base; adding
    i Integer.;                            b Boolean;
endobject type;                        endobject type;

process NP1;
    dcl x Base, y Sort; /* Objektvariablen */
    dcl b Boolean, i Integer; /* übliche Sortenvariablen */
    ...
    task x := create Base(1) , y := create Sort( 2, True );
    task 'Übergabe von x an NP2';
    ...
    task i := y!i;
endprocess;

process NP2;
    dcl a Base, b Sort;
    ...
    task 'Empfang von x aus NP1 mit a';
    task b := create Sort( a!i, False );
    ...
endprocess;
```

Spezifikation der Transformationsziele für *Base* und *Sort*

Es folgen schrittweise der Grundaufbau eines Referenzobjekts, die Betrachtung von Vererbung und eine Strategie zum Löschen für eine konkrete Basisvariante.

Schritt 1: Da auch Vererbung von Objektbeschreibungen unterstützt werden soll, erfolgt die semantische Transformation erst auf einen Prozeßtyp, von dem anschließend eine Prozeßinstanzmenge gebildet wird. Dieser Prozeßtyp wird Prozeduren für die Zugriffe auf die Attribute bereitstellen. Mit dem obigen Beispiel wäre die Grundstruktur des Prozeßtyps wie folgt:

```
process type Base; fpar i Integer;
    exported procedure i_EXTRACT; returns Integer;
        start;
            return i;
    endprocedure;

    exported procedure i_MODIFY; fpar in e Integer;
        start; task i:= e; return;
    endprocedure;

    /* Weitere Prozeduren zur Verwaltung des Objekts und der
       Verhaltensgraph fehlen noch.
       ...
    */
endprocess type;

process Base(0) : Base;
```

Bei Zuweisungen des *PId*-Werts einer Objektreferenz wird nur der Verweis auf den zugehörigen Prozeß kopiert. In diesem Sinne besitzt die Definition von *Base* eine Referenzsemantik.

Schritt 2: Die Vererbung von Referenzobjekten entspricht der Vererbung von Prozeßtypen. Es wird eine Laufzeittypinformation in die Klasse aufgenommen:

```
process type Base; fpar i Integer;
    dcl is_type Charstring := '';     /* Laufzeittypinformation */

    /* Zugriffsprozeduren für lokale Variablen aus Schritt 1 */
    exported procedure i_EXTRACT referenced;
    exported procedure i_MODIFY referenced;

    /* Abfrage der Laufzeittypinformation des Referenzobjekts */
    exported procedure Base_VERIFY; returns Boolean;
        start;
            return if is_type='Base' then True else False fi;
    endprocedure;

    /* Starttransition, setzt Laufzeittypinformation */
    start virtual; task is_type := 'Base'; nextstate available;

    /* In diesem Zustand werden eingehende Prozedurrufe
       bearbeitet. Ausnahmen, geworfen bei Referenzfehlern,
       werden implizit an den Rufer übertragen, müssen aber
       im Speicherprozeß selbst ignoriert werden.
    */
    state available; onexception AccessErrorHandler;
        input procedure i_EXTRACT; nextstate -;
        input procedure i_MODIFY; nextstate -;
        input procedure i_VERIFY; nextstate -;
        /* Andere eingehende Rufe werden mit der
           Ausnahme InvalidReference abgewiesen.
        */
        input procedure * raise InvalidReference; nextstate -;
```

```
    exceptionhandler AccessErrorHandler;
        handle *; nextstate -;
endprocess type;
```

Die Angabe zusätzlicher Prozeßparameter (Attribute) bei Vererbung erzeugt einen neuen Satz von Zugriffsprozeduren für diese Attribute.

```
process type Sort inherits Base; fpar b Boolean;

    /* Zugriffsprozeduren für b analog zu i von Base */
    exported procedure b_EXTRACT referenced;
    exported procedure b_MODIFY referenced;

    /* Abfrage der Laufzeittypinformation des Referenzobjekts */
    exported procedure Sort_VERIFY; returns Boolean;
        start;
            return if is_type='Sort' then True else False fi;
    endprocedure;

    /* Starttransition, wird mit jeder Vererbung redefiniert */
     start redefined; task is_type := 'Sort'; nextstate available;

    /* Zugriffe auf das neue Attribut b im Prozeßgraph */
    state available; onexception AccessErrorHandler;
        input procedure b_EXTRACT; nextstate -;
        input procedure b_MODIFY; nextstate -;
        input procedure b_VERIFY; nextstate -;

endprocess type;
```

Schritt 3: Auch ein einfaches Löschen von Referenzen wird unterstützt. Hier gibt es ein Problem mit der SDL-Semantik für den Ruf entfernter Prozeduren: der Ruf blockiert wenn der gerufene Prozeß nicht mehr existiert. Das ist für ungültige Referenzzugriffe kein adäquates Verhalten. Deshalb werden die Referenzobjekte unter keinen Umständen gestoppt und die Prozeßtypdefinition erhält eine Ergänzung:

```
process type Base; fpar i Integer;

    /* Inhalte aus Schritt 1 und 2
        ...
    */

    virtual exported procedure DELETE;
        start virtual;
            /* Hier kann eine Finalisierungsaktion eingetragen
               werden, weshalb die Definition der Prozedur
               virtual ist.
            */
            return;
    endprocedure;

    state available; input procedure DELETE;
            nextstate deleted;

    state deleted;
        /* In diesem Zustand werden alle eingehenden
           Prozedurrufe abgewiesen.
        */
        input procedure * raise InvalidReference; nextstate -;
endprocess type;
```

Durch den Aufruf von *DELETE* wird ein Zustandswechsel durchgeführt. In diesem werden alle weiteren Prozedurrufe (in diesem Fall auch ein wiederholter *DELETE*-Ruf) mit einer Ausnahme abgelehnt.

Die Erklärung verschiedener objektorientierter Konstruktionen auf der Grundlage dieses Modells erfolgt wie bei der Sammelobjektverwaltung in einzelnen Abschnitten.

Konstruktion

Für die Konstruktion eines Referenzobjekts sollte eine adäquate Syntax genutzt werden, die transformiert wird.

Beispiel: **task** y := **create** Sort(42,True);

Falls die Bezeichnerauflösung nicht eindeutig ist, kann der Objektbezeichner *Sort* qualifiziert werden. Die semantische Transformation erzeugt die Aktionen:

```
create Sort(42,True);
task   y := offspring;
```

Die implizite Prozeßvariable **offspring** (für den letzten erzeugten Prozeß) sollte wieder den ursprünglichen Wert nach der Ausführung der Objekterzeugung enthalten. Auf eine Konstruktornotation und Referenzen auf lokale Variablen wird in dieser Basisvariante verzichtet.

Zugriff auf Werte von Referenzen

Für die Funktionalität, die Attribute des Referenzobjekts zu extrahieren, wird man eine Reihe von vordefinierten Prozeduren benötigen, die Referenzobjekte mit verschiedenen Zielstellungen kopieren (z.B. Kopie der Attribute mit oder ohne Rekursion bei Referenzen, Konvertierung entlang der Vererbungskette). Das ist jedoch eine konzeptionelle Detailentscheidung, die hier nicht weiter betrachtet wird. Der Zugriff auf Elemente von Klassen kann exakt wie der Zugriff auf Strukturen erfolgen.

Beispiel:

```
task i := y!i;          /* Zugriff auf ein Element */
task y!i := 42;         /* Modifikation eines Elements */
task x := y.copy();     /* Kopie eines Objekts */
```

Die semantische Transformation ist

```
task i := call i_EXTRACT to y;
call i_MODIFY(42) to y;
task y := call copy to x;
```

In allen Varianten sollte die implizite Prozeßvariable **sender** wieder auf den Wert vor dem Prozedurruf zurückgesetzt werden.

Vererbung

Vererbung von Objektbeschreibungen wird als Bestandteil der Modelltransformation auf Prozeßvererbung abgebildet. Deshalb sind Objekt- (d.h. Prozeß-) und Sortenvererbung natürlich nicht kombinierbar. Es gibt demzufolge mit diesem Grundmodell zwei, nicht in der Vererbung kombinierbare Methoden, Daten zu beschreiben.

Zuweisungen

Alle Referenzen sind *PId*-Werte. Damit besteht uneingeschränkte Zuweisungskompatibilität zwischen allen Objektbeschreibungen und zusätzlich der Sorte *PId*. Durch syntaktische als auch semantische Einschränkungen kann verhindert werden, daß Werte von Objektbeschreibungen in einer Spezifikation wie *PId*-Werte nutzbar sind. Auf Details soll an dieser Stelle verzichtet werden.

Methoden

Die Realisierung von Methoden liegt für Referenzobjekte auf der Hand: Prozeduren mit allen bekannten Eigenschaften, wie Vererbung und Virtualität können direkt genutzt werden. Man benötigt nur eine passende Syntax. Es bietet sich die Nutzung algorithmischer Operatordefinitionen mit dem Schlüsselwort **method** anstelle von **operator** an. Innerhalb der algorithmischen Definition hat man Zugriff auf die Attribute, da formal eine Prozedur innerhalb eines Prozesses spezifiziert wird, die natürlich Zugriff auf die Prozeßvariablen hat.

```
object type Base; struct
    i Integer;
    virtual methods print : -> Charstring;
    method print; returns Charstring;
        start; return 'Base';
    endmethod;
endobject type;

object type Sort inherits Base adding
    b Boolean;
    redefined methods print : -> Charstring;
    method print; returns Charstring;
        start;
            /* Zugriff der Methode auf den Prozeßparameter */
            decision b;
                (True):     return 'b: TRUE';
                (False):    return 'b: False;
            enddecision;
    endmethod;
endobject type;
```

Der Ruf der Prozedur am Referenzobjekt muß mit der Referenz, d.h. dem *PId*-Wert des Prozesses, adressiert werden (späte Bindung per *PId*-Wert). Die Methode wird am Objekt gerufen. Transformationsregeln erlauben adäquate Methodenrufe:

```
string_variable := reference_variable!print();
```

die zu

```
string_variable := call print to reference_variable;
```

transformiert werden. Existiert die gerufene Prozedur nicht, wird entsprechend der Prozeßspezifikation des Referenzobjekts die Ausname *InvalidReference* geworfen.

Methoden unterliegen vorerst den Sichtbarkeitsregeln für Prozeduren, demzufolge gibt es wie für Prozeduren keine Überladung. Die Definition erfolgt syntaktisch innerhalb von Objektbeschreibungen. Vererbung und Nutzung von Virtualität folgen den vorhandenen SDL-Regeln für Vererbung und Virtualität von Prozeduren in Prozeßtypen. Eine implizite Variable **this** innerhalb einer Methode könnte das gerufene Objekt repräsentieren, um den *PId*-Ausdruck **self** in Methoden zu vermeiden.

Wegen des zugrundeliegenden Prozedurmodells ist die algorithmische Methodendefinition vorgegeben. Die Verwendung von expliziten oder impliziten Prozeßvariablen (**self**, **sender**, ...) muß genauso wie in Operatoren untersagt werden, da diese unerwartete Werte des Referenzprozesses liefern würden.

Speicherverwaltung

Wenn ein Objekt erzeugt wurde, so ist der Zugriff auf Attribute und Methoden so lange möglich bis die *DELETE*-Methode explizit gerufen wurde. Danach werden Zugriffe mit der Ausnahme *InvalidReference* abgewiesen. Das entspricht einer expliziten Speicherverwaltung.

6.2.5 Varianten der Einzelobjektverwaltung

Die Einzelobjektverwaltung ist ebenso flexibel wie die Sammelobjektverwaltung. Dieser Unterabschnitt wird einige nur auf die Einzelobjektverwaltung anwendbare Veränderungen vorstellen.

Konstruktoren

Durch entsprechende Syntaxtransformation könnte der SDL-Nutzer bei der Definition von Datentypen auch eine Konstruktornotation erhalten. Die Konstruktoraktionen können geeignet in virtuelle *start*-Transitionen der Prozeßtypdefinitionen übernommen werden. Es ist eine Frage der Transformation der konkreten Syntax auf das Modell, ob und wann im Vererbungsfall der Konstruktor der Basisklasse gerufen wird und wie Überladung von Konstruktoren funktioniert. Im einfachsten Fall wird die SDL-Semantik für die Redefinition der *start*-Transitionen übernommen.

Beispiel:

```
object type Base struct
  i Integer;
  /* nutzerdefinierter Konstruktor mit fester Signatur */
  method create; fpar my_i Integer;
    start;
      /* alle Aktionen für start-Transition von Base */
      task i := my_i;
      return;
  endmethod;
endobject type;
```

Wurzel aller Objekttypen

Viele Programmiersprachen stellen einen universellen Basistyp für alle Objektbeschreibungen bereit, der allgemeingültige Signaturelemente enthält. Natürlich ist auch ein Wurzelprozeßtyp für Referenzobjekte definierbar, der insbesondere als Objekttypdefinition im Modul *Predefined* spezifiziert ist. Damit werden universell einsetzbare Methoden sichtbar bzw. für SDL-Werkzeuge sogar konfigurierbar. Die Regeln für Zuweisungskompatibilität müssen auf die Existenz einer sichtbaren Wurzelsorte angepaßt werden, da es dann genau eine Vererbungshierarchie gibt.

Die SITE-Werkzeuge nutzen das Konzept der abstrakten, aber nicht sichtbaren Wurzelsorte, um beispielsweise die Kodierungsoperatoren für alle Datendefinitionen bereitzustellen.

Speicherverwaltung

Die Grundvariante gibt dem SDL-Nutzer die Möglichkeit Objekte zu löschen. Prinzipiell kann man alle Erweiterungen der Sammelobjektverwaltung (s. Seite 105) auch auf das Modell der Einzelobjektverwaltung anwenden. Das Maximalprinzip kann sehr natürlich mit der maximal möglichen Anzahl von Prozeßinstanzen umgesetzt werden. Allerdings fehlt das Recycling gelöschter Objekte, man muß das Löschen und Erzeugen von Objekten präzisieren.

Veränderte Zuweisungen

Das Basismodell für Referenzobjekte hat den Nachteil, daß es zwei unterschiedliche Methoden gibt, Daten zu spezifizieren. Beispielsweise kann eine Struktur als Sorte oder als Objekt definiert werden. Einziger Unterschied ist die Zuweisungssemantik.

Beispiel: Die Datentypen *ValueStruct* und *ReferenceStruct* sind sich zwar sehr ähnlich, es besteht jedoch keine Zuweisungskompatibilität.

```
newtype ValueStruct struct
    i   Integer;
    b   Boolean;
endnewtype;
```

```
object type ReferenceStruct struct
    i   Integer;
    b   Boolean;
endobject type;
```

Eigentlich möchte man für genau eine Datendefinition sowohl Referenzen als auch Werte nutzen. Die folgende Idee bietet eine Lösung für dieses Problem.

Man beschränkt den SDL-Anwender auf die Nutzung vordefinierter Sorten. Es sind also nur Spezifikationen von Objektbeschreibungen zulässig. Damit hätte SDL bei Zuweisungen eine Referenzsemantik, wie sie aus Programmiersprachen wie *Python* oder *Java* bekannt ist. Eine Wertzuweisung erreicht man normalerweise, indem eine Referenz explizit kopiert wird. Man kann aber auch bei der Definition des Datentyps oder auch den Bezeichnern eine Kennzeichnung anbringen, daß bei Zuweisungen die Referenz implizit kopiert wird. In Analogie zu SDL-2000 sei ein mit **value** anstelle von **object** markierter Datentyp als Wertetyp (Zuweisungen mit impliziter Kopie der Referenz) und eine unmarkierte Datenbeschreibung als Objekttyp (Zuweisungen als Referenz) bezeichnet.

Referenziert man einen Datentyp mit einer neuen Markierung, ändert sich nicht die Objektbeschreibung sondern nur die Zuweisungssemantik, d.h. die Zuweisungskompatibilität geht nicht verloren. Die auftretenden Fälle für die Markierungen werden an einer exemplarischen Strukturdefinition erläutert.

links := rechts	Wertetyp A	Objekttyp A
Wertetyp A	Kopie des Werts	Kopie des Werts
Objekttyp A	Kopie des Werts	Referenzzuweisung
Wertetyp B	nur möglich, wenn der dynamische Typ von A mindestens ein B-Objekt ist; Kopie Werts	nur möglich, wenn der dynamische Typ von A mindestens ein B-Objekt ist; Kopie Werts
Objekttyp B	Kopie, wenn das kopierte A-Objekt mindestens ein B-Objekt ist; sonst Zuweisung von Nil	Referenzzuweisung, wenn der dynamische Typ von A mindestens ein B-Objekt ist; sonst Zuweisung von Nil

Die Datentypen A und B stehen in einer Vererbungsrelation unabhängig von der Richtung. Sind die Datentypen links und rechts gleich, so gibt es unterschiedliche Markierungen aber keine Vererbung.

Bild 23: Polymorphe Zuweisungen für Objekt- und Wertetypen

Verschiedene Strukturdefinitionen: Es gibt zwei Möglichkeiten, eine Struktur zu definieren.

```
value type ValueStruct struct
   i  Integer;
   b  Boolean;
endvalue type;
```

```
object type ReferenceStruct struct
   i  Integer;
   b  Boolean;
endobject type;
```

Bei Zuweisungen von *ValueStruct*-Werten wird implizit ein Kopiervorgang ausgelöst, während *ReferenceStruct* eine Referenzsemantik besitzt. Die beiden Datenbeschreibungen sind natürlich nicht untereinander zuweisungskompatibel, da es unterschiedliche Objekttypen sind.

syntype-**Definition eines Objekttyps:** Eine Datenbeschreibung sei als Objekttyp definiert, die andere Datenbeschreibung wird mit einer *syntype*-Konstruktion als Wertetyp von diesem Objekttyp abgeleitet.

```
syntype ValueStruct =
      value ReferenceStruct
endsyntype;
```

```
object type ReferenceStruct struct
   i  Integer;
   b  Boolean;
endobject type;
```

Wegen der *syntype*-Konstruktion sind beide Datenbeschreibungen zuweisungskompatibel. Allerdings impliziert eine Zuweisung, an der ein *ValueStruct*-Wert beteiligt ist, einen Kopiervorgang. Man kann natürlich auch eine Variable direkt als Wertevariable definieren:

```
dcl var value ReferenceStruct;
```

syntype-**Definition eines Wertetyps:** Eine Datenbeschreibung ist als Wertetyp definiert, die andere Datenbeschreibung wird mit einer *syntype*-Konstruktion als Objekttyp von diesem Wertetyp abgeleitet. Wegen der *syntype*-Konstruktion sind beide Datenbeschreibungen zuweisungskompatibel.

```
value type ValueStruct struct
   i  Integer;
   b  Boolean;
endvalue type;
```

```
syntype ReferenceStruct =
      object ValueStruct
endsyntype;
```

Eine Zuweisung von oder an den Datentype *ValueStruct* impliziert wieder einen Kopiervorgang.

Vererbung: Die **object/value**-Markierung des zu definierenden Datentyps ist bei Vererbung unabhängig vom Basistyp.

Die Wertetypen sind bezüglich der Zuweisungskompatibilität wesentlich flexibler als Sorten. Es bedarf allerdings einer genauen Festlegung, welche Kombination aus Wert bzw. Referenz exakt welche Semantik bei der Kopie auslöst. An dieser Stelle werden zwei mögliche Varianten vorgestellt, um die Flexibilität zu demonstrieren:

1. vollständig polymorphe Zuweisungen und
2. eine mit SDL-2000 vergleichbare Zuweisungssemantik.

Vollständig Polymorph

links := rechts	Wertetyp A	Objekttyp A
Wertetyp A	Kopie des Werts	Kopie des Werts
Objekttyp A	Kopie des Werts	Referenzzuweisung
Wertetyp B	nur möglich, wenn der statische Typ von A eine Spezialisierung von B ist; Kopie Werts als B	Kopie des Werts als B, wenn der dynamische Typ von A mindestens ein B Objekt ist; dynamischer Fehler sonst
Objekttyp B	Kopie als A, wenn das kopierte A-Objekt mindestens ein B-Objekt ist; sonst Zuweisung von Nil	Referenzzuweisung, wenn der dynamische Typ von A mindestens ein B-Objekt ist; sonst Zuweisung von Nil

Die Datentypen A und B stehen in einer Vererbungsrelation. Sind die Datentypen links und rechts gleich, so gibt es unterschiedliche Markierungen aber keine Vererbung.

Bild 24: Zuweisungen für Objekt- und Wertetypen mit Projektion

Die implizite Kopie für Referenzen wird als Methode an den zuzuweisenden Wert gebunden, d.h. unabhängig vom statischen Datentyp wird das vollständige Objekt kopiert. Bezogen auf das Modell sind also die Prozeßinstanzmengen der Kopie und des Originals gleich, unabhängig von der Art der Zuweisung. Damit sind sowohl Werte- als auch Referenzzuweisungen polymorph.

Bild 23 erklärt die auftretenden Zuweisungsvarianten. Die Kopieroperation ist im Modell eine virtuelle Prozedur, die man in die Prozeßtypdefinitionen aus Unterabschnitt *„Spezifikation der Transformationsziele für Base und Sort"* (S.107) wie folgt einfügt:

```
/* virtuelle Kopie für Base */
virtual exported procedure COPY; returns PId;
    start virtual;
        create Base(i);
        return offspring;
endprocedure;

/* redefinierte Kopie für Sort */
redefined exported procedure COPY; returns PId;
    start redefined;
        create Sort(i,b);
        return offspring;
endprocedure;
```

Der Ruf der Kopiermethode kann entgegen der Übersicht aus Bild 23 auch entfallen, wenn ein Wertetyp an einen Objekttyp zugewiesen wird. Dann erhält man Referenzen auf lokale Variablen, die bei zu lösender Speicherverwaltung zum Problem der Lebensdauer von Referenzen führen.

Werteprojektion

Wenn die Datentypbeschreibung der zuzuweisenden Seite ein Wertetyp ist, so wird von diesem Wertetyp ein Objekt erzeugt und dieses nur mit den benötigen Attributen initialisiert. Die Zuweisung realisiert damit eine Projektion auf den Wertetyp der zuzuweisenden Seite. Die in Bild 24 getroffenen Festlegungen für die auftretenden Fälle sind so allgemein gehalten, daß die Projektion den zuzuweisenden Wert auch verändern kann. Für SDL-2000 reduziert sich die Projektion auf eine identische Kopie, da bei Wertezuweisungen die Typbeschreibungen wie in SDL übereinstimmen müssen. Die Tabelle in Bild 25 gibt die entsprechenden Einschränkungen an.

links := rechts	Wertetyp A	Objekttyp A
Wertetyp A	Kopie des Werts	Kopie des Werts
Objekttyp A	Kopie des Werts	Referenzzuweisung
Wertetyp B	nicht möglich	Kopie des Werts als B, wenn der dynamische Typ von A ein B-Objekt ist; dynamischer Fehler sonst
Objekttyp B	Kopie, wenn das kopierte A-Objekt mindestens ein B-Objekt ist; sonst Zuweisung von Nil	Referenzzuweisung, wenn der dynamische Typ von A mindestens ein B-Objekt ist; sonst Zuweisung von Nil

Die Datentypen A und B stehen in einer Vererbungsrelation. Sind die Datentypen links und rechts gleich, so gibt es unterschiedliche Markierungen aber keine Vererbung.
Alle Kopien sind vollständig.

Bild 25: SDL-2000-konforme Zuweisungen für Objekt- und Wertetypen

Eine Besonderheit ergibt sich bei virtuellen Methoden mit der Projektionsvariante. Diese können durchaus in Wertetypen definiert werden, wobei das virtuelle Argument natürlich kovariant ist. Das kann zu statischen Fehlern führen, wenn der definierende Datentyp abgeleitet wird.

```
value type Foo;
    virtual method my_copy : -> Foo;
    method my_copy; returns Foo;
        dcl f Foo := this; /* Zuweisung in Ableitungen möglicherweise statisch falsch */
        start;
            return f;
    endmethod;
endvalue type;
```

Hier gibt es zwei Lösungen:

a. Die Nutzung von **this** wird eingeschränkt oder untersagt.
b. Man führt kovariante Deklarationen ein so wie in SDL-2000 vorgeschlagen, z.B.

```
dcl f this Foo := this;
```

Damit wird die Variable *f* bei einer Ableitung des Datentyps *Foo* automatisch angepaßt.

Sichtbarkeitsregeln

Die bisherige Vererbung von Datenbeschreibungen bietet ein Regelwerk, Literale oder Operatoren von der Vererbung auszuschließen bzw. die Sichtbarkeit der Operatoren auf die Sorte zu beschränken. Eine feinere Abstufung der Sichtbarkeitskonzepte, insbesondere im Vererbungsfall für Objekttypen, ist sinnvoll. Entsprechend zu definierende Regeln betreffen nur die statische Semantik von SDL, d.h. letztendlich die Definition etwas komplexerer Transformationsregeln. Die Vorschläge aus SDL-2000 können direkt übernommen werden.

Alternative Datentypkonstruktionen

Der Nutzer von *„SDL in Combination with ASN.1“* kennt neben den gewöhnlichen Strukturdefinitionen auch Aufzählungen und Auswahlkonstruktionen. Es ist wünschenswert, daß derartige Datentypen bei Vererbung um weitere Aufzählungs- oder Auswahlkomponenten angereichert werden. Es sollte kein Problem sein, durch entsprechende Speicherdefinition der Referenzprozesse auch diese Konstruktionen zu ermöglichen,

Beispielswiese sind Aufzählungen mit einer *Integer*-Variablen und Prozeduren zur Realisierung der Literalnamen darstellbar. Es erhöht sich lediglich der Transformationsaufwand.

6.2.6 Variationen beider Grundmodelle

Einige interessante Modellvariationen sind sowohl auf die Sammel- als auch die Einzelobjektverwaltung anwendbar. Dieser Abschnitt diskutiert derartige Varianten.

Überladene Methoden

Durch die Abbildung von Methoden auf Prozeduren können Signaturen nicht zur Unterscheidung von Methodennamen mit herangezogen werden - es gibt keine Überladung. Das ist eine wesentliche Einschränkung gegenüber Operatoren.

Deshalb wäre es logisch, die Signatur in den Namen der Prozeduren mit aufzunehmen, und die Auflösungsregeln für Operatoren auch auf Prozeduren bzw. nach Anwendung der Transformation auf Methoden anzuwenden. Das ist formal möglich, man stößt aber in Spezifikationen auf unerwartete Nebeneffekte.

Bei Prozedurrufen ist es möglich Argumente wegzulassen, so daß die Anzahl der Argumente nicht mit den Vorgaben der statischen Signatur übereinstimmt.

Beispiel:

```
methods
    foo : Integer -> Boolean;                    /* 1 */
    foo : Integer, Boolean -> Boolean;           /* 2 */

... foo(42)    /* paßt zu 1, es kann auch 2 gemeint sein */
```

Man benötigt also eine Empfehlung, daß bei mehrdeutigen Aufrufen alle Argumente angegeben werden müssen. Bei Methodenrufen, dem eigentlichen Hintergrund dieser Änderung, wird das prinzipiell[1] erfüllt sein.

Auch versagt die Überladung von Prozeduren bzw. Methoden unerwartet in einigen Fällen. Betrachtet man zwei verschiedene Referenzdatentypen *A* und *B*, und die Methoden

```
func: A -> B;
func: B -> A;
```

so wird das Problem deutlich. Bei der Einzelobjektverwaltung sind beide Datentypen *syntype*-Konstruktionen von *PId*, d.h. beide Definitionen reduzieren sich auf

```
func : PId -> PId;
```

und erzeugen einen statischen Fehler. Bei der Sammelobjektverwaltung ist die Situation ähnlich, wenn die zugehörigen Sorten von *A* und *B* eine gemeinsame Basissorte besitzen. Verlangt man, daß Signaturen auch bezüglich der verwendeten Referenzdatentypen unterscheidbar sind, so muß ein Satz von Regeln definiert werden, wie die Prozedur- bzw. Methodenauflösung bei Referenzen funktioniert. Die bisher noch ausstehende formale Semantik von SDL-2000 sollte hier eine umfassende Antwort geben. Ein Vergleich mit Java und C++ demonstriert aber die zu erwartenden Schwierigkeiten bei SDL-2000 recht deutlich:

1. Bei dynamischer Bindung können in SDL-2000 die beteiligten Signaturen kovariant sein. Java kennt grundsätzlich keine Kovarianz, C++ unterstützt lediglich kovariante Rückgabewerte. Der Satz von Regeln für kovariante Rückgabewerte in C++ ist umfangreich, einige C++-Compiler haben damit Probleme. Demzufolge kann man davon ausgehen, daß der Regelsatz für kovariante Signaturen in SDL-2000 unvollständig oder die Unterstützung von SDL-Werkzeugen mangelhaft sein wird.

---

1. C++ kennt das Konzept von *default*-Werten bei Funktionen, das ist aber so in SDL nicht vorgesehen.

2. Java als auch C++ lösen die späte Bindung von Methoden anhand des ersten Arguments auf. SDL-2000 wird auch eine Auflösung mit mehreren Argumenten unterstützen. Das wird eine Reihe, bisher unbekannter Fehlerfälle in Werkzeugen als auch Spezifikationen verursachen.

Abgesehen von diesen zwei Punkten, gestattet SDL-2000 zusätzlich noch Zuweisungen zwischen Referenzen und Werten, d.h. das Komplexitätsniveau der Zuweisungsregeln erhöht sich um weitere Details. Es ist denkbar, daß die ITU-T-Datenkonzepte bezüglich der Überladungsproblematik zu komplex sind und so von SDL-Werkzeugen nicht unterstützt werden.

Bei der Einzelobjektverwaltung erfolgt die späte Bindung über die PId-Adresse des Referenzobjekts, d.h. es kann ohne zusätzlichen Aufwand nur ein Argument für späte Bindung von Methoden unterstützt werden. Für die Sammelobjektverwaltung ist die Bindung explizit mit einer *decision*-Konstruktion ausgeführt. Hier kann man durchaus komplizierte Algorithmen für mehrere Argumente entsprechend der Regeln des SDL-2000-Standards unterstützen.

Mehrfachvererbung

SDL bietet nicht das Konzept der Mehrfachvererbung. Wegen der direkten Modellierung des Datenkonzepts mit vorhandenen SDL-Strukturen, läßt sich damit auch die Mehrfachvererbung nicht ohne wesentliche Änderung der Transformationsregeln einführen. Außerdem ist das Konzept der Mehrfachvererbung nicht unumstritten, so daß hier darauf gänzlich verzichtet wird.

Mehrfachvererbung wird im SDL-2000-Modell nur für den Datentyp *PId* benötig. Es ist denkbar, für diesen speziellen Fall die ITU-T-Lösung zu übernehmen.

Zugriffssynchronisation

Eine Synchronisation des Zugriffs muß nur diskutiert werden, wenn mehrere SDL-Prozesse mit einer Referenz auf den gleichen Datenbereich zugreifen können. Es gibt zwei grundlegende Problemstellungen.

gleichzeitige Zugriffe: Sobald ein Prozeß schreibend auf den Datenbereich einer Referenz zugreift, muß eine geeignete Semantik für andere zugreifende Prozesse definiert werden.

sichere Bereiche: Es soll garantiert werden, daß mehrere Zugriffe eines Prozesses exklusiv ausgeführt werden.

Da in SDL-2000 erzeugte Referenzen von einem Prozeß, bzw. allgemeiner von Agenten, nicht direkt an andere Agenten versendet werden können, besteht das Problem des gleichzeitigen Zugriffs auf Referenzen nicht. Allerdings lassen sich Agenten verschachteln und auf diese Art Referenzvariablen in Blöcken definieren, auf die alle inneren Agenten zugreifen können. Diese Zugriffe werden jedoch auf ein Modell für exportierte Variablen abgebildet. Ob diese Hintertür für prozeßübergreifende Referenzen ausreicht und ob insbesondere die formale SDL-2000-Semantik alle möglichen Fälle ausreichend klärt, wird die Praxis zeigen.

Gleichzeitige Zugriffe auf Referenzen: Alle Zugriffe auf Referenzen werden als Ruf von entfernten Prozeduren auf Grundlage der formalen SDL-Semantik auf Signale abgebildet. Damit ist eine Synchronisation der Referenzzugriffe durch das SDL-Modell selbst gegeben:

- alle Zugriffsanforderungen (Signale) werden im *input*-Puffer des Referenzprozesses linearisiert[1] und
- jedes Signal wird exklusiv in den entsprechenden Transitionen abgearbeitet.

Diese Semantik hat einige Konsequenzen:

1. Der inkonsistente Zugriff mehrerer aktiver Prozesse auf Werte im gemeinsamen Speicher kann nicht modelliert werden. Das ist beispielsweise in C++ mit *thread*-Unterstützung möglich und bei der Abbildung von SDL auf Zielsprachen zu beachten.

a. Referenzzugriffe oder Methodenrufe können durch die Ausführung anderer Zugriffe verzögert werden,

b. durch Zugriffe auf Referenzen im Referenzprozeß selbst, z.B. innerhalb von Methoden, kann es zu Blockierungssituationen kommen.

Die Konsequenz a ist durchaus ein realistisches Verhalten bei verteilter Programmausführung, Punkt b stellt ein ernsthaftes Problem dar. Es bietet sich eine Lösung in zwei Schritten an:

- Das Transformationsmodell für den Ruf entfernter Prozeduren wird bei Referenzprozessen abgeändert. Ein Referenzprozeß kann eingehende Prozedurrufe während eines eigenen Prozedurrufs beantworteten. Bei lange andauernden Aktionsfolgen (Zyklen in Methoden) versagt das Schema. Es kann auch nicht entschieden werden, ob die Ausführung einer Aktionsfolge terminiert (generelle Unlösbarkeit des Halteproblems algorithmischer Beschreibungen).
- Man kann davon ausgehen, daß der Zugriff auf eine Referenz schnell erfolgt. Überschreitet der Zugriff ein unbestimmtes Zeitlimit, liegt ein Problem vor. Durch entsprechende Selbstkontrolle basierend auf dem SDL-*timer*-Konzept, kann man derartige Situationen erkennen und eine potentielle Blockierung beim Zugriff durchbrechen. Alternativ kann auch der Referenzprozeß selbst garantieren, daß jedes eingehende Signal nach einer endlichen Zeit verarbeitet wird. Diese Eigenschaft kann jedoch ein SDL-Prozeß für sich selbst nicht garantieren, da interne Aktionsschleifen nicht unterbrechbar sind. Hier müßte das semantische Modell von SDL erweitert werden. Für alle zeitbasierten Unterbrechungsmethoden ist die Bestimmung der zulässigen Antwortzeit ein in der Praxis schwer zu lösendes Problem.

Beide Ergänzungsvarianten sind als massive Eingriffe in die Transformationssemantik von SDL eine höchst komplexe, demzufolge fehleranfällige Erweiterung. Die erreichte Form der Synchronisation kann als „schwach“ bezeichnet werden: die Manipulation des Datenbereichs einer Referenz bleibt exklusiv, sobald Methoden auf andere Referenzen zugreifen ist die Ausführung nicht mehr exklusiv. Diese Semantik wird für die meisten SDL-Programmierumgebungen zutreffend sein.

Sichere Bereiche: Beide Grundvarianten können mit sehr einfachen Schnittstellenerweiterungen[1] auch exklusive Referenzzugriffe realisieren. Hier ist eine große Palette von Modellspezifikationen denkbar. Durch Transformation kann eine geeignete Syntax bereitgestellt werden.

Es sollte beachtet werden, daß Monitorkonzepte immer das Problem von Blockierungen implizieren.

6.2.7 Fazit

Offensichtlich kann man mit gewissem Aufwand benötige objektorientierte Konzepte für Daten mit SDL nachbilden. Man bemerkt auch, daß sich abhängig von der Grundvariante bestimmte Wunschkonzepte mit unterschiedlichem Aufwand realisieren lassen. Deshalb wird man vor einer Entscheidung für oder gegen eine der Varianten einen Katalog der notwendigen Erweiterungen zusammenstellen, möglichst mit Blick auf die prinzipielle Realisierbarkeit. Mit dem Modell als Hintergrund wird die Erweiterung konsistent realisiert. Diese Herangehensweise kann man auch als modellbasierte Weiterentwicklung von SDL bezeichnen. Nach der Festlegung von Syntax und informaler Semantik für Erweiterungen hat man immer noch die Freiheit, ein alternatives Kalkül für die formale Semantik zu wählen.

---

1. z.B. Realisierung von *mutex*-Konzepten

Mit diesen Hintergedanken wurden die Vorschläge auch bei den ITU-T-Sprachentwicklern präsentiert. Aufgrund der vielfältigen Interessen war es jedoch unmöglich, sich auf einen modellbasierten Katalog zu einigen. Vielmehr wurden nach der Präsentation Einzelkonzepte zusammengetragen, mit denen man zwangsläufig auf Probleme bei der Umsetzung mit den Modellen trifft. ITU-T entschied sich daraufhin für eine konzeptbasierte Entwicklung ohne formale Grundlage. Diese Entwicklung wird im nächsten Abschnitt vorgestellt.

6.3 Substitution des Datenkonzepts durch ITU-T

Mit SDL-2000 erhält der Nutzer eine Syntax zur Beschreibung von Datenobjekten, deren Zuweisungen sowohl per Referenz als auch per Wert möglich sind. Die Syntaxelemente zur algebraischen Beschreibung wurden zugunsten verbesserter funktionaler Elemente aus der Sprache entfernt.

Semantisch ist das Datenmodell von SDL-2000 eine Neuentwicklung, das ACT ONE-Modell wird nicht mehr verwendet. Dieser radikale Schritt läßt sich historisch gut nachvollziehen:

- SDL hat einen gewaltigen Schritt in Richtung einer effizient implementierbaren Sprache getan. Es gab Befürchtungen, daß ein ACT ONE-basiertes Datenmodell diese Entwicklung bremst.
- Es ist ein Fakt, daß der SDL-Verhaltensteil ein neues semantisches Modell erhält. Eine einheitliche Repräsentation der Semantik des Verhaltensteils und der Daten fördert das Verständnis der Sprache. Für die Standardisierungsprozedur kann die formale Semantik als Anhang nachgereicht werden.
- Die Lösung von einem formalen Modell bietet mehr Spielraum bei der konzeptionellen Neuentwicklung des Datenmodells. Werkzeughersteller strebten zum Zeitpunkt der Entscheidung ein zu SDL-96 rückwärtskompatibles Modell, bestehend aus einer Kombination von Java- und C++-Elementen, an. Es bestand kein Konsens, daß die rein konzeptbasierten Forderungen gegenläufig zu sowohl der semantischen Fundierung als auch der Werkzeugentwicklung sind.

Dieser Abschnitt gibt einen Einblick zum Stand der noch nicht abgeschlossenen Entwicklung. Prinzipiell gibt es konkrete syntaktische Vorstellungen zur Definition und Verwendung von Daten. Die Semantik ist informal (englischer Text), z.T. auch durch formale Regeln für einzelne Konzepte vorgegeben. Es gibt keine Implementierung. Deshalb gliedert sich der Abschnitt in die Vorstellung der Grundideen und weiterer Einzelkonzepte. Die Darstellung kann bezüglich eines aktualisierteren Sprachstandards falsch sein.

Für eine ausführliche und insbesondere aktuelle Darstellung muß die Sprachentwicklung bei der ITU-T, d.h. das aktuell gültige Z.100-Dokument herangezogen werden.

6.3.1 Grundlegende Eigenschaften

Das ACT ONE-Modell ist nicht mehr Bestandteil der SDL-2000-Sprachdefinition. Deshalb sind alle axiomatischen Beschreibungsmöglichkeiten aus den Syntaxregeln der Datendefinition entfernt worden. Ein Datentyp ist nur noch eine strukturelle Beschreibung für

- Literale,
- Strukturen,
- alternative Feldbelegungen (*choice*) und
- Kommunikationsschnittstellen (Spezialfall für Datentypen).

Für diese Arten von Definitionen sind die Konstruktion von Werten und grundlegende Manipulationsmechanismen (Kopie, Vergleich) erklärt. Daneben gibt es einen Satz vordefinierter Daten, deren Schnittstellen durch ein Modul *Predefined* bestimmt werden. Das Verhalten dieser Daten entspricht per Festlegung in der Sprachdefinition dem der vordefinierten Daten aus SDL-96.

Definition of Datentypen

Bei der Definition eines Datentyps wird festgelegt, mit welcher Semantik Werte zugewiesen werden, entweder per Wert (Wertetyp) oder per Referenz (Objekttyp). Dazu ist anstelle des SDL-Schlüsselworts **newtype** die Kombination **value type** für Wertzuweisungen bzw. **object type** für Referenzzuweisungen zu verwenden.

Beispiel: Die Definition einfacher Literale mit Wertesemantik erfolgt mit

```
value type SimpleLiterals
    literals a,b,c;
endvalue type;
```

bzw. auch verkürzt mit einer an C++ erinnernden Notation

```
value type SimpleLiterals { literals a,b,c; };
```

Für Strukturen ist oft eine Referenzsemantik gefordert:

```
object type SimpleStructure;
    struct /* alternative Felder mit choice */
        i   Integer;
        b   Boolean optional;
endobject type;
```

Die Festlegung der Zuweisungssemantik kann bei der Bezugnahme auf den Datentyp geändert werden.

Beispiel: Bei der Definition einer Variable kann das Zuweisungsverhalten neu festgelegt werden.

```
dcl val value SimpleStructure;
```

Für Wertetypen ist ein strenges Zuweisungskonzept definiert. Prinzipiell entsprechen Sorten beim Übergang nach SDL-2000 den Wertetypen. Das soll die Rückwärtskompatibilität zu SDL-96 sichern. Auf Grund der Objekttypen gibt es neue Regeln für Zuweisungen von Referenzen bzw. Zuweisungen mit Wert-/Referenzkombination.

Innerhalb von Datendefinitionen können Operatoren und Methoden definiert werden. Operatoren dürfen im Gegensatz zu Methoden keine Seiteneffekte bei Argumenten verursachen. Die Intension (und informale Forderung) ist, daß Operatoren nur „neu konstruierte Werte" zurückgeben. Argumente können sowohl per Referenz als auch als *in/out*-Parameter (nur für Variablen) übergeben werden. Methoden besitzen ein implizites *in/out*-Argument des definierenden Datentyps.

Die Spezifikation des Verhaltens erfolgt algorithmisch wie in SDL-96 oder mit einer C++-ähnlichen Syntax.

Beispiel:

```
object type SimpleStructure;
    struct /* alternativ choice */
        private i Integer default 0;
        b   Boolean;
    methods Inc_i(Integer); /* kein Rückgabewert */
    method Inc_i(inc Integer) { i := i+inc; }
endobject type;
```

Wie bereits im Beispiel demonstriert, gibt es C++-ähnliche Attribute für die Sichtbarkeit innerer Definitionen. Es ist auch möglich, Datendefinitionen zu verschachteln (lokale Definitionen in Datentypen).

Definition von Kommunikationssschnittstellen

Kommunikationsschnittstellen sind aus *interface*-Beschreibungen implizit definierte Ableitungen der Sorte *PId*. Mit dieser Konstruktion kann an einem *PId*-Wert getestet werden, ob der durch diesen Wert adressierte Empfänger ein Signal verarbeiten kann. Eine vorhandene Signalroute als auch der Empfang werden jedoch nicht garantiert. Da mehrere Schnittstellen zusammengefaßt werden können, unterstützen Kommunikationsschnittstellen im Gegensatz zu den anderen Datendefinitionen Mehrfachvererbung. Die implizite Vergabe des *interface*-Namens als Datentypname erfordert zusätzliche Einschränkungen bei der Vergabe von Namen für Daten, Agenten und deren Schnittstellen.

6.3.2 Vererbung

Datendefinitionen können (einfach) vererbt werden. Eine Ableitung darf die Art der Definition nicht ändern (z.B. keine Einführung von Strukturelementen in Literaldefinitionen), wohl aber die Art der Zuweisungssemantik:

Beispiel:

```
value type ExtendedStructure inherits SimpleStructure;
    struct
        c   Charstring;
endvalue type;
```

Der Wechsel der Zuweisungssemantik ist für Parameter von Methoden sehr problematisch. Erfolgt keine Redefinition, können Zuweisungen des Basistyps im Kontext der Ableitung statisch falsch sein. Das ist ein sehr ungewöhnlicher Effekt für Vererbung, mit Blick auf Kontextparameter in SDL aber nicht unbekannt.

Beispiel: Angenommen, in *SimpleStructure* gäbe es eine Kopiermethode:

```
method MyCopy() returns SimpleStructure {
    dcl ret value SimpleStructure := this;
    return ret;
}
```

Im Kontext von *ExtendedStructure* ist die Initialisierung von *ret* falsch.

Innerhalb einer Datendefinition kann eine Referenz auf eine umgebende Datendefinition mit dem Schlüsselwort **this** gekennzeichnet werden. Wird die umgebende Definition vererbt, werden alle gekennzeichnete Bezeichner durch den Namen der neuen Definition ersetzt. Dieses flexible Konzept ersetzt die SDL-96-Regel für kovariante Operatorsignaturen bei Vererbung. Es ist mit der englischen Sprachdefinition im Z.100 nicht geklärt, welche internen Strukturen eigentlich von der Ersetzung betroffen sind. Einerseits geht man von einer kovarianten Ersetzung der Datendefinition aus und andererseits strebt man den Verzicht auf ein Ersetzungsmodell bei Vererbung an.

Alle Datentypen besitzen einen gemeinsamen, abstrakten Basistyp *Any*. Dieser stellt in allen Datentypen verfügbare Operationen und Methoden bereit.

```
abstract object type Any
    operators
        equal (this Any, this Any) -> Boolean;
        clone (this Any) -> this Any;
    methods
        virtual is_equal(this Any) -> Boolean;
        virtual copy(this Any)->this Any;
endobject type;
```

Man kann davon ausgehen, daß die vorgegebenen Methoden für spezielle Datenkonstruktionen vordefiniert sind. Es ist nicht ganz klar, unter welchen Umständen eine Datendefinition diese Methoden redefinieren kann.

6.3.3 Zuweisungen mit Referenzen

Die Zuweisung von Objekttypen erfolgt polymorph, d.h. Zuweisungen von Referenzen in Richtung Basistyp sind ohne Informationsverlust möglich.

Beispiel:

```
dcl s  object SimpleStructure,
    e  object ExtendedStructure := (. 42, True, 'Hello world!' .);
...
task s := e;   /* eine polymorphe Referenzzuweisung */
```

Die umgekehrte Richtung ist problematisch. Es wäre ein dynamischer Typfehler, wenn eine Referenz zugewiesen wird, die nicht die erwarteten Objekteigenschaften aufweist. Aufgelöst wird dieser Fehler durch Zuweisung des Werts *Null*; es wird keine Ausnahme geworfen. In anderen Programmiersprachen, z.B. C++ wären derartige Zuweisungen statisch falsch, es müßte eine explizite *cast*-Operation spezifiziert werden.

Beispiel:

```
task e := s;   /* ein Zuweisungsversuch */
```

Kombiniert man in einer Zuweisung einen Wert *w* mit einer Referenz *r*, so gilt für die Zuweisungen:

- **task** w := r;
  Der dynamische Typ von *r* muß dem statischen Typ von *w* entsprechen, sonst wird eine vordefinierte Ausnahme geworfen.
- **task** r := w;
  Aus dem Wert von *w* wird per Kopie eine Referenz erzeugt, die an r mit den entsprechenden Regeln zugewiesen wird.

Demzufolge gibt es in SDL-2000 keine Referenzen auf Variablen.

Für *PId*-Werte (z.B. der impliziten Variablen **sender**) kann mit einem Zuweisungsversuch an eine Variable einer Kommunikationsschnittstelle festgestellt werden, ob die Adresse eine bestimmte Schnittstelle unterstützt. Das ist eine spezielle Form polymorpher Zuweisungen.

6.3.4 Referenzkonstruktionen und Gültigkeit

Referenzen besitzen einen *default*-Wert: *Null*. Dieser Wert zeigt an, daß kein Objekt an die Referenz gebunden ist. Einen nicht initialisierten Datenbereich für eine Referenz erhält man durch den Aufruf des Operators *Make* ohne Argumente. Es ist Aufgabe des Anwenders, den Datenbereich entsprechend der vorgegebenen Spezifikation konsistent zu initialisieren. Man kann auch auf vordefinierte Konstruktionen (z.B. für Strukturinitialisierung) zurückgreifen. Der den Datenbereich erzeugende SDL-Agent wird der Besitzer.

Eine Referenz wird grundsätzlich repliziert per Signal oder eine auf Signalen aufbauende Aktion an einen anderen Agenten übertragen. Der Empfänger wird Besitzer der neu erzeugten Referenz. Folglich sind der Zugriff auf Referenzen und deren Lebenszeit an genau eine aktive Instanz gebunden.

Interessanterweise ist die Verwendung von Referenzen für konstante Ausdrücke erlaubt. Die Berechnung der Referenz erfolgt im Agenten, der die Referenz nutzt. Insbesondere erfolgt die Berechnung bei jedem Zugriff auf den konstanten Ausdruck.

Beispiel: **synonym** int_ref **object** Integer := 1;

Der Test *int_ref = int_ref* liefert (mit großer Wahrscheinlichkeit) in SDL-2000 das Ergebnis *False*.

Diese Art der Berechnung und mögliche Seiteneffekte sollten bei Ausdrücken vermieden werden, deren Auswertung unbestimmt ist, z.B. bei

- bedingten Fortsetzungen (*enabling condition*),
- Verzeigungen (*range*-Ausdruck),
- optimierbaren Ausdrücken (True or ...).

Möglicherweise ergeben sich aus der ausstehenden, formalen Semantik noch entsprechende Einschränkungen.

6.3.5 Bindung von Methoden und Operatoren

Operatoren- als auch Methodenrufe werden zuerst statisch aufgelöst. Dazu werden die am Ruf beteiligten Werte mit allen möglichen Signaturen verglichen. Wegen der Polymorphie von Referenzen kann der Vergleich nicht exakt durchgeführt werden, es wird die Definition mit den geringsten Abweichungen (eine komplexe, polymorphe Metrikdefinition) ermittelt. Gibt es mehrere Möglichkeiten für die Signaturen, dann liegt bei statisch zu bindenden, also nicht virtuellen Methoden ein Fehler vor. Bei dynamischer Bindung bilden die zusammengehörigen virtuellen Methodensignaturen eine Menge. Bilden die berechneten Signaturvarianten keine Teilmenge der virtuellen Signaturmengen, liegt ebenfalls ein Fehler vor.

Durch die Spezifikation von Virtualität kann dynamische Bindung von Methoden erreicht werden. In der derzeitigen Version des neuen Standards können sowohl die Signaturen als auch die Argumente virtuell gekennzeichnet werden. Aus dem englischen Text ist nicht zu erkennen, wie die dynamische Bindung erfolgt, man kann jedoch davon ausgehen, daß dynamische Bindung über mehrere Argumente (*multi-dispatch*) möglich ist. Unklare Details sind u. a.:

- die exakte Funktion der virtuellen Argumente,
- Aufrufe von virtuellen Methoden mit Argumenten, die bei Vererbung ihre Zuweisungscharakteristik verändert haben,
- Verwendung von *Null* als (implizites) Argument in Kombination mit später Bindung,
- Auswirkung der Regel für Zuweisungsversuche bei Argumenten.

6.3.6 Fazit zur ITU-T-Lösung

Bisher gibt es eine englische Beschreibung für das Datenmodell, die in den neuen Sprachstandard eingearbeitet ist und formale Semantikregeln, die bisher nicht verifiziert wurden. Für viele Konzepte, die syntaktisch vorgesehen sind, gibt es in der Beschreibung nur ungenügende oder keine Erklärungen, so daß man die Korrektheit komplizierterer Beispiele nicht mehr bestimmen

kann. Entsprechende SDL-Werkzeuge sind wegen der umfangreichen Änderungen nicht verfügbar, es gibt kaum einen Nachweis über die Implementierbarkeit einzelner Konzepte. Damit konnte auch das konsistente Zusammenspiel der neuen, durchaus komplizierten Einzelkonzepte in einer großen SDL-Beschreibung bisher nicht getestet werden.

Wenn SDL-2000-Werkzeuge verfügbar sind, werden diese insbesondere im Bereich der Datentypen beim jetzigen Stand der Entwicklung eine unterschiedliche dynamische und vermutlich sogar statische Semantik implementieren. Gründe sind die Komplexität und unklare Definitionen von Konzepten. Diese Entwicklung ist für die lange Tradition von SDL als formale Beschreibungstechnik unverständlich.

Mit dem Konzept der Wertetypen wurde eine für die meisten Anwendungen ausreichende Rückwärtskompatibilität zu Sorten erreicht. Lediglich axiomatische Beschreibungen müssen von Hand in algorithmische transformiert werden. Die „einfache“ Verwendung der neuen Konstrukte wird die meisten Erwartungen eines SDL-Anwenders an eine objektorientierte Sprache erfüllen.

Die Erwartungen an das SDL-2000-Datenkonzept waren bei der Entscheidung, konzeptbasiert zu entwickeln, sehr hoch. Bei der Fixierung des Standards fielen immer mehr Konzepte durch offensichtliche Widersprüche aus der Sprachdefinition heraus. Beispielsweise gibt es im Standard keinen Hinweis auf eine Speicherverwaltung, die während der Entwicklung als wichtig eingestuft wurde.

Arbeiten von Löwis [Löw00] beschäftigen sich damit, wie ein formales Modell für SDL-Daten mit der allgemeinen Methodik zur semantischen Fundierung von SDL-2000 [Pri01] definiert werden kann. Man wird mit diesen formalen Ansätzen garantiert auf Probleme stoßen, die offene oder auch widersprüchliche Sprachkonzepte betreffen. An dieser Stelle können die modellbasierten Ansätze dieser Arbeit durchaus zur Klärung der Probleme herangezogen werden.

6.4 Einschätzung der neuen Datenkonzepte

Es ist aber nicht das Anliegen dieser Arbeit systematisch konzeptüberführende Vergleiche zwischen den dargestellen Erweiterungsvarianten anzustellen. Objektorientierte Erweiterungen sind jedoch eine mögliche Verbesserung der Daten in SDL, weshalb eine Einschätzung der Varianten basierend auf den allgemeinen Kriterien zur Verbesserung des Datenkonzepts aus Abschnitt 3.3.2 „*Sprachunabhängige Erweiterungen*” (S.46) vorgenommen wird.

1. *Eine verbesserte Notation für Datentypen sollte sich harmonisch in die zentralen Strukturierungskonzepte Vererbung, Parametrisierung und Sichtbarkeit einpassen.*

   Sammelobjektverwaltung: Die Datentypen werden wie bisher genutzt. Referenzen sind prinzipiell auch nur Datentypen, die sich in die vorhandenen Strukturierungskonzepte hervorragend integrieren.

   Einzelobjektverwaltung: Das Grundmodell muß mit zwei unterschiedlichen Datenstrukturen auskommen. Im Abschnitt „*Veränderte Zuweisungen*” (S.113) wird die Situation mit Hilfe einer komplexen Transformation verbessert. Die vordefinierten (primitiven) Daten bleiben ACT ONE-Sorten und stehen damit außerhalb der Beschreibung für Objekte. Das ist eine übliche (C++, Java) Einschränkung.

   ITU-T-Substitution: Die zu SDL passende Strukturierung der Daten ist eine der Stärken des neuen Konzepts. Das neue Konzept der Kommunikationsschnittstellen paßt noch nicht optimal zum Gesamtbild der Daten (zweite Variante für Datenbeschreibungen) und letztendlich der Sprache (Mehrfachvererbung, keine Empfangsgarantie).

2. *Neue Datenkonstruktionen sollten eindeutig definiert werden, damit SDL nicht die Qualität einer formalen Sprache verliert.*

    Sammel- und Einzelobjektverwaltung: Die Transformationsregeln sind detailliert als Text formulierbar. Eine Einarbeitung in die bisherige formale SDL-Sprachdefinition (s. Abschnitt 1.3 „*Definition der Sprache SDL im Sprachstandard*" (S.10)) ist jedoch unrealistisch. Grundsätzlich lassen sich Antworten zu allen potentiellen Fragestellungen der Anwender aus den Modellen ableiten. Möglicherweise entsprechen diese Antworten nicht den Vorstellungen, weshalb beide Modelle sehr flexibel bezüglich Änderungen sind.

    ITU-T-Substitution: ITU-T hat bisher kein gesichertes, formales Modell für die neuen Datenkonzepte entwickelt.
    Im Gegensatz zu den modellbasierten Entwicklungen gibt es relativ detaillierte Vorstellungen zu den Möglichkeiten der Einzelkonzepte, ein insgesamt funktionierendes semantisches Modell fehlt jedoch. Möglicherweise sind nicht alle Konzepte von SDL-Werkzeugen realisierbar.

3. *Bereits vorhandene Datendefinitionen sollten ohne großen Aufwand auf die neu zu definierenden Konzepte abbildbar sein.*

    Sammel- und Einzelobjektverwaltung: Die transformationsbasierten Konzepte sind vollständig rückwärtskompatibel. Es ist aber sinnvoll, die Verwendung von Sorten einzuschränken.

    ITU-T-Substitution: In der ITU-T-Sprachdefinition ist eine Prozedur zum Umbau vorhandener Sortendefinitionen in Wertedatendefinitionen angegeben, die jedoch axiomatisch definierte Sorten ausschließt. Die semantischen Eigenschaften (Zuweisungen, vordefinierte Daten) der Wertetypen sind mit denen der Sorten vergleichbar. Man hat die Hoffnung, daß diese Art der Kompatibilität den praktischen Kompatibilitätsanforderungen genügt.

4. *Alle Bestandteile einer Datendefinition, die inhaltlich zu dieser Datendefinition gehören, insbesondere Operatoren, sollten in dieser Datendefinition und nur dort erfaßt werden.*

    Sammel- und Einzelobjektverwaltung, ITU-T-Substitution: Bei allen Vorschlägen orientiert sich die Datenbeschreibung an die der Sorten. Zusätzlich gibt es Methoden, die aber eine besondere Bindung an den definierenden Kontext haben. Es ist weiterhin möglich, Operationen in beliebige Datendefinitionen zu verlagern. Man kann aber wegen der flexibleren Zuweisungskonzepte vermutlich auf „externe" Operatoren verzichten.

5. *Die Möglichkeiten zur strukturellen Beschreibung von Sorten sollten erweitert werden.*

    Sammelobjektverwaltung: Bis auf die zusätzlichen Referenzkonstruktionen wurde an der Definition von Sorten nichts verändert. Man ist auf weitere Verbesserungen angewiesen.

    Einzelobjektverwaltung: Die Einführung von Objektdefinitionen in Verbindung mit Vererbung ist eine wesentliche strukturelle Verbesserung. Es ist lediglich eine Frage des Aufwands, auch andere Strukturverbesserungen zu berücksichtigen.

    ITU-T-Substitution: Durch den Neuentwurf der Datendefinitionen kann man von einer grundsätzlichen Verbesserung der Beschreibungsmöglichkeiten ausgehen.

6. *Neue Konstruktionen sollten durch SDL-Werkzeuge implementierbar sein.*

    Sammelobjektverwaltung: Bisher wurden nur die Sammelobjektverwaltung prototypisch implementiert und damit nachgewiesen, daß die Ausführung eines SDL-Prozesses mit Referenzen auf der Basis von C++ möglich ist. Implementierungsprobleme mit ACT ONE-basierten Datenbeschreibungen bleiben bestehen.

Einzelobjektverwaltung: Wegen der neuen Struktureinheit für Referenzobjekte ist der Implementierungsaufwand recht hoch. Prinzipiell unterscheiden sich aber Referenzobjekte wenig von den Klassenobjekten der Programmiersprachen, so daß man die Frage nach der Implementierbarkeit positiv beantworten kann. Da die Grundvariante ACT ONE-basierte Sorten nicht verbietet, bleibt die bisherige Implementierungsproblematik erhalten. Die in Abschnitt „*Veränderte Zuweisungen*" (S.113) vorgeschlagenen Erweiterungen umgehen auch dieses Problem.

ITU-T-Substitution: Wegen der informalen Beschreibung des ITU-T-Modells ist es äußert schwierig abzuschätzen, wie eine Implementierung im Detail funktionieren kann. Bei der Realisierung von Methoden gibt es prinzipielle Bedenken zur Komplexität des Modells.

7. *Es sollte Sprachelemente geben, die Eigenschaften eines SDL-Programms als Teil einer realen Systemumgebung unterstützen.*

Sammel- und Einzelobjektverwaltung, ITU-T-Substitution: Der Kommunikationsaspekt wird von keinem neuen Konzept direkt abgedeckt. Allerdings erleichtert die Einführung von Referenzen die Integration externer Beschreibungen, die letztendlich auch Kommunikation realisieren können, erheblich.

Fazit und Ausblick

SDL wurde als formale Spezifikationstechnik entwickelt. Diesem formalen Aspekt wird auch das Datenkonzept gerecht. Mit der fortschreitenden Entwicklung der Computersprachen entstand der Wunsch, die Kluft zwischen der Spezifikation eines Systems und seiner Implementierung zu schließen. SDL-Nutzer erwarten, daß eine Spezifikationen ausgeführt werden kann. Die Sprache wird zunehmend mit modernen Programmiersprachen verglichen. Mit diesem Hintergrund entstehen ganz neue Anforderungen, die SDL in der Version SDL-96 insbesondere mit Blick auf das Datenkonzept nicht erfüllt.

In der vorliegenden Arbeit werden bei einer systematischen Betrachtung des Datenkonzepts diverse Mängel herausgearbeitet. Einige dieser Mängel sind von allgemeiner Natur, beispielsweise die eingeschränkte Implementierbarkeit von Analysewerkzeugen für SDL. Andere ergeben sich aus den neuen Anforderungen an die Sprache, z.B. erwartet man in einer objektorientierten Sprache ein Referenzkonzept mit polymorphen Zuweisungen. Diese zwei wichtigen Konzepte fehlen den Daten, obwohl SDL objektorientiert ist. Deshalb wird in der Arbeit ein sehr allgemeiner Katalog zur Veränderung der Sprache SDL bezüglich des Datenkonzepts basierend auf den erkannten Problemen und auf allgemeingültigen Richtlinien der SDL-Sprachentwicklung erstellt.

Neben Anforderungen an neue Datenkonzepte wird auch eine systematische Herangehensweise für Sprachänderungen entworfen. Bereits bei diesen methodischen Betrachtungen lassen sich Eigenschaften ableiten, die für eine spätere Einschätzung konkreter Änderungen von Bedeutung sind.

SDL-Werkzeuge haben sich auf die vorhandene Nachfrage eingestellt und bieten unterschiedliche Lösungen an, um Daten besser zu spezifizieren. Der SDL-Anwender muß für sich entscheiden, ob die präsentierte Werkzeuglösung mit ihren Vor- und Nachteilen zweckmäßig ist. Der aufgestellte Anforderungskatalog gibt dazu eine hilfreiche Orientierung. In der Arbeit werden immer unter dem besonderen Blickwinkel der Implementierbarkeit diverse SDL-Modifikationen aus unterschiedlichen Anwendungsgebieten vorgestellt und anhand des Anforderungskatalogs bewertet. Die Veränderungen befassen sich mit dem Einsatz von Programmiersprachen und alternativen Beschreibungstechniken sowie der Entwicklung von SDL-spezifischen neuen Datenkonzepten. Insbesondere die Erfahrungen des Autors aus der SITE-Entwicklung und der Mitarbeit an der SDL-Sprachentwicklung bei der ITU-T fließen in diese Betrachtungen mit ein.

So wie jede vorgestellte Verbesserung am Datenkonzept hat auch eine zukünftige SDL-2000-Implementierung Schwächen für spezifische Anwendungen. Die Richtung, eine verbesserte SDL-Version mit alternativen Spezifikationstechniken zu kombinieren, scheint vielversprechend. Die Kombination mit ASN.1 ist im Telekommunikationsbereich bereits traditionell. Eine ODL-Kombination hat wegen der besseren Standardkonformität ebenfalls gute Chancen für zukünftige Anwendungen.

Der nächste Schritt im SDL-Umfeld ist die Bereitstellung von Werkzeugen, die SDL-2000 unterstützen. Es zeigt sich immer wieder, daß bei der Programmgenerierung aus SDL die volle Mächtigkeit der Sprache nicht benötigt wird und z.T. aus Effizienzgründen auch nicht erwünscht ist. Deshalb wird die Unterstützung für SDL-2000 ein langsamer, werkzeugspezifischer Übergang von SDL-96 nach SDL-2000 sein. Die Unterstützung des *exception*-Konzepts mit SITE ist ein Beispiel dieses Übergangs. Mit der Bereitstellung von Werkzeugen, die auch verbesserte objektorientierte Daten unterstützen, ergeben sich neue Forschungsaspekte.

Unterstützung der neuesten ASN.1-Version:

Eine Anpassung der SITE-Werkzeuge an den neuesten Standard wurde mit der Realisierung einer X.680-konformen Syntaxanalyse für ASN.1 durch studentische Arbeiten begonnen. Auch hier werden die von SITE realisierten Kommunikationsaspekte der Kombination von SDL mit ASN.1 ein bedeutender Implementierungsaspekt sein, obwohl diese Anwendung von der ITU-T noch immer nicht standardisiert worden ist.

Zusammenspiel neuer SDL-2000-Konzepte mit alten SDL-Spezifikationen:

Es ist für industrielle Anwendungen unumgänglich, daß alte SDL-Spezifikationen mit neuen zusammenarbeiten können. Eine formale Transformation alter Spezifikationen nach SDL-2000 wird keine vollständige Lösung sein, da durch die Einführung des Referenzkonzepts in SDL-2000 dem Nutzer ein erhebliches Optimierungspotential in die Hand gegeben wurde. Dieses Potential möchte er natürlich auch in Kombination mit alten Spezifikationen nutzen. Die Entwicklung einer Übergangsmethodologie wird mit der Verfügbarkeit erster SDL-2000-Werkzeuge sicher ein spannendes Forschungsfeld.

Eingeschränkte Codegenerierung mit hoher Optimierung:

Zunehmend wird SDL zur Spezifikation sogenannter eingebetteter Systeme verwendet. Die Programme dieser Systeme laufen als Betriebssystem in einer begrenzten Hardwareumgebung, oft sogar unter Echtzeitbedingungen. Ein typisches Beispiel ist die Software von mobilen Endgeräten.

Die automatische Codegenerierung, insbesondere die der Daten, ist für derartige Systeme wegen der begrenzten Ressourcen äußerst problematisch. Wohlüberlegte Einschränkungen von SDL und gute Optimierungsstrategien sind hier gefragt. Möglicherweise führt das Referenzkonzept für Daten auch zu neuen Strategien sowohl bei der Spezifikation als auch der Implementierung. Die theoretischen Erkenntnisse könnten durch die flexible SITE-Technologie schnell in die Praxis umgesetzt werden.

Automatische Ableitung eines Referenzinterpreters für SDL:

Trotz der gegenwärtigen Defizite bei der formalen Definition der Daten ist zu erwarten, daß für SDL-2000 eine formale Basis entwickelt wird. Auf dieser neuen semantischen Basis hat Prinz [Pri01] die Möglichkeit einer automatischen Ableitung von Werkzeugen untersucht, die eine SDL-Beschreibung syntaktisch und semantisch analysieren und schließlich interpretieren. Die freie Verfügbarkeit derartiger, funktionstüchtiger Referenzwerkzeuge wäre ein großer Schritt vorwärts, wenn es um die Standardkonformität von SDL-Werkzeugen geht.

Normative, universelle Kommunikationsschnittstellen:

Eine mit SDL-2000 realisierte IDL/ODL-Integration ist sicher eine interessante Implementierungsaufgabe. Allerdings wird nicht jeder Anwender von SDL eine IDL/ODL-Schnittstelle zu einem SDL-System als adäquate Lösung ansehen, da hier eine CORBA-Architektur vorausgesetzt wird. Es ist durchaus denkbar, daß Schnittstellen zu einer Gruppen von Applikationen, beispielsweise Datenbanken oder auch sehr speziell nur Namensdienste, entwickelt und möglicherweise standardisiert werden. Analog zur Realisierung von Protokollschichten können das Module sein, die geeignete parametrisierbare Definitionen enthalten. Der Forschungsschwerpunkt liegt hier auf der Universalität der Module.

Anhänge

A Abkürzungen

Namen von Sprachen, Organisationen und einige wenige Protokollbegriffe, die in der Arbeit benutzt werden, sind abgekürzt. Sofern Werkzeuge unter ihrer Kurzform bekannt sind, kommt auch diese zur Anwendung. Die Übersicht enthält alle verwendeten Abkürzungen mit den entsprechenden Langformen.

ACT ONE	Algebraic specification techniques for Correct design of Trusty software systems (version 1)
ASN.1	Abstract Syntax Notation One
BER	Basic Encoding Rules
CCITT	siehe ITU-T
CDR	Common Data Representation
CER	Canonical Encoding Rules
CIF	Common Interchange Format
CORBA	Common Object Request Broker Architecture
CR	Common Representation
CSP	Communication Sequential Processes
DER	Distinguished Encoding Rules
ETSI	European Telecommunication Standard Institute
HTML	Hypertext Markup Language
IDL	Interface Definition Language
INAP	Intelligent Network Application Protocol
ITU-T	International Telecommunication Union - Telecommunication Standardization Sector, ehemals CCITT
ODL	Object Definition Language
OMG	Object Management Group
OSI	Open System Interconnection
PDU	Protocol Data Unit
PER	Packed Encoding Rules
RTTI	Run Time Type Information
SDL	Specification Description Language, Synonym für SDL-96
SDL/GR	SDL Graphic Representation

SDL/PR	SDL Phrase Representation
SDL-<Jahr>	SDL standardisiert im Jahre <Jahr>
SITE	SDL Integrated Tool Environment
UML	Unified Modeling Language

Übliche Abkürzungen in der deutschen Sprache werden nur verwendet, wenn die Langform vom eigentlichen Inhalt ablenken würde.

B Literatur

AFW92 Klaus Ahrens, Joachim Fischer, Dorotha Witaszek: Objektorientierte Prozeßsimulation in C++. Informatik-Preprint der Humboldt-Universität zu Berlin, Berlin, 1992.

AFW93 Klaus Ahrens, Joachim Fischer, Dorotha Witaszek: Eine Laufzeitbibliothek für SDL'92. Informatik-Preprint der Humboldt-Universität zu Berlin, Berlin, 1993.

ASU92 Alfred v. Aho, Ravi Sethi, Jeffrey D. Ullman: Compilerbau. Addison-Wesley GmbH, Deutschland, Band I und II, 1992.

BGKS+95 Ulf Behnke, Michael Geipl, Gerd Kurzbach, Ralf Schröder, Nils Fischbeck, Renee Mundstock: Development of broadband ISDN telecommunication services using SDL'92, ASN.1 and automatic code generation. In Participation Proceedings of 8th international conference of Formal Description Techniques for Distributed Systems and Communication Protocols FORTE'92, Montreal, Canada, 1995.

BHS91 F. Belina, D. Hogrefe and A. Sarma: SDL With Applications from Protocol Specification. Carl Hanser Verlag and Prentice Hall International (UK), Ltd., Hertfordshire, Great Britain, 1991.

Bie97 Frank Bielig: Implementierung einer SDL-Laufzeitbibliothek auf CORBA-Basis. Diplomarbeit, Institut für Informatik, Humboldt-Universität zu Berlin, 1997.

BjJo82 Dines Bjørner, Cliff B. Jones: Formal specification and software development. Prentice Hall Publ., 1982.

BLS98 Harald Böhme, Martin v. Löwis, Ralf Schröder: Final Report of Project SITE2NT. Technical Report (confidential), Department of Comuter Science, Humbold University Berlin, 1998.

Böh97 Harald Böhme: Objektorientierte Codegenerierung für SDL'92. Diplomarbeit, Institut für Informatik, Humboldt-Universität zu Berlin, April 1997.

BoWi96 Mark Born, Mario Winkler: SDL-92, CORBA-IDL und TINA-ODL im ODP-Entwurfsprozeß. Diplomarbeit, Institut für Informatik, Humboldt-Universität zu Berlin, 1996.

C++98 ISO: C++. International Standard ISO/IEC 14882, Genf 1998.

Cho96 Chorus Systems: CHORUS/COOL ORB r4 Product Description. Technischer Bericht CS/TR-96-207.3 CHORUS,1996.

Cin99 Cinderella: Cinderella SDL. Technische Beschreibung, Dänemark, 1999.

Cis99 Daniel Cisowski: Ein Editor zur Dokumentation von SDL Patterns. Diplomarbeit, Fachbereich Informatik, Universität Kaiserslautern, 1999.

CLS01 V. Courzalis, M. v. Löwis, R. Schröder: Using SDL in a Stateless Environment. In Proceedings of 10th International SDL Forum SDL 2001: Meeting UML, Editors: Rick Reed, Jeanne Reed, Springer, Copenhagen, Denmark, June 2001.

Dum96 Carsten Dumke: Lexikalische Analyse von SDL-92 mit ASN.1 (Z.105) und theoretische Grundlagen der Fehlerbehandlung. Studienarbeit, Institut für Informatik, Humboldt-Universität zu Berlin, 1996.

Dum99 Carsten Dumke: Grundlagen und Technologie einer CIF-Analysekomponente für SDL. Diplomarbeit, Institut für Informatik, Humboldt-Universität zu Berlin, 1999.

EhMa85 Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specification (Bd. 1). Springer Verlag Berlin Heidelberg New York Tokyo, 1985.

EHS97 Jan Ellsberger, Dieter Hogrefe, Armadeo Sarma: SDL - formal object-oriented language for communicating systems. Prentice Hall, 1997.

EiBe96 Peter van Eijk, Axel Belinfante: The term processor Kimwitu - Manual and cookbook (version 4.0). Technical report, University of Twente, Enschede, Netherlands, 1996.

EN301 ETSI: Intelligent Network (IN); Intelligent Network Application Protocol (INAP); Capability Set 2 (CS2); Part 1: Protocol specification. (Draft) European Standard (Telecommunications series), version 1.1.1 (1998-03), Sophia Antipolis, France, 1998.

ETSI93 ETSI Project Team 37: Use of SDL in Eropean Telecommunication Standards (Rules for testability and facilitating validation). Delivarable 4 Version 1.0 (unpublished), ETSI Secretariat, Sophia Antipolis Cedex, France, September 1993.

FiAh96 Joachim Fischer, Klaus Ahrens: Objektorientierte Prozeßsimulation in C++, Addison-Wesley GmbH, 1996.

FiHo93 Joachim Fischer, Eckardt Holz. Towards an object-oriented technology for specifications and implementations of distributed systems. In Proceedings of TOOLS-USA, Englewood Cliffs, 1993. Prentice-Hall. TOOLS‘II conference.

Fis95 Nils Fischbeck: Automatisches Layout für SDL/GR. Diplomarbeit, Institut für Informatik, Humboldt-Universität zu Berlin, 1995.

FiSc93 Joachim Fischer, Ralf Schröder: Combined Specification using SDL and ASN.1 in SDL‘93 Using Objects, Proceedings of the Sixth SDL Forum Darmstadt, Germany, Seite 293-304, Ove Faergemand and Amardeo Sarma (editors), North-Holland, 1993.

GGP99 Uwe Glässer, Reinhard Gotzhein, Andreas Prinz: Towards a new formal SDL semantics based on Abstract State Machines. In SDL‘99 The Next Millennium, R.Dssouli, G.v.Bochmann, Y.Lahav (editors), proceeding of the Ninth SDL Forum Montreal, Canada, Elsevier, 1999.

Gora98 Walter Gora: ASN.1, Abstract Syntax Notation. FOSSIL-Verlag, 1998.

HaPe91 Øystein Haugen, Birger Møller-Pedersen: Tutorial on Object-Oriented SDL. Technical report of Norwegian Computing Center Oslo, Norway, 1991.

HKV97 P. Hakansson, J. Karlsson, L.Verhaard: Combining SDL and C. In SDL‘97 Time for Testing SDL, MSC and Trends, A. Cavalli, A.Sarma (editors), pages 383-396, ELSEVIER, Proceedings of the Eight SDL Forum Evry, France, 1997.

Hoa85 C.A.R. Hoare: Communicating Sequential Processes. Prentice Hall, 1985.

KLS99 Gerd Kurzbach, Martin v. Löwis, Ralf Schröder: External Communication with SDL systems. In SDL‘99 The Next Millennium, R.Dssouli, G.v.Bochmann, Y.Lahav (editors), proceeding of the Ninth SDL Forum Montreal, Canada, Elsevier, 1999.

Kün97 Ralf Kühnel: Die Java™ 1.1 Fibel. Addison Wesley Longman, Bonn, 1997.

Kur94 Gerd Kurzbach: Ein syntaxorientierter Editor für SDL-92. Diplomarbeit, Institut für Informatik, Humboldt-Universität zu Berlin, 1994.

LCI92 Marl A. Linton, Paul R. Calder, and John A. Interrante: InterViews Reference Manual Version 3.1., 1992.

Löw00 Martin v. Löwis: Datentypsemantik für SDL-2000, FBT'2000 Formale Beschreibungstechniken für verteilte Systeme, 10. GI/ITG Fachgespräch in Lübeck, 2000.

LöSc98a Martin v. Löwis of Menar, Ralf Schröder: Object oriented data concepts for SDL. In Proceedings of the 1st workshop of the SDL Forum Society on SDL and MSC (volume I), Y.Lahav, A.Wolisz, J.Fischer, E.Holz (editors), pages 45-54, Informatikberichte der Humboldt-Universität zu Berlin, 1998.

LöSc98b Objekt-orientierte Datenkonzepte für SDL. In Hartmut König, Peter Langendörfer (Hrsg.): FBT'98 Formale Beschreibungstechniken für verteilte Systeme, 8.GI/ITG Fachgespräch in Cottbus, Juni 1998, Shaker Verlag Aachen 1998.

Lut93 Andreas Lutsch: Ein Parser für SDL. Studienarbeit, Institut für Informatik, Humboldt-Universität zu Berlin, 1993.

Lut94 Andreas Lutsch: Codegenerierung für SDL'92. Diplomarbeit, Institut für Informatik, Humboldt-Universität zu Berlin, 1994.

LWF+00 Martin v. Löwis, Dorotha Witaszek, Joachim Fischer, Klaus Ahrens, Bertram Neubauer: Architektur von SDL-Laufzeitsystemen in C++. Informatik-Bericht Nr. 141, Humboldt-Universität zu Berlin, Berlin, 2000.

Mae93 Kai Uwe Maetzel: Interpretation von Prozess-Schemen, Diplomarbeit, Institut für Informatik, Humboldt-Universität zu Berlin, 1993.

Mer97 Ulf v. Mersewsky: Generierung von ASDL-CR und Konvertierung nach SITE-CR. Studienarbeit, Institut für Informatik, Humboldt-Universität zu Berlin, 1997.

Mer98 Ulf v. Mersewsky: Crossreferencer - Ein Beispiel für eine CORBA-basierte Integration in die Entwicklungsumgebung SITE. Diplomarbeit, Institut für Informatik, Humboldt-Universität zu Berlin, 1998.

Neu97 Toby Neumann: Eine Laufzeitbibliothek für SDL-92 in Java. Studienarbeit, Institut für Informatik, Humboldt-Universität zu Berlin, 1997.

OFM+94 A. Olsen, O. Færgemand, B. Møller-Pedersen, R. Reed, J. R. W. Smith: Systems Engineering Using SDL-92. North-Holland 1994.

OMG97 Object Management Group: Unified Modelling Language - UML (version 1.1). OMG standard, 1997.

OMG98 Object Management Group: The Common Object Request Broker: Architecture and Specification, OMG document formal/98-02-33, 1998.

Pie00 Piefel, Michael: Ein automatisch generierter SDL-Compiler, Diplomarbeit, Institut für Informatik, Humboldt-Universität zu Berlin, 2000.

Pri01 Prinz, Andreas: Formal semantics for SDL-definition and implementation. Humboldt-Universität zu Berlin, Philosophische Fakultät II, Habilitation, 2001.

SaNe93 Michael Sample, Gerald Neufeld: Snacc 1.0: A High Performance ASN.1 to C/C++ Compiler. University of British Columbia, Vancouver, Canada, 1993.

Scha92 Andreas Schade: Eine Common Representation für SDL'92 Spezifikationen. Studienarbeit, Institut für Informatik, Humboldt-Universität zu Berlin, 1992.

Scha94 Andreas Schade: Statische Semantikanalyse für SDL'92. Diplomarbeit, Institut für Informatik, Humboldt-Universität zu Berlin, 1994.

Schr94 Ralf Schröder: SDL'92 data handling in combination with ASN.1. Master thesis, Department of Computer Science, Humboldt University Berlin, 1994.

Schr02a Ralf Schröder: SDL Integrated Tool Environment. URL, www.informatik.hu-berlin.de/SITE, Department of Computer Science (System Analysis), Humboldt University Berlin, 1997-2002.

Schr02b Ralf Schröder: SITE@vantage. Tutorial (Siemens AG internal use only), version 1.11, Humboldt University Berlin, 2002.

Schw97 Matthias Schwalbe: Algorithmen und Implementierung eines inkrementalen Scanner- und Parser-Programms. Diplomarbeit, Institut für Informatik, Humboldt-Universität zu Berlin, 1997.

Schw99 Markus Schwaiger: Werkzeugunterstützte Anwendung von SDL Patterns. Diplomarbeit, Fachbereich Informatik, Universität Kaiserslautern, 1999.

SM99 Sun Microsystems, Inc.: The Source for Java Technology. URL, java.sun.com, Sun Microsystems, Inc., 1995-1999.

Ste93 Douglas Steedman: Abstract Syntax Notation One (ASN.1) The Tutorial & Reference.Technology Appraisals, Twickenham, 1993.

Str92 Bjarne Stroustrup: Die C++-Programmiersprache. Addison Weslay, Bonn, 1992.

Tel98 Telelogic: SDT 3.2 user's guide. Technische Beschreibung, Telelogic, Schweden, 1998.

Ver95 Verilog: Geode user's Guide. Technische Beschreibung, Verilog, Frankreich, 1995.

X.200 CCITT: Data communication networks open system interconnection (OSI) model and notation, service definition. Blue Book Recommendation X.200, Melbourne, 1988.

X.208 CCITT: Data communication networks open system interconnection (OSI) Abstract Syntax Notation One. In Blue Book Recommendation X.200, Melbourne, 1988.

X.209 CCITT: Data communication networks open system interconnection (OSI) Basic Encoding Rules of Abstract Syntax Notation One. In Blue Book Recommendation X.200, Melbourne, 1988.

X.219 CCITT: Data communication networks open system interconnection (OSI) Remote operations: model, notation and service definition. In Blue Book Recommendation X.200, Melbourne, 1988.

X.680 ITU-T: Data networks and open system communications, OSI networking and system aspects - Abstact Syntax Notation One (ASN.1); Information Technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation, ITU-T Recommendation X.680, Geneva, 1998.

X.690 ITU-T: Data networks and open system communications, OSI networking and system aspects - Abstact Syntax Notation One (ASN.1); Information technology - ASN.1 Encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER), and Distinguished Encoding Rules (CER), ITU-T Recommendation X.690, Geneva, 1998.

X.691 ITU-T: Data networks and open system communications, OSI networking and system aspects - Abstact Syntax Notation One (ASN.1); Information technology - ASN.1 Encoding rules: Specification of Packed Encoding Rules (PER), ITU-T Recommendation X.690, Geneva, 1998.

X.880 ITU-T: Concepts, model and notation; Information technology - Remote Operations, ITU-T Recommendation X.880, Geneva, 1994.

X.920 ITU-T: Open distributed processing: Interface definition language, Information technology, Recommendation X.920, Geneva, 1997.

Z.100 CCITT: SDL - Specification Description Language, International Standard Recommendation Z.100, Geneva,1992.

Z100A CCITT: Annex A in SDL - Specification Description Language: Index of non-terminals and keywords, International Standard Recommendation Z.100, Geneva,1992.

Z100B CCITT: Annex B in SDL - Specification Description Language: Glossary, International Standard Recommendation Z.100, Geneva,1992.

Z.100C CCITT: Annex C in SDL - Specification Description Language: Intitial Algebra Model, International Standard Recommendation Z.100, Geneva,1992.

Z.100D CCITT: Annex D in SDL - Specification Description Language: Definition of pakkage Predefined, International Standard Recommendation Z.100, Geneva,1992.

Z.100F CCITT: Annex F in SDL - Specification Description Language: Formal Semantics, International Standard Recommendation Z.100, Geneva,1992.

Z100I CCITT: Appendix I in SDL - Specification Description Language: SDL Methodology Guidelines, International Standard Recommendation Z.100, Geneva,1992.

Z100II CCITT: Appendix II in SDL - Specification Description Language: SDL Bibliography, International Standard Recommendation Z.100, Geneva,1992.

Z100A1 ITU-T: Addendum 1 - Master list of corrections to Z.100, International Standard Recommendation Z.100, Geneva,1996.

Z.105 ITU-T: SDL in combination with ASN.1, Recommendation Z.105, Geneva, 1995.

Z.106 ITU-T: Common interchange format for SDL, Recommendation Z.106, Geneva, 1996.

Z.130 ITU-T: ITU Object Definition Language, Recommendation Z.130, Geneva,1999.

C Index

A

Ableitung 9
abstrakte Sorte 36
abstrakte Syntax 10
abstrakter Typ 9
Agent 13
Aktion 6
Aktionsverzweigung 7
algorithmische Notation 13
algorithmischer Operator 31
Anwendung eines Operators 23
ASN.1 55
Attribut 86
Attribut (ODL) 72
Attributbeschreibung 86
Auflösung von Bezeichnern 28
Aufruf einer Methode 87
Ausdruck 23
Ausführung (Prozeß) 7
Ausführung (SDL) 11
Ausnahme (ODL) 72
Ausnahmebehandlung 13
Auswertung eines Terms 25
Axiom 24
axiomatischer Operator 31

B

Basiskommunikation 18
Basistyp 9
BER 58
Berechnung eines Operators 25
Berechnung eines Terms 25
Bezeichner 8, 23
Bindung von Methoden 87
Block 8

C

CDR 73
Common Representation 15
CSP 11

D

Datentyp 9, 22
Datentyp (ODL) 72
Datentypbezeichner 22
Datentyperweiterung 57
Datentypzuweisung 56
Datenwert 22
Datenwert (ODL) 72
dynamisch typsicher 88
dynamische Semantik 10
dynamische Typbeschreibung 87

E

Einbettung 43
Einzelobjektverwaltung 93
Empfangspuffer 7
entfernte Prozedur 8
externe Sorte 38

F

Fehlerterm 36
Finalisierung 89
Funktion (ODL) 72

G

Generator 36
Gleichung 24
Grundzustand 6
Gruppe (ODL) 72

I

Informationsobjekt 58
Instanz 85
Integration 42
interface-Konzept 13

K

Kernkonzept 10
Kommunikationspfad 8
konkrete Syntax 10
konstruierter Datentyp 56

L

Laufzeittypinformation 89
Laufzeitumgebung 9
Literal 22
Literalbezeichner 22

M

MetaIV 10
Methode 86
Modul 8
Modul (ODL) 72

N

Namen (ASN.1) 57
Nebeneffekt 86
nicht typsicher 89

O

Objekt 85
Objekt (ODL) 72
Objektorientierung 2
Objektreferenz 93
Objektreferenz (ODL) 72
Objekttyp 113
Objekttyp (SDL-2000) 121
ODL 71
Operator 22
Operator (allg.) 86
Operatorbezeichner 22

P

Parametrisierung (ASN.1) 58
PDU 77
Polymorphie 87
primitiver Datentyp 56
Prozedur 7
Prozeß 7
Prozeßinstanz 7
Prozeßinstanzmenge 7

Q

qualifier 27

R

REFERENCE (Sorte) 97
Referenz 86
Referenzsorte 87
resolution by context 28
run time type information 89

S

Sammelobjektverwaltung 93
Schnittstelle (ODL) 72
SDL 6
SDL/GR 5
SDL/PR 5
SDL-Beschreibung 9
SDL-Programm 9
SDL-Systemumgebung 9
Service 8
Signal 7
Signalbeschreibung 7
Signalinstanz 7
Signatur 22
Sorte 22
Sorte einer Variablen 23
späte Bindung 87
Speicherprozeß 93
Speicherverwaltung 89
Spezifikation 9
statisch typsicher 88
statische Bindung 87
statische formale Semantik 10
statische Semantik 10
statische Sortenvariable 104
statischer Typ 87
strenges Datentypkonzept 24
Syntax 10
Syntaxzuweisung 57
syntype-Konstruktion 33
syntype-Test 34
System 8

T

Teilbereichstyp (ASN.1) 56
Term einer Sorte 23
Termgleichheit 24
Typ 9
Typdefinition 9
Typkennzeichner 56
Typsicherheit 88

U

Überladung von Namen 22
ungetypte Referenz 87
ungültige Referenz 99

V

Variable (ACT ONE) 23
Variable (SDL) 9
Vererbung 9
Verhaltensgraph 7
Virtualität (Daten) 88
Virtualität (SDL) 9
voll qualifizierter Bezeichner 28

vollständiges Transformationsmodell . 42

W

Wert (ASN.1) 56
Wert einer Sorte 24
Wertebezeichner 22
Wertesemantik 33
Wertetyp 113
Wertetyp (SDL-2000) 121
Wertzuweisung (ASN.1) 56

Z

Zugangspunkt 77
Zustandsübergang 6
Zuweisung 33
zuweisungskompatibel 24
Zuweisungsregeln 24

Erklärung

Ich erkläre hiermit, daß

- ❍ ich die vorliegende Dissertationsschrift
  „SDL-Datenkonzepte - Analyse und Verbesserungen“
  selbständig und ohne unerlaubte Hilfe angefertigt habe;
- ❍ ich mich nicht bereits anderwärtig um einen Doktorgrad beworben habe oder einen solchen besitze;
- ❍ mir die Promotionsordnung der Mathematisch-Naturwissenschaftlichen Fakultät II der Humboldt-Universität zu Berlin bekannt ist.

Tabellarischer Lebenslauf

Dipl. Inf. Ralf Schröder, geb. 21.08.1966 in Luckenwalde

1973-1983	Besuch der Polytechnischen Oberschule „Ernst Thälmann“ in Luckenwalde
1983-1985	Abitur an der Spezialklasse für Mathematik und Physik der Humboldt Universität zu Berlin
1985-1988	Dienst bei der NVA der DDR
1988-1989	Mathematikstudium an der Humboldt Universität zu Berlin
1989-1994	Fortsetzung des Studiums am neu gegründeten Institut für Informatik der Humboldt Universität zu Berlin
1993	Praktikum bei der GMD/FOKUS in Berlin, Implementierung einer Analysekomponente für ASN.1-Werte
1994	Abschluß als Diplom-Informatiker am Institut für Informatik der Humboldt Universität zu Berlin
1994-2002	Wissenschaftlicher Mitarbeiter für die Durchführung von Drittmittelprojekten (ProjeX, PLATINUM, VESUV, SITE@vantage) am Institut für Informatik der Humboldt Universität zu Berlin 1994-1999 Verantwortungsbereich: Einsatz, Wartung und Weiterentwicklung der SDL-Werkzeuge des Lehrstuhls Systemanalyse 1999-2002 Arbeit am Dissertationsthema „SDL-Datenkonzepte - Analyse und Verbesserungen“
ab 2002	Entwicklungsingenieur Software bei der DeTeWe Funkwerk Köpenick GmbH in Berlin