UNIVERSITÄT
KOBLENZ · LANDAU
Institut für Softwaretechnik

FB 4
Informatik

Ein Referenzschema für die Sprachen der IEC 61131

Tassilo Horn
Jürgen Ebert

Nr. 13/2008

Arbeitsberichte aus dem Fachbereich Informatik

Universität Koblenz-Landau • Campus Koblenz • Universitätsstraße 1 • 56070 Koblenz
Telefon +49 261 287-0 • http://www.uni-koblenz.de

Die Arbeitsberichte aus dem Fachbereich Informatik dienen der Darstellung vorläufiger Ergebnisse, die in der Regel noch für spätere Veröffentlichungen überarbeitet werden. Die Autoren sind deshalb für kritische Hinweise dankbar. Alle Rechte vorbehalten, insbesondere die der Übersetzung, des Nachdruckes, des Vortrags, der Entnahme von Abbildungen und Tabellen – auch bei nur auszugsweiser Verwertung.

The "Arbeitsberichte aus dem Fachbereich Informatik" comprise preliminary results which will usually be revised for subsequent publication. Critical comments are appreciated by the authors. All rights reserved. No part of this report may be reproduced by any means or translated.

Arbeitsberichte des Fachbereichs Informatik

ISSN (Print): 1864-0346
ISSN (Online): 1864-0850

Herausgeber / Edited by:

Der Dekan:
Prof. Dr. Zöbel

Die Professoren des Fachbereichs:
Prof. Dr. Bátori, Prof. Dr. Beckert, Prof. Dr. Burkhardt, Prof. Dr. Diller, Prof. Dr. Ebert, Prof. Dr. Furbach, Prof. Dr. Grimm, Prof. Dr. Hampe, Prof. Dr. Harbusch, Jun.-Prof. Dr. Hass, Prof. Dr. Krause, Prof. Dr. Lämmel, Prof. Dr. Lautenbach, Prof. Dr. Müller, Prof. Dr. Oppermann, Prof. Dr. Paulus, Prof. Dr. Priese, Prof. Dr. Rosendahl, Prof. Dr. Schubert, Prof. Dr. Staab, Prof. Dr. Steigner, Prof. Dr. Troitzsch, Prof. Dr. von Kortzfleisch, Prof. Dr. Walsh, Prof. Dr. Wimmer, Prof. Dr. Zöbel

Kontaktdaten der Verfasser

Tassilo Horn, Jürgen Ebert
Institut für Softwaretechnik
Fachbereich Informatik
Universität Koblenz-Landau
Universitätsstraße 1
D-56070 Koblenz
EMail: horn@uni-koblenz.de, ebert@uni-koblenz.de

Ein Referenzschema für die Sprachen der IEC 61131

Tassilo Horn, Jürgen Ebert
{horn,ebert}@uni-koblenz.de
Institut für Softwaretechnik

19. November 2008

1 Einführung

Das Ziel der Automatisierungs- bzw. Prozessleittechnik ist die maschinelle Umsetzung eines komplexen Prozesses, wie z.B. der Herstellung und Portionierung eines Kuchenteigs in einer Großbäckerei, in einem *Prozessleitsystem*.

In der Regel arbeiten solche Prozessleitsysteme *zyklusorientiert*. Jeder Zyklus beginnt mit dem Auslesen der Werte aller vorhandenen *Sensoren* (*sensor*) der Anlage, welche daraufhin in Eingabevariablen gespeichert werden. Dabei sind beispielsweise Thermometer oder Manometer mögliche Sensoren. Danach wird die Kontrolle an die vom Benutzer programmierte Steuerung übergeben. Aufgrund der zuvor eingelesenen Sensorenwerte können nun die Werte der den *Aktoren* (*actuator*) zugeordneten Ausgabevariablen geändert werden. Ventile und Pumpen sind Beispiele für Aktoren. Als letzter Schritt werden diese Werte vom Prozessleitsystem an die Aktoren übertragen, und ein neuer Zyklus beginnt.

Bei den Steuerungen unterscheidet man zwischen *verbindungsprogrammierten (VPS)* und *speicherprogrammierbaren Steuerungen (SPS, programmable logic controller, PLC)*. Bei verbindungsprogrammierten Steuerungen ist die Programmlogik fest verdrahtet, wohingegen speicherprogrammierbare Steuerungen in Software programmiert werden. Damit sind SPS deutlich flexibler und haben verbindungsprogrammierte Steuerungen zum großen Teil verdrängt.

Die IEC[1] hat für speicherprogrammierbare Steuerungen den Standard *IEC 61131: Programmable Controllers* eingeführt. Dessen dritter Teil *Programming Languages* ([IEC03b]) beschäftigt sich mit fünf verschiedenen Sprachen, die zur Programmierung von prozessleittechnischen Systemen verwendet werden können.

Die Norm definiert die beiden textuellen Sprachen *Instruction List* (*IL*, *Anweisungsliste*, *AWL*) und *Structured Text* (*ST*, *Strukturierter Text*, *ST*), die beiden visuellen Sprachen *Ladder Diagram* (*LD*, *Kontaktplan*, *KP*) und *Function Block Diagram* (*FBD*, *Funktionsbaustein-Sprache*, *FBS*) und die *Sequential Function Charts* (*SFC*, *Ablaufsprache*, *AS*), welche sowohl eine textuelle als auch eine visuelle Repräsentation haben.

[1] *International Electrotechnical Commission*: internationales Normierungsgremium für Normen im Bereich Elektronik und Elektrotechnik mit Sitz in Genf

Neben der Einteilung nach konkreter Syntax ist ebenfalls eine Einteilung nach Programmiersprachenparadigma möglich. Die beiden textuellen Sprachen IL und ST sind *kontrollflussorientiert*, die beiden visuellen Sprachen LD und FBD sind *datenflussorientiert* und SFCs sind *zustandsorientiert*.

Obwohl durch diese Norm, die zwischenzeitlich eine weite Verbreitung gefunden hat, eine Annäherung der praktisch eingesetzten Sprachen in konkreten Systemen an eben diese erkennbar ist, muss doch festgestellt werden, dass weiterhin von unterschiedlichen Herstellern und an verschiedenen Forschungseinrichtungen abweichende Dialekte und Varianten der genannten Sprachen verwendet werden. Dies hat verschiedene Ursachen. Eine davon liegt in der Historie der teilweise schon lange im Einsatz befindlichen Systeme. Zudem kommen prozessleittechnische Anlagen inklusive Programmiersystem zumeist aus einer Hand. Es muss aber auch festgestellt werden, dass die Norm selbst in der Definition ihrer Sprachen teilweise noch unpräzise ist und viele Details den Implementatoren eines Systems überlässt, was eine Angleichung ebenfalls erschwert.

Es besteht also ein dringender Bedarf, die Sprachen dem heutigen Stand der Technik entsprechend in Syntax und Semantik präzise zu definieren. Eine solche präzise Sprachbeschreibung ist insbesondere für den Bau von sprachverarbeitenden Werkzeugen (Editoren, Profilern, Analysatoren, Übersetzern, Interpretern, Simulatoren, etc.) grundsätzlich erforderlich.

Der vorliegende Bericht unternimmt den Versuch, die Syntax und Semantik der IEC-Sprachen formal zu definieren. Hierzu wird der Ansatz der Metamodellierung ([Obj07]) verfolgt, welcher es erlaubt sowohl textuelle als auch visuelle Sprachen in ihrer abstrakten Syntax präzise zu erfassen. In Metamodellen werden alle wesentlichen Sprachelemente sowohl einzeln als auch in ihrer Beziehung zueinander explizit erfasst, während vernachlässigbare Details ihrer konkreten Syntax, wie z.B. das Layout oder konkrete lexikalische Symbole, ignoriert werden.

Die Definition der Metamodelle erfolgt hier unter Verwendung von UML Klassendiagrammen in einer Form, die sicherstellt, dass jedes konkrete Programm eindeutig durch seinen abstrakten Syntaxgraphen als Instanz dieses Metamodells definiert ist ([ERW08], [ERSB08], [EWD$^+$98]). Dieser Syntaxgraph stellt dann die interne Repräsentation des Programms innerhalb der Werkzeuge dar.

Um die breite Verwendbarkeit des Metamodells zu ermöglichen, muss diese Beschreibung für konkrete, im Einsatz befindliche Varianten der IEC-Sprachen entsprechend angepasst werden können. Die hier dargestellte Sprachdefinition ist daher als Vorschlag für eine Referenzbeschreibung ([Win00]) der Sprachen aufzufassen, die einerseits die in der Norm festgelegten Sprachen präzise erfasst, die aber auch gleichzeitig verwendet werden kann, die Abweichungen konkreter Implementationen explizit darstellbar zu machen.

In den folgenden Kapiteln wird die abstrakte Syntax der Sprachen der IEC 61131-3 formal durch ein integriertes Metamodell definiert. Der Bericht beabsichtigt nicht eine tutorielle Einführung in die Sprachen zu geben. Der Verständlichkeit halber wird die Darstellung jedoch durch Beispiele, kurze sprachliche Beschreibungen der Semantik und durch weitere tabellarische Übersichten ergänzt. Dabei wird immer auf die Beschreibungen der Norm Bezug genommen.

Die Elemente des Metamodells sind in verschiedene Pakete gegliedert. Das Paket `common` (Abb. 1) und dessen Unterpakete enthalten alle Elemente, die in mehreren der Programmiersprachen der IEC 61131-3 verwendet werden. Diese gemeinsamen Elemente sind Gegenstand vom Kapitel 2.

Das Paket `configuration` enthält die in Kapitel 2.5 beschriebenen Konfigurationselemente. Zudem finden sich die Pakete zu allen fünf Programmiersprachen wieder. Auf diese wird in Kapitel 3 eingegangen.

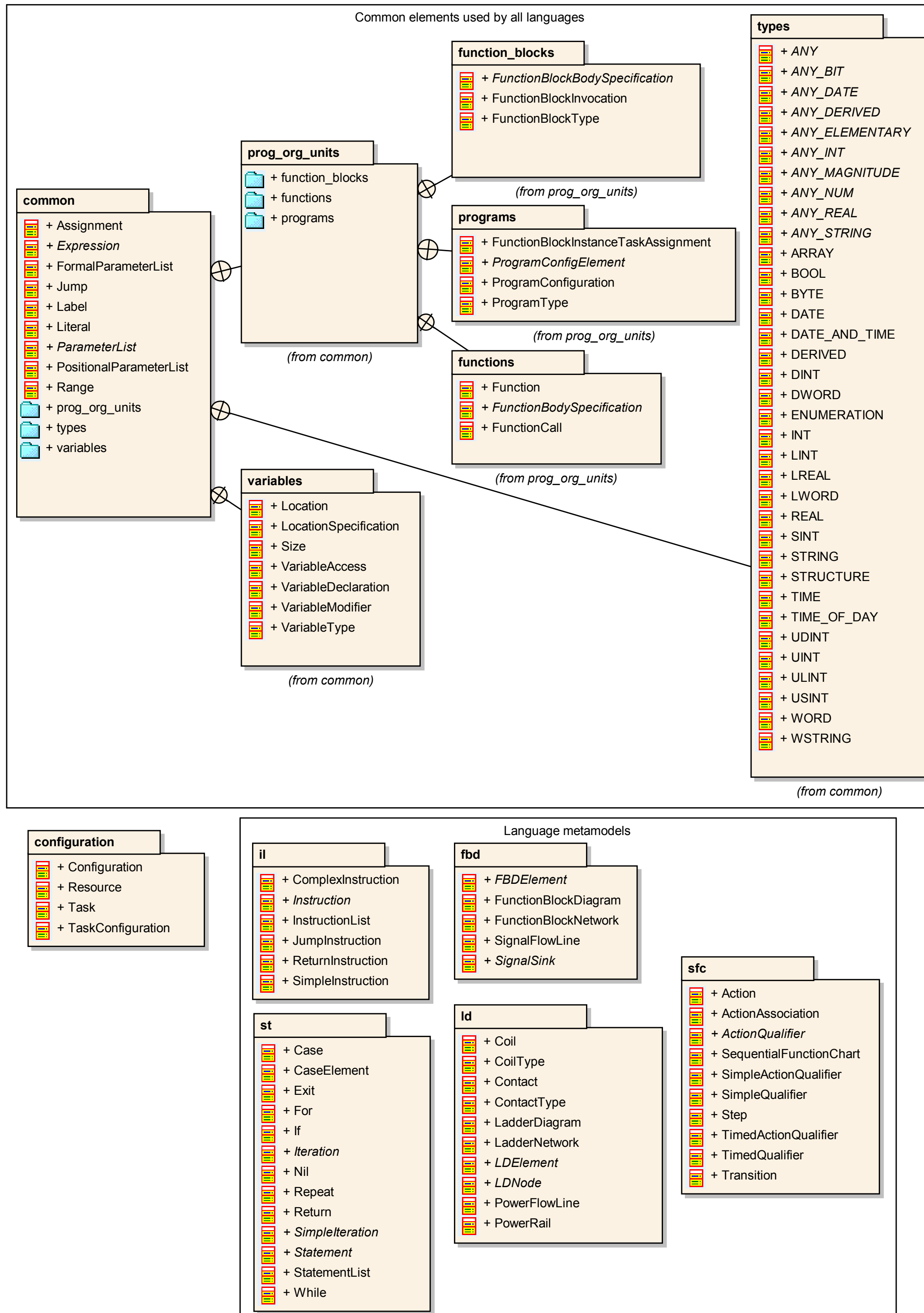

Abbildung 1: Die Paketstruktur des integrierten Metamodells

2 Gemeinsame Elemente

Die fünf Sprachen der Norm IEC 61131-3 ([IEC03b]) teilen sich einige gemeinsame Elemente, die in jeder dieser Sprachen verwendet werden. Das sind Datentypen, Literale, Variablen und die verschiedenen Programmorganisationseinheiten *Funktion* (*function*), *Funktionsbaustein* (*function block*) und *Programm* (*program*).

2.1 Datentypen

Die Norm definiert alle Datentypen, die man auch bei einer höheren Programmiersprache erwarten würde. Zusätzlich existieren vordefinierte, elementare Datentypen für Zeitdauern, Uhrzeiten und Daten.

Da die IEC 61131-3 Funktionen definiert, welche Parameter verschiedener Typen verarbeiten können (sog. überladene Funktionen), existiert eine Klassifikations-Hierarchie zwischen den elementaren Datentypen, die in Abbildung 2 angegeben ist. Die `ANY`-Typen werden auch *generische Typen* genannt.

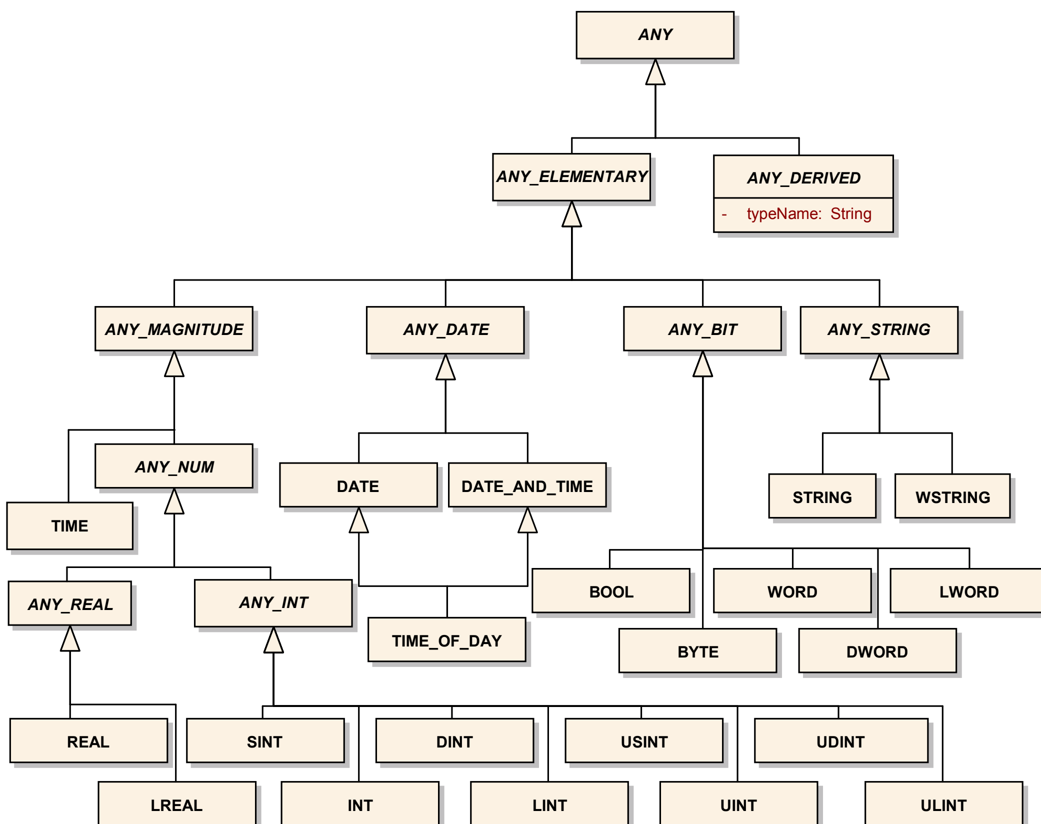

Abbildung 2: Datentyphierarchie nach IEC 61131-3

Vom Benutzer definierte Funktionen können diesen Mechanismus jedoch nicht nutzen. Hier müssen die Parameter einer Funktion oder eines Funktionsbausteins einen nicht-generischen Typ haben.

Ein Benutzer kann zudem neue Typen definieren. Dies können Felder (*array*), Aufzählungstypen (*enumerations*), abgeleitete Typen oder Strukturen (*structures*) sein.

Abbildung 3 zeigt den Ausschnitt des Metamodells, der die Definition neuer Typen beschreibt.

Abbildung 3: Metamodellausschnitt zur Definition neuer Typen

Auf Literale und Variablen wird in den beiden folgenden Abschnitten eingegangen, und Funktionsbausteintypen (`FunctionBlockType`) sind Thema des Abschnitts 2.4.2 auf Seite 14.

Jeder neue Typ hat einen eindeutigen Namen. Ein Beispiel für die Definition eines Aufzählungstyps (`ENUMERATION`) namens `VOLTAGE` mit Initialwert `5V_TO_10V` ist in Listing 1 angegeben.

```
1  TYPE
2    VOLTAGE : (1V_TO_5V, 5V_TO_10V, 10V_TO_20V, 20V_TO_100V) := 5V_TO_10V;
3  END_TYPE
```

Listing 1: Definition eines Aufzählungstyps

Ein von `INT` abgeleiteter, auf einen Wertebereich von eins bis eintausend eingeschränkter Typ `NATURAL_NUMBER` mit Initialwert eins wird in Listing 2 definiert.

```
TYPE NATURAL_NUMBER : INT (1..1000) := 1; END_TYPE
```

Listing 2: Definition eines abgeleiteten Typs

In Listing 3 wird ein zweidimensionaler Array-Typ mit jeweils zwei natürlichen Zahlen definiert. Die Initialwerte der Komponenten (`[1,1],[1,2],[2,1],[2,2]`) werden durch eckige Klammern eingeschlossen angegeben.

```
TYPE NAT_NUM_ARRAY :
  ARRAY [1..2, 1..2] OF NATURAL_NUMBER := [1, 2, 3, 4];
END_TYPE
```

Listing 3: Definition eines Array-Typs

```
TYPE SIGNAL_STRUCT :
  STRUCT
    NUMS : NAT_NUM_ARRAY;
    VOL  : VOLTAGE := 1V_TO_5V;  (* Use another default value *)
  END_STRUCT
END_TYPE
```

Listing 4: Definition eines strukturierten Typs

Zuletzt wird in Listing 4 ein neuer strukturierter Typ namens `SIGNAL_STRUCT` definiert.

In den Beispielen und im Metamodellausschnitt aus Abbildung 3 wurden schon Literale benutzt, die im folgenden Kapitel erläutert werden.

2.2 Literale

Literale haben im Allgemeinen die Form <type>#<value>. Die Ganzzahl -12 kann also geschrieben werden als `INT#-12`.

Für alle Typen, die vom generischen Typ `ANY_NUM` abgeleitet werden, darf die Typangabe aber entfallen. So sind -12, 34 und 0 alles gültige Ganzzahlliterale, -12.4 und 931.041 sind gültige Fließkommaliterale.

Für Wahrheitswerte (Typ `BOOL`) sind `FALSE`, `0`, `TRUE` und `1` gültige Literale. Dabei entspricht `0` `FALSE`, und `1` entspricht `TRUE`.

Zeichenketten vom Typ `STRING` (ein Byte pro Zeichen) werden von einfachen Hochkommata eingeschlossen, Zeichenketten vom Typ `WSTRING` (zwei Byte pro Zeichen) in doppelte Hochkommata.

Einige Literale zu den für die Programmierung von Prozessleitsystemen besonders wichtigen Zeit- und Datumstypen sind in Tabelle 1 angegeben.

Typ	Beispiel	Beispiel (Kurzschreibweise)
TIME	TIME#-3d4h31m0.5s	T#1.5m
TIME_OF_DAY	TIME_OF_DAY#12:37:51	TOD#1:2:3
DATE	DATE#2008-08-26	D#1997-1-1
DATE_AND_TIME	DATE_AND_TIME#2008-8-1-1:2:3	DT#1997-1-1-12:29:17

Tabelle 1: Literale zu Zeit- und Datumstypen

Der Typ `TIME` dient zur Darstellung von Zeitdauern. Die Angabe erfolgt mit ganzzahligen Werten für Tage (`d`), Stunden (`h`), Minuten (`m`), Sekunden (`s`) und Millisekunden (`ms`). Es ist ebenfalls möglich, einen Wert als Fließkommazahl anzugeben. In diesem Fall darf jedoch keine Angabe in kleinerer Einheit folgen. Damit sind die Literale `T#1.25d` und `T#1d6h` äquivalent.

2.3 Variablen

Variablen werden innerhalb von Blöcken beginnend mit einem Schlüsselwort und endend mit dem Schlüsselwort END_VAR deklariert. Alle möglichen Schlüsselworte sind in Tabelle 2 angegeben.

Schlüsselwort	Bedeutung
VAR_INPUT	Eingabevariable
VAR_OUTPUT	Ausgabevariable
VAR_IN_OUT	Ein-/Ausgabevariable
VAR	lokale Variable
VAR_TEMP	temporäre Variable
VAR_GLOBAL	globale Variable
VAR_EXTERNAL	externe Variable zum Zugriff auf globale Variable
VAR_CONFIG	Konfigurationsvariable (vgl. Kapitel 2.5)
VAR_ACCESS	Zugriffspfade (vgl. Kapitel 2.5)

Tabelle 2: Schlüsselwörter von Variablendeklarationsblöcken

Der Teil des Metamodells, welcher die Deklaration von und den Zugriff auf Variablen veranschaulicht, ist in Abbildung 4 angegeben.

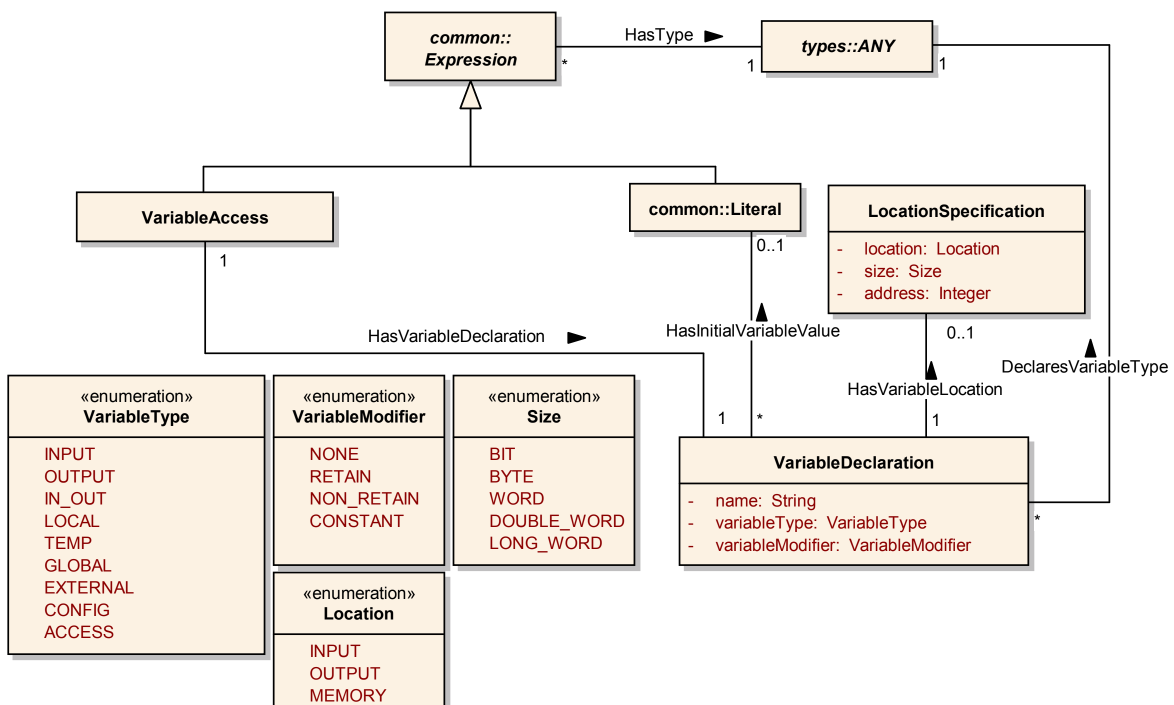

Abbildung 4: Metamodellausschnitt zu Variablen (Deklaration und Zugriff)

Innerhalb eines Deklarationsblockes können beliebig viele Variablen mittels VariableDeclarations deklariert werden. In der Regel geschieht dies durch Angabe eines Namens (Attribut name), eines Typs (ANY) und eines optionalen Initialisierungswertes (Literal). Die VariableDeclaration gibt zudem noch den Variablentyp (VariableType) und den eventuell vorhandenen Mo-

difizierer (VariableModifier) an. Der Modifizierer CONSTANT zeichnet eine nur lesbare Konstante aus, die Modifizierer RETAIN und NON_RETAIN beeinflussen das Warm- bzw. Kaltstartverhalten.

Bei einem Warmstart einer speicherprogrammierbaren Steuerung werden die mit RETAIN markierten Variablen mit den zuvor gespeicherten Werten initialisiert. Bei einem Kaltstart werden diese Variablen mit den benutzerdefinierten Initialwerten (sofern vorhanden) oder den typspezifischen Standardwerten belegt. Variablen, die mit NON_RETAIN markiert sind, werden immer mit den benutzerdefinierten Initialwerten bzw. den typspezifischen Standardwerten belegt.

Die Benutzung einer Variable wird durch einen Variablenzugriff (VariableAccess) repräsentiert.

Listing 5 zeigt einige Beispieldeklarationen lokaler Variablen.

```
1 VAR
2   my_int   : INT;
3   a_bool   : BOOL := TRUE;
4   duration : TIME := T#12m30s;
5 END_VAR
```

Listing 5: Beispiel zu Variablendeklarationen

Die Variable my_int hat den Typ INT. a_bool ist eine boolsche Variable und wird mit TRUE initialisiert. Die Variable duration enthält eine auf 12 Minuten und 30 Sekunden initialisierte Zeitdauer.

Direkt adressierte Variablen. Bei den Deklarationen aus Listing 5 werden den lokalen Variablen automatisch dem Typ entsprechende Speicherorte für ihren Wert zugewiesen. Mittels sogenannter *direkt adressierter Variablen* kann der Speicherort (LocationSpecification im Metamodell) aber auch explizit festgelegt werden. In Listing 6 sind einige Beispieldeklarationen angegeben.

```
1 VAR_GLOBAL
2   emergency_halt AT %IX3   : BOOL;
3   enable_pump    AT %Q8    : BOOL := FALSE;
4                  AT %MB7.2 : USINT;
5 END_VAR
```

Listing 6: Beispiel zu direkt adressierten Variablen

Direkte Variablen beginnen immer mit einem Prozentzeichen, gefolgt von einem Ortsindikator: I für Eingang, Q für Ausgang und M für Speicher. An diesen schließt sich ein Indikator für die Größe an. Ist dieser X oder wird er weggelassen, so wird von einem Bit ausgegangen. Die Indikatoren B, W, D und L stehen für Größen von einem Byte, einem Wort, einem Doppelwort bzw. einem LWORD. Zuletzt folgt eine Nummer, welche die Speicherstelle spezifiziert. Eine hierarchische Angabe ist im Rahmen des Standards ebenso erlaubt. Wie diese verwirklicht wird, ist jedoch dem Implementierer überlassen.

In Listing 6 wird dem dritten Input-Bit der symbolische Name emergency_halt zugewiesen, und dem achten Ausgangsbit wird der Name enable_pump und der Initialwert FALSE zugewiesen. In der dritten Deklaration wird nur der Typ USINT für die angegebene direkt adressierte Variable vereinbart. Die hierarchische Ortsangabe könnte etwa das zweite Byte im siebten Speicherwort des Hauptspeichers bedeuten.

2.4 Programmorganisationseinheiten

Zur Strukturierung der Programme einer SPS existieren die drei sogenannten Programmorganisations-einheiten (*program organization unit*)

- Funktion (*function*),
- Funktionsbaustein (*function block*) und
- Progamm (*program*),

auf die in den folgenden Abschnitten genauer eingegangen wird.

2.4.1 Funktionen

Der Ausschnitt des Metamodells, welcher die Deklaration und den Aufruf von Funktionen umfasst, ist in Abbildung 5 angegeben.

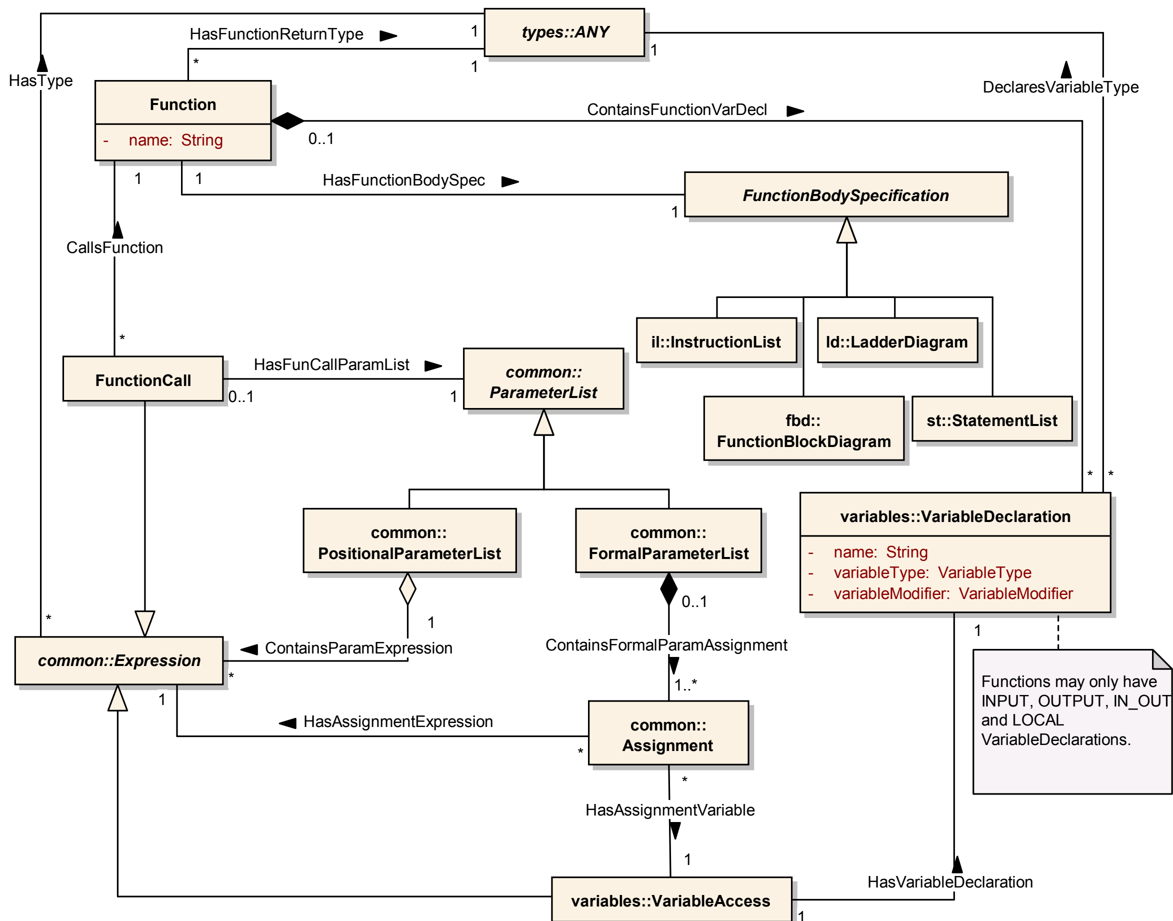

Abbildung 5: Metamodellausschnitt zu Funktionen (Deklaration und Aufruf)

Eine Funktion (`Function`) hat einen Namen (Attribut `name`) und einen Rückgabetyp (`ANY`).

Ihre Parameter werden durch VariableDeclarations angegeben. In einer Funktion können Eingabe-, Ausgabe-, E/A- und lokale Variablen deklariert werden (VariableType::INPUT, OUTPUT, IN_OUT und LOCAL).

Zudem besitzt eine Funktion einen Rumpf (FunctionBodySpecification). Dieser kann in allen Sprachen der IEC 61131-3 mit Ausnahme der Sequential Function Charts formuliert werden.

Bei einem Funktionsaufruf (FunctionCall) werden die Argumente in einer Parameterliste (ParameterList) übergeben. Hier unterscheidet der Standard zwischen positionalen (PositionalParameterList) und formalen Parameterlisten (FormalParameterList). Einige Beispiele dazu finden sich im folgenden Abschnitt.

Funktionen haben analog zu Funktionen anderer Programmiersprachen kein Gedächtnis, d.h. wird eine Funktion mehrfach mit den gleichen Eingabeparametern aufgerufen, so ist garantiert, dass die Rückgabewerte immer übereinstimmen (*referenzielle Transparenz*).

Mit dem Funktionsaufruf ist die letzte Spezialisierung eines Ausdrucks eingeführt worden. Die vollständige Expression-Hierarchie ist in Abbildung 6 angegeben.

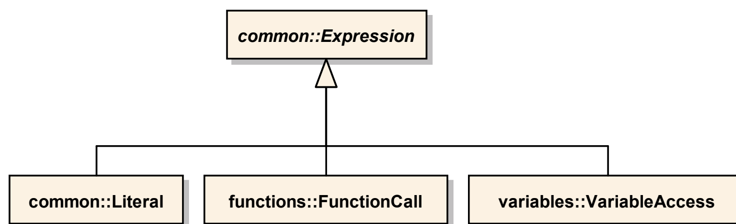

Abbildung 6: Die Expression-Hierarchie

Deklaration und Aufruf. In Listing 7 wird mit der textuellen Sprache Structured Text (siehe Kapitel 3.1) eine Funktion SAFE_DIV definiert.

```
FUNCTION SAFE_DIV : REAL
  (* External interface specification *)
  VAR_INPUT
    NUMERATOR   : REAL;
    DENOMINATOR : REAL := 1.0;
  END_VAR

  (* Function body specification *)
  SAFE_DIV := DIV(NUMERATOR, DENOMINATOR);
END_FUNCTION
```

Listing 7: Funktionsdeklaration in Structured Text

Ihr Rückgabetyp wird mit REAL angegeben.

Danach werden die beiden Eingabevariablen NUMERATOR und DENOMINATOR deklariert. Dem Nenner wird der Initialwert 1 zugewiesen, so dass eine Division durch null vermieden werden kann, falls die Funktion aufgerufen wird, ohne den zweiten Parameter zu spezifizieren.

Der Schnittstellenspezifikation schließt sich der Funktionsrumpf an. Hier werden die Eingabewerte einfach an die Standardfunktion DIV weiter delegiert.

Ein Aufruf von SAVE_DIV ist in Listing 8 angegeben.

```
FRAC := SAFE_DIV(7, 1+2);
```

Listing 8: Funktionsaufruf mit positionaler Parameterliste

Der Funktion SAVE_DIV werden im obigen Beispiel die beiden Parameter positional übergeben. In der Deklaration von SAVE_DIV wurde der Zähler vor dem Nenner deklariert, und da im Funktionsaufruf die 7 als erstes übergeben wird, findet sie als Zähler (NUMERATOR) Verwendung. Diese Art der Parameterübergabe ist auch in den meisten anderen Programmiersprachen gebräuchlich.

Zudem erlaubt die IEC-Norm aber eine sogenannte Parameterübergabe mittels *formaler Parameterlisten*. Ein zu Listing 8 äquivalenter Beispielaufruf ist in Listing 9 angegeben.

```
FRAC := SAFE_DIV(NUMERATOR := 7, DENOMINATOR := 1+2);
```

Listing 9: Funktionsaufruf mit formaler Parameterliste

Hierbei wird den VAR_INPUT- und VAR_IN_OUT-Variablen der externen Schnittstelle direkt mit dem Zuweisungsoperator := ein Wert zugewiesen. Mittels des Operators => können auch eventuell vorhandene Ausgabeparameter direkt an Variablen, die am Aufrufpunkt sichtbar sind, gebunden werden.

Angenommen, die Funktion SAFE_DIV hätte noch eine boolsche Ausgabevariable ZERO_DENOM, die genau dann den Wert TRUE annehmen würde, wenn SAVE_DIV mit einem Nennerwert null aufgerufen wird, so könnte ein Aufruf die in Listing 10 angegebene Form haben.

```
FRAC := SAFE_DIV(NUMERATOR := X, ZERO_DENOM => DIV_BY_ZERO, DENOMINATOR := Y);
```

Listing 10: Funktionsaufruf mit zusätzlichem Ausgabeparameter

Wie das Beispiel verdeutlicht, kann bei Funktionsaufrufen mit formalen Parameterlisten die Reihenfolge der Parameter beliebig sein.

Spezialparameter zur Ausführungskontrolle. Optional kann ein Hersteller eines SPS-Systems den Standardfunktionen einen boolschen Eingabeparameter EN (für *enable*) mit Initialwert TRUE und einen boolschen Ausgabeparameter ENO (für *enable output*) hinzufügen.

Wird eine Funktion aufgerufen und EN ist FALSE, so wird der Funktionsrumpf nicht ausgeführt und ENO wird vom System auf FALSE gesetzt. Dadurch ist eine einfache Möglichkeit zur Deaktivierung von Funktionen gegeben.

Ist EN beim Aufruf TRUE, so wird die Funktion ausgeführt und ENO mit TRUE initialisiert. Innerhalb des Funktionsrumpfs kann der Wert von ENO verändert und beispielsweise zum Signalisieren von Fehlern verwendet werden.

Standardfunktionen. Die Norm IEC 61131-3 definiert eine große Anzahl von Standardfunktionen für verschiedene Anwendungsgebiete. Einige Beispiele sind:

- Typkonversion: `INT_TO_REAL`, `TRUNC`, `WORD_BCD_TO_INT`
- Arithmetische und numerische Funktionen: `ADD`, `MUL`, `EXPT`, `DIV`, `ABS`, `SQRT`, `LOG`, `SIN`
- Boolsche und Bitstring-Funktionen: `AND`, `OR`, `XOR`, `NOT`, `SHL`, `SHR`, `ROL`, `ROR`
- Selektions- und Vergleichsfunktionen: `SEL`, `MIN`, `MAX`, `LIMIT`, `MUX`, `GT`, `GE`, `EQ`, `NE`
- Zeichenketten-Funktionen: `LEN`, `CONCAT`, `INSERT`, `FIND`, `REPLACE`
- Funktionen zu Zeittypen: `ADD_TIME`, `SUB_TIME`, `MULTIME`, `DIVTIME`

2.4.2 Funktionsbausteine

Function Blocks (*Funktionsbausteine*) unterscheiden sich von Funktionen dadurch, dass sie keinen besonders ausgewiesenen Rückgabewert besitzen, sondern ausschließlich über Eingabe-, Ausgabe- und Ein-/Ausgabeparameter ihre berechneten Werte weitergeben.

Der Ausschnitt des Metamodells, welches ihre Deklaration, Instanziierung und Aufruf umfasst, ist in Abbildung 7 angegeben.

Abbildung 7: Metamodellausschnitt zu Funktionsbausteinen (Deklaration, Instanziierung und Aufruf)

Die Deklaration eines Funktionsbausteintyps (`FunctionBlockType`) besteht analog zu Funktionen aus einem Variablendeklarationsteil und einem Rumpf. Zuzüglich zu den schon von Funktionen bekannten Variablentypen können Funktionsbausteine temporäre und externe Variablen enthalten. Temporäre Variablen werden bei jedem Aufruf auf ihren Initialwert gesetzt, und externe Variablen dienen zum Zugriff auf globale Variablen.

Die Funktionalität eines Bausteintyps wird in seinem Rumpf (`FunctionBlockBodySpecification`) spezifiziert. Dieser kann mit allen Sprachen der IEC 61131 definiert werden.

Um einen Funktionsbaustein zu verwenden, muss zuerst eine Instanz des gewünschten Typs erzeugt werden. Dies erfolgt mittels einer Variablendeklaration, die eine Variable des Bausteintyps einführt. Danach kann diese Instanz analog zu Funktionen mit einer Parameterliste (`ParameterList`) aufgerufen werden (`FunctionBlockInvocation`).

Jede Funktionsbaustein-Instanz verwaltet die Werte ihrer internen Variablen und Datenstrukturen und hat somit ein Gedächtnis. Daher resultiert der mehrfache Aufruf eines Funktionsbausteins in der Regel in verschiedenen Ausgabewerten.

Deklaration, Instanziierung und Aufruf. Die Definition von Funktionsbausteintypen unterscheidet sich kaum von der von Funktionen. Ein Beispiel (als strukturierter Text) ist Listing 11 zu entnehmen. Es handelt sich um eine Beispielimplementation des Standard-Funktionsbausteins `CTU` (für *count up*, Vorwärtszähler).

```
1  FUNCTION_BLOCK CTU
2   (* external interface *)
3     VAR_INPUT CU : BOOL; R  : BOOL; PV : INT; END_VAR
4     VAR_OUTPUT Q : BOOL; CV : INT; END_VAR
5   (* internally used variables and constants *)
6     CONSTANT PVmax : INT := 1024; END_VAR
7     VAR CU_TRIG : R_TRIG; END_VAR
8   (* function block body specification *)
9     CU_TRIG(CU);
10    IF R THEN CV := 0;
11    ELSIF CU_TRIG.Q AND (CV < PVmax) THEN CV := CV + 1;
12    END_IF;
13    Q := (CV >= PV);
14 END_FUNCTION_BLOCK
```

Listing 11: Definition eines Funktionsbausteintyps

Drei Eingabevariablen werden deklariert. `CU` (*count up*) ist ein boolscher Eingang, für den in Zeile 7 ein Flankendetektor (`R_TRIG`) instanziiert wird.

Ist beim Aufruf der `R`-Eingang (*reset*) auf `TRUE` gesetzt, so wird der Zähler `CV` (*current value*) auf 0 zurückgesetzt. Ansonsten wird bei jeder steigenden Flanke auf dem Eingang `CU` (dann liefert der `Q`-Ausgang der `R_TRIG`-Instanz `CU_TRIG` den Wert `TRUE`) der aktuelle Wert `CV` des Zählers inkrementiert, es sei denn, der implementationsabhängige Maximalwert `PVmax` wurde erreicht.

Der Wert am Eingang `PV` gibt an, ab welchem Wert der `Q`-Ausgang des Zählers `TRUE` anzeigen soll.

Ein Beispiel für die Instanziierung eines Bausteintyps und den Aufruf einer Instanz gibt Listing 12.

```
1  VAR my_counter : CTU; out : BOOL; val : INT; END_VAR
2  my_counter(CU := %IX7, R := %IX1, PV := 128, Q => out, CV => val);
```

Listing 12: Instanziierung und Aufruf eines Funktionsbausteins

Die Instanziierung erfolgt bei der Variablendeklaration, und beim Aufruf erfolgt die Parameterübergabe wie bei Funktionen wahlweise mittels formaler oder positionaler Parameterlisten (vgl. Abschnitt 2.4.1). Die im Beispiel gezeigte formale Parameterliste hat jedoch den Vorteil, dass die Werte der beiden Ausgabevariablen `Q` und `CV` direkt den Variablen `out` und `val` zugewiesen werden können. Alternativ kann nach dem Aufruf mittels `my_counter.Q` und `my_counter.CV` auf die Ausgänge zugegriffen werden.

Spezialparameter zur Ausführungskontrolle. Mittels der von Herstellern optional implementierbaren Spezialparametern `EN` und `ENO` kann analog zu Funktionen (vgl. Abschnitt 2.4.1) die Ausführung kontrolliert werden.

Wann immer der `ENO`-Ausgang zu `FALSE` evaluiert, muss zusätzlich garantiert werden, dass sich die Werte an den Ausgängen des Funktionsbausteins nicht von der vorherigen Ausführung unterscheiden.

Standardfunktionsbausteintypen. Genau wie bei Funktionen definiert die Norm IEC 61131-3 eine große Anzahl von Bausteintypen, welche von einem Hersteller eines SPS-Systems bereit gestellt werden müssen, um als standardkonform zu gelten.

Einige Typen inklusive einer Kategorisierung sind beispielsweise:

- Flipflops: `SR`, `RS`
- Flankendetektoren: `R_TRIG`, `F_TRIG`
- Zähler: `CTU`, `CTD`, `CTUD`
- Timer: `TON`, `TOF`, `TP`

Bei allen aufgeführten Kategorien ist sofort erkennbar, dass eine Speicherung des aktuellen Status unumgänglich für die Implementation ist und diese daher nicht allein mit Funktionen (ohne zusätzliche globale Variablen) erfolgen kann.

2.4.3 Programme

Die IEC 61131-1 ([IEC03a]) definiert Programme wie folgt:

> A *program* is a logical assembly of all the programming language elements and constructs necessary for the intended signal processing required for the control of a machine or process by a programmable controller system.

Programme ähneln sehr stark Funktionsbausteinen. Folgende Eigenschaften unterscheiden sie jedoch:

- Programmdefinitionen werden von den Schlüsselworten `PROGRAM` und `END_PROGRAM` eingeschlossen.

- In einem Programm können `VAR_ACCESS`-Variablen deklariert werden, auf die mittels eines Kommunikationsservices zugegriffen werden kann. Diese Services werden in IEC 61131-5 ([IEC00]) definiert.
- In Programmen können globale Variablen deklariert werden.
- Programme können ausschließlich in Ressourcen (vgl. Abschnitt 2.5) instanziiert werden, während Funktionsbausteine nur innerhalb von Programmen und anderen Funktionsbausteinen instanziiert werden dürfen.
- Direkt dargestellte Variablen können nur in Programmen und Konfigurationen (vgl. Abschnitt 2.5) verwendet werden, nicht aber in Funktionsbausteinen.

In Abbildung 8 ist der Ausschnitt des Metamodells angegeben, der die Deklaration und Verwendung von Programmen beschreibt.

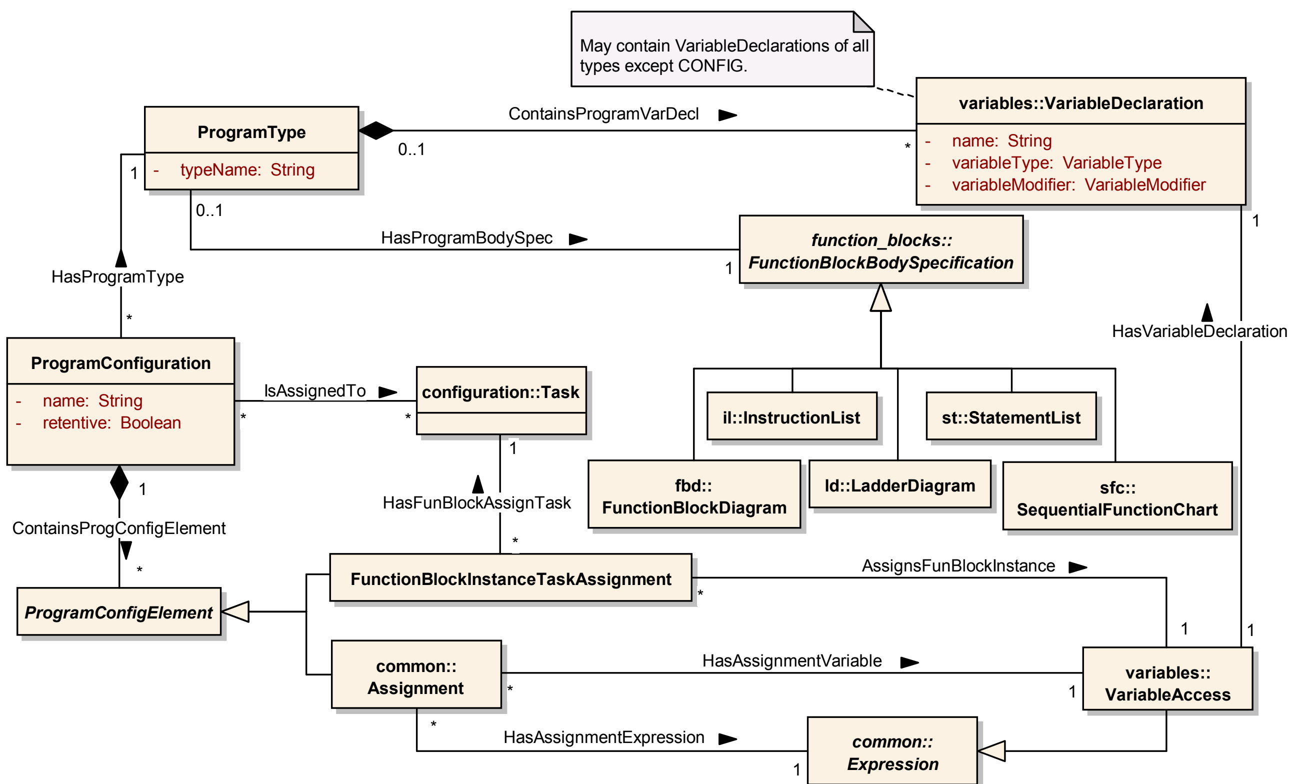

Abbildung 8: Metamodellausschnitt zu Programmen (Deklaration und Verwendung)

Der obere Teil des Diagramms weist starke Ähnlichkeit mit dem Metamodellausschnitt zu Funktionsbausteinen (Abb. 7) auf. Der einzige Unterschied ist, dass in einer Programmtyp-Deklaration (`ProgramType`) auch globale Variablen und Zugriffspfade definiert werden können.

Die Instanziierung und Ausführung eines Programms unterscheidet sich jedoch sehr stark von Funktionsbausteinen. Sie findet nämlich nicht auf der Ebene der Programmierung sondern auf Konfigurationsebene statt, welche Thema des nächsten Abschnitts ist.

Zum Verständnis des Metamodells müssen *Tasks* (`Task`) schon hier eingeführt werden. Eine Task ist ein Konfigurationselement, welches die Ausführung von ihr zugewiesenen Programmkonfigurationen

(`ProgramConfiguration`) anstößt. Dabei ist eine Programmkonfiguration eine Instanz des angegebenen Programmtyps. Zusätzlich können Tasks priorisiert werden, und es kann festgelegt werden, ob eine Task zyklisch oder nur einmal die zugewiesenen Programmkonfigurationen zur Ausführung bringt.

Mit einer Programmkonfigurationen wird ein Programmtyp instanziiert und es können Variablen dieser Instanz initialisiert werden. Dazu wird wie bei formalen Parameterlisten ein `Assignment` benutzt. Weiterhin können einzelne Funktionsbausteininstanzen der Programmkonfiguration verschiedenen Tasks zugewiesen werden.

Ein Beispiel dazu ist im folgenden Abschnitt in Listing 13 aufgeführt.

2.5 Konfigurationselemente

In diesem Kapitel soll kurz auf den Aufbau speicherprogrammierbarer Steuerungen eingegangen werden. Dieser ist in Abbildung 9 angegeben.

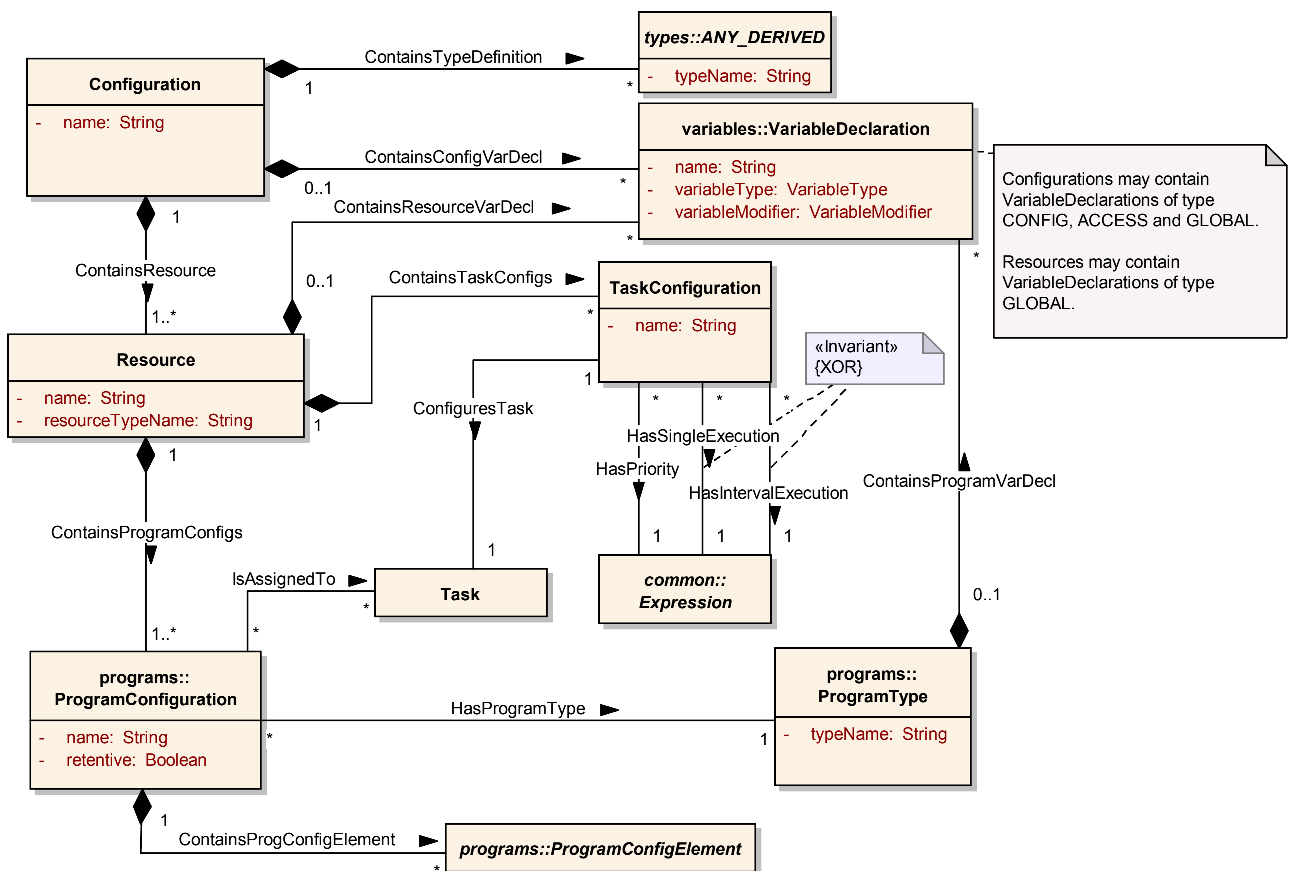

Abbildung 9: Metamodellausschnitt zu Konfigurationen

Die oberste Konfigurationseinheit bei SPS-Systemen bildet die *Konfiguration* (`Configuration`). Man kann jene als Gesamtheit des Systems inklusive aller CPUs auffassen. In der Konfiguration können globale Deklarationen und Typdefinitionen vorgenommen werden. Zudem können Zugriffspfade

angelegt werden, mit denen Kommunikation zwischen mehreren Konfigurationen ermöglicht wird. Diese Kommunikationsservices sind Gegenstand der Norm IEC 61131-5 ([IEC00]).

Die Untereinheiten einer Konfiguration sind *Ressourcen* (`Resource`). In der Regel existiert pro CPU eine Ressource.

Innerhalb einer Ressource sind wiederum *Tasks* (`Task`) mittels `TaskConfigurations` definiert. Tasks bestimmen den Zeitpunkt, die Priorität und die Art der Ausführung. So könnte es beispielsweise eine Task geben, die alle 100 Millisekunden mit höchster Priorität ausgeführt wird. In einer sicherheitskritischen Anwendung kann nun eine Instanz des Programmtyps `CHECK_TEMP` mittels einer `ProgramConfiguration`, dieser Task hinzugefügt werden.

Wird eine Programminstanz keiner Task zugewiesen, so wird sie zyklisch im Hintergrund mit niedrigster Priorität ausgeführt.

In Listing 13 ist ein Beispiel zu einer Konfiguration angegeben.

```
1   CONFIGURATION CELL_1
2     VAR_GLOBAL w: UINT; END_VAR                           (* globale Variablen *)
3
4     RESOURCE STATION_1 ON PROCESSOR_TYPE_1                        (* Ressource *)
5       VAR_GLOBAL z1: BYTE; END_VAR        (* globale Variablen der Ressource *)
6       TASK SLOW_1(INTERVAL := t#20ms, PRIORITY := 2);(* Task-Konfigurationen *)
7       TASK FAST_1(INTERVAL := t#10ms, PRIORITY := 1);
8       PROGRAM P1 WITH SLOW_1 : F(x1 := %IX1.1);  (* Programm-Konfigurationen *)
9       PROGRAM P2 : G(OUT1 => w, FB1 WITH SLOW_1, FB2 WITH FAST_1);
10    END_RESOURCE
11
12    VAR_CONFIG                                       (* Konfigurationsvariablen *)
13      STATION_1.P1.COUNT : INT := 1;
14    END_VAR
15  END_CONFIGURATION
```

Listing 13: Beispiel einer Konfiguration

Es wird eine Konfiguration `CELL_1` deklariert. In ihr ist eine globale Variable `w` und die Ressource `STATION_1` deklariert.

Innerhalb `STATION_1` werden zwei zyklische Tasks konfiguriert. Dabei hat die Ausführung von `FAST_1` Priorität vor der Ausführung von `SLOW_1`. Weiterhin werden zwei Programme konfiguriert. `P1` hat den Programmtyp `F` und wird der Task `SLOW_1` zugewiesen. `P2` hat den Programmtyp `G` und läuft mit niedrigster Priorität zyklisch im Hintergrund, weil es keiner Task zugewiesen wurde. Die Funktionsbausteininstanzen `FB1` und `FB2`, die innerhalb des Programmtyps `G` deklariert sind, werden den beiden Tasks `SLOW_1` und `FAST_1` zugewiesen.

3 Programmiersprachen

In den folgenden Abschnitten werden die fünf Sprachen der IEC 61131-3 vorgestellt. Dabei wird zunächst auf die textuellen, kontrollflussorientierten Sprachen *Structured Text* und *Instruction Lists* eingegangen. Darauf folgen die visuellen, datenflussorientierten *Function Block Diagrams* und *Ladder Diagrams*. Zuletzt werden die sowohl textuell als auch visuell darstellbaren, zustandsorientierten *Sequential Function Charts* eingeführt.

3.1 Structured Text

Structured Text (*Strukturierter Text*, ST) ist eine höhere Programmiersprache und syntaktisch etwas an Pascal angelehnt. Der entsprechende Metamodellausschnitt ist in Abbildung 10 angegeben.

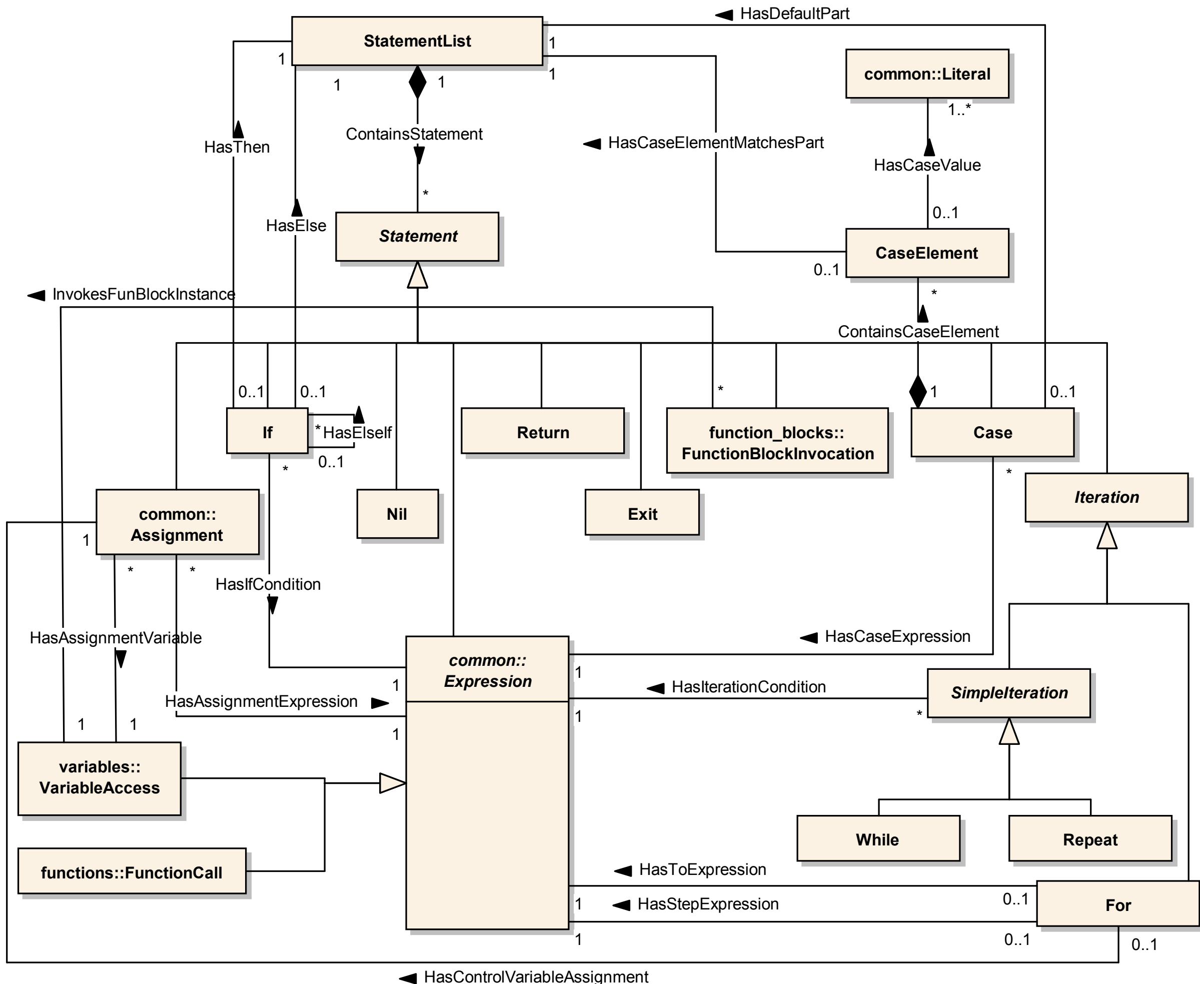

Abbildung 10: Metamodellausschnitt zu Structured Text

Ein Programm in strukturiertem Text besteht aus Anweisungen (`Statement`), die auf Ausdrücken (`Expression`) aufbauen. Ein Ausdruck liefert bei der Auswertung immer genau einen Wert eines bestimmten Datentyps.

Für alle Basisfunktionen wie `ADD`, `SUB`, `AND` oder `EQ` existieren Infixoperatoren, die alternativ zu Funktionsaufrufen benutzt werden können. Die Operatoren haben unterschiedliche Prioritäten, und durch Klammerung kann die Auswertungsreihenfolge explizit gemacht werden. Im Metamodell sind diese Operatoren aus Gründen der Übersichtlichkeit ausgespart.

Funktionsaufrufe erfolgen wahlweise mit positionaler Parameterliste (vgl. Listing 7 auf Seite 12) oder mit formaler Parameterliste (vgl. Listing 9 auf Seite 13).

Folgende Arten von Anweisungen (`Statement`) lassen sich unterscheiden:

- Zuweisungen (`Assignment`)
- Selektion (`If` und `Case`)
- leere Anweisung (`Nil`)
- Rücksprung (`Return`)
- Abbruch der innersten Schleife (`Exit`)
- Funktionsbausteinaufrufe (`FunctionBlockInvocation`)
- Schleifen (`For`, `While` und `Repeat`)

Ausführungssemantik. Strukturierter Text ist eine typische kontrollflussorientierte Sprache. Im Gegensatz zu Instruction Lists, Function Block Diagrams und Ladder Diagrams ist sie strukturiert, d.h. sie beschränkt sich auf die drei einfachen Kontrollstrukturen Sequenz (Hintereinanderausführung von Anweisungen), Selektion und Iteration und enthält keine Sprunganweisung. Damit ist sie insbesondere für die Implementierung der algorithmischen Teile einer speicherprogrammierbarer Steuerung geeignet.

Ein Beispiel zu Structured Text. Ein einfaches Beispiel, welches einige Sprachelemente illustriert, ist in Listing 14 angegeben.

```
1   SUM := 0;                      (* Assignment *)
2   FOR I := 1 TO 3 DO             (* For-Iteration *)
3     J := 0;
4     WHILE J <= 2 DO              (* While-Iteration *)
5       J := J + 1;                (* Assignment and function call via operator *)
6       IF FLAG THEN EXIT; END_IF  (* If-Selection *)
7       SUM := ADD(SUM, J);        (* Assignment and function call *)
8     END_WHILE
9     SUM := SUM + I;
10  END_FOR
```

Listing 14: Ein einfaches Programm in Structured Text

Gilt `FLAG = TRUE`, so hat die Variable `SUM` nach der Ausführung den Wert 6. Ansonsten ist ihr Wert 15.

3.2 Instruction Lists

Instruction Lists (IL) werden im deutschen Sprachraum auch *Anweisungslisten (AWL)* genannt und sind Assemblerprogrammen nachempfunden. In Abbildung 11 ist der entsprechende Ausschnitt des Metamodells angegeben.

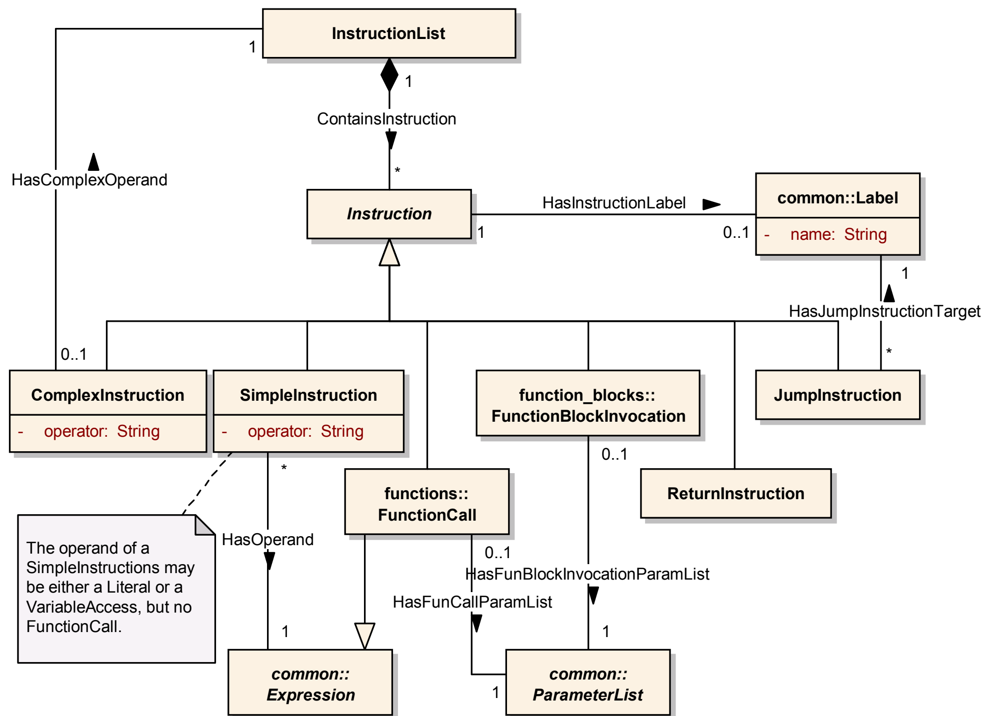

Abbildung 11: Metamodellausschnitt zu Instruction Lists

Eine Anweisungsliste (`InstructionList`) besteht aus einer Folge von Anweisungen (`Instruction`). Eine Anweisung kann mit einem `Label` markiert sein.

Bei den Anweisungen lassen sich sechs verschiedene Typen unterscheiden. `SimpleInstructions` haben einen Operator und genau einen Operanden. Dieser kann eine (direkte) Variable oder ein Literal sein. Alle definierten Operatoren sind in Tabelle 3 angegeben. Zudem können zu einigen Operatoren Modifizierer angegeben werden (Tabelle 4).

Operationen mit Klammer-Modifizierer werden im Metamodell als `ComplexInstructions` bezeichnet. Sie ähneln `SimpleInstructions`, jedoch ist ihr Operand eine in Klammern eingeschlossene Anweisungsliste.

Zudem sind Funktionsaufrufe und Aufrufe von Funktionsbaustein-Instanzen Anweisungen im Sinne von Instruction Lists.

Mit einer `JumpInstruction` kann zu einem `Label` gesprungen werden.

Zuletzt existiert eine `ReturnInstruction`, welche Verwendung finden kann, wenn ein Funktions- oder Funktionsbaustein-Rumpf mit einer Anweisungsliste spezifiziert wird.

Operator	Modifizierer	Operandentyp	Erläuterung
`LD`	`N`		Operand wird in Akkumulator geladen
`ST`	`N`		Akkumulatorwert wird im Operand gespeichert
`S`		`BOOL`	Operand wird auf `TRUE` gesetzt
`T`		`BOOL`	Operand wird auf `FALSE` gesetzt
`AND`	`N, (`	`BOOL`	
`OR`	`N, (`	`BOOL`	
`XOR`	`N, (`	`BOOL`	
`ADD`	`(`		Addition
`SUB`	`(`		Subtraktion
`MUL`	`(`		Multiplikation
`DIV`	`(`		Division
`GT`	`(`		$>$
`GE`	`(`		$\geq$
`EQ`	`(`		$=$
`NE`	`(`		$\neq$
`LE`	`(`		$\leq$
`LT`	`(`		$<$

Tabelle 3: Operatoren von Instruction Lists

Modifizierer	Bedeutung
`N`	Vor Operationsausführung wird der Operand negiert
`C`	Bedingte Ausführung (nur wenn aktueller Akkumulatorwert = `TRUE`)
`(`	Schachtelung durch Klammerung komplexer Operanden

Tabelle 4: Modifizierer für Operationen

Ausführungssemantik. Wie bei Einregistermaschinen existiert bei der Auswertung eines AWL-Programms ein spezielles Arbeitsregister: der *Akkumulator*. Eine Operation verknüpft den aktuellen Akkumulatorwert mit dem Operanden, und das Operationsergebnis wird zum neuen Akkumulatorwert. Listing 15 gibt ein einfaches Beispiel für eine `SimpleInstruction`.

```
1  LD  10
2  ADD 7
```

Listing 15: Ein Beispiel zu einfachen Anweisungen

Zuerst wird der Akku auf den Wert 10 gesetzt. Danach wird 7 hinzu addiert, und der Akkumulator enthält den Wert 17.

In Listing 16 ist ein Beispiel mit einer `ComplexInstruction` angegeben.

Wieder wird der Akku zunächst mit dem Wert 10 initialisiert. Danach wird eine geschachtelte, komplexe Anweisung ausgeführt. Dazu wird zunächst der aktuelle Akkuwert 10 auf einen Stack geschoben, um ihn später wieder zurückladen zu können. Dann wird der nun frei gewordene Akkumulator mit dem Wert 20 initialisiert und danach drei abgezogen. Damit ist der aktuelle Akkuwert 17. Durch die schließende Klammer, die das Ende der geschachtelten Anweisung anzeigt, wird der oberste Akku-

```
LD  10
ADD (
  LD 20
  SUB 3
)
```

Listing 16: Ein Beispiel zu geschachtelten Anweisungen

wert des Stacks wieder zum aktuellen Akkuwert, und der in der geschachtelten Anweisung berechnete Wert wird zum Operanden. Damit hat der Akkumulator nach Ausführung des gesamten Programms den Wert 27.

Funktionsaufrufe. Funktionen können in Anweisungslisten aufgerufen werden, indem ihr Name als Operator verwendet wird. Listing 17 zeigt einen Aufruf der Minimumsfunktion.

```
LD  10  (* Akku auf 10 setzen *)
MIN 7   (* MIN(Akkuwert, 7) ==> 7 ist neuer Akkuwert *)
```

Listing 17: Ein einfacher Funktionsaufruf

Der aktuelle Akkumulatorwert 10 wird zum ersten Funktionsparameter, alle weiteren Parameter werden durch Kommata getrennt als Operanden aufgeführt. Nach der Ausführung des Aufrufes wird der Akkumulatorwert mit dem Funktionsergebnis überschrieben.

Eine Funktion kann ebenfalls mit einer formalen Parameterliste aufgerufen werden. Damit ist der Aufruf in Listing 18 äquivalent zu jenem in Listing 17.

```
MIN(IN1 := 10, IN2 := 7)
```

Listing 18: Ein Funktionsaufruf mit formaler Parameterliste

Wiederum wird der aktuelle Akkumulatorwert mit dem Ergebnis des Funktionsaufrufs überschrieben.

Aufrufe von Funktionsbausteininstanzen. Eine Instanz eines Funktionsbausteintyps kann mit dem `CAL`-Operator aufgerufen werden. Für die Parameterübergabe existieren vier verschiedene Alternativen, die in den folgenden Listings beispielhaft vorgestellt werden. Es sei eine Instanz vom Funktionsbausteintyp `CTU` (Vorwärtszähler) namens `CNT1` deklariert, und diese soll aufgerufen werden. Wie in Listing 11 auf Seite 15 ersichtlich, besitzen Bausteine des Typs `CTU` die Eingänge `CU`, `R` und `PV` und die Ausgänge `Q` und `CV` in dieser Reihenfolge.

Im Folgenden wird jedes mal die Instanz `CNT1` aufgerufen, `%IX3` wird als Input für den Inkrement-Eingang `CU` gewählt, der Reset-Eingang `R` wird mit `FALSE` belegt und der Grenzwert, ab dem der Ausgang `Q` `TRUE` liefert, wird auf 10 gesetzt. Der Wert des Ausgangs `Q` wird in der Variablen `out` gespeichert.

Listing 19 zeigt einen Aufruf mit positionaler Parameterliste. Auf den Ausgang `Q` kann mittels `CNT1.Q` zugegriffen werden.

```
CAL CNT1(%IX3, FALSE, 10)
LD  CNT1.Q
ST  out
```

Listing 19: Aufruf mit positionaler Parameterliste

Eine weitere Möglichkeit ist der Aufruf mit formaler Parameterliste, der in Listing 20 angegeben ist.

```
CAL CNT1(CU := %IX3, R := FALSE, PV := 10, Q => out)
```

Listing 20: Aufruf mit formaler Parameterliste

Die Eingänge des Funktionsbausteins können alternativ vor dem Aufruf mit `LD`- und `ST`-Operationen initialisiert werden.

```
LD  FALSE
ST  CNT1.R
LD  10
ST  CNT1.PV
LD  %IX3
ST  CNT1.CU
CAL CNT1
LD  CNT.Q
ST  OUT
```

Listing 21: Aufruf nach Setzen der Parameter mit `LD` und `ST`

Eine vierte Variante des Aufrufs ist speziell für den Fall gedacht, dass sich seit dem letzten Aufruf nur ein Eingang verändert hat und alle weiteren Eingänge den Wert des vorherigen Aufrufs beibehalten. Bei dieser Form wird der Name des Eingangs als Operator verwendet, die Funktionsbaustein-Instanz als Operand. Ein Beispiel ist in Listing 22 gegeben.

```
LD  %IX3
CU  CNT1
LD  CNT.Q
ST  OUT
```

Listing 22: Aufruf mit Eingangsoperatoren

Direkt nach dem Setzen des `CU`-Eingangs von `CNT1` erfolgt implizit der Aufruf. Es sei angemerkt, dass diese Form des Aufrufs nur für die am häufigsten verwendeten Standard-Funktionsbausteintypen anwendbar ist.

3.3 Function Block Diagrams

Die *Function Block Diagram* Sprache (*FBD)* wird im deutschen Sprachraum *Funktionsbaustein-Sprache (FBS)* genannt.

Der sie betreffende Metamodellausschnitt ist in Abbildung 12 angegeben.

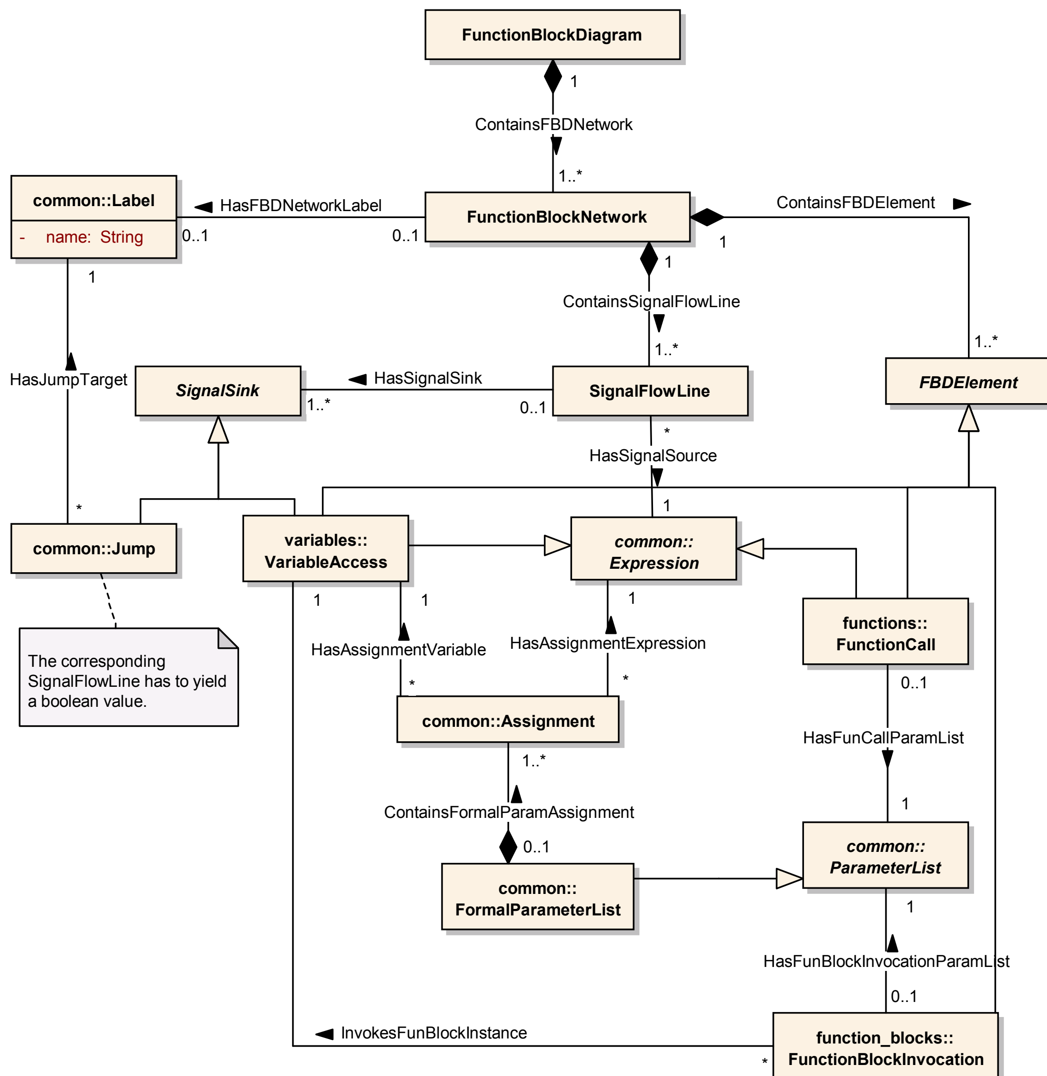

Abbildung 12: Metamodellausschnitt zur Funktionsbaustein-Sprache

Ein Funktionsbaustein-Diagramm (`FunctionBlockDiagram`) besteht aus Netzwerken (`FunctionBlockNetwork`). Ein Netzwerk kann optional mit einem Label versehen werden.

Ein Funktionsbaustein-Netzwerk besteht aus Elementen (`FBDElement`), nämlich aus Variablenzugriffen (`VariableAccess`), Funktionsaufrufen (`FunctionCall`) und Funktionsbausteinaufrufen (`FunctionBlockInvocation`).

Signalflusslinien (`SignalFlowLine`) übertragen den Wert eines Quellelements an alle Zielelemente. Valide Quellelemente sind alle Ausdrücke (`Expression`) also Variablen, Literale und Funktionsaufrufe.

Ziel einer Signalflusslinie (`SignalSink`) kann wiederum eine Variable sein oder ein Sprungbefehl (`Jump`). In letzterem Fall muss die Signalflusslinie allerdings einen boolschen Wert führen, und der Sprung wird genau dann ausgeführt, wenn dieser `TRUE` ist.

Bei Funktions- und Funktionsbausteinaufrufen ist zu beachten, dass nicht diese selbst Ziel der eingehenden Signalflusslinien sind, sondern ihre entprechenden Eingabevariablen. Da Funktionsaufrufe Ausdrücke sind, können diese jedoch als Quelle einer Signalflusslinie fungieren. Bei Funktionsbausteinaufrufen sind wiederum die entsprechenden Ausgabevariablen Quelle der Signalflusslinie.

Schleifen können in Funktionsbaustein-Diagrammen durch Rückkopplung mit Signalflusslinien oder durch Labels und Sprünge realisiert werden.

Ausführungssemantik. Funktionsbaustein-Diagramme entsprechen dem datenflussorientierten Paradigma. Ein Element wird demnach genau dann ausgewertet, wenn alle Eingänge mit Werten belegt wurden.

Ein Beispiel zur Funktionsbaustein-Sprache. In Abbildung 13 ist die Definition einer Funktion `WEIGH` (inklusive visueller Deklaration) angegeben.

```
(* ----- Graphical declaration ----- *)
       +-------------------------+
       |          WEIGH          |
BOOL---|EN                    ENO|---BOOL
       |                         |
WORD---|GROSS_WEIGHT             |---WORD
INT----|TARE_WEIGHT              |
       +-------------------------+


(* ----- Function block body specification ----- *)
              +----------+                 +----------+
              |BCD_TO_INT|                 |INT_TO_BCD|
              |          |    +-------+    |          |
EN------------|EN        |    |  SUB  |    |          |
              |       ENO|----|EN  ENO|---|EN     ENO|--ENO
GROSS_WEIGHT--|          |    |       |    |          |
              |          |----|       |---|          |--WEIGH
              +----------+    |       |    +----------+
TARE_WEIGHT-------------------|       |
                              +-------+
```

Abbildung 13: Der Funktionsbausteintyp `WEIGH` in FBS

Die Funktion erhält ein Bruttogewicht (`GROSS_WEIGHT`) als binär kodierte Dezimalzahl (BCD) und ein Leergewicht (`TARE_WEIGHT`) als Ganzzahl. Berechnet und zurückgegeben werden soll das Nettogewicht als BCD.

Dazu wird zuerst das Bruttogewicht in einen `INT` konvertiert, davon das Leergewicht abgezogen und dieses Ergebnis wieder in eine binär kodierte Dezimalzahl zurück konvertiert.

Die Signalflusslinien, die im Beispiel als Zweite und Dritte zwischen den Funktionsaufrufen `BCD_TO_INT` und `SUB` gezeichnet sind, übertragen den Funktionswert von `BCD_TO_INT` und den Wert der Variablen `TARE_WEIGHT` an zwei hier nicht benannte Eingabevariablen von `SUB` und nicht direkt an den Aufruf.

3.4 Ladder Diagrams

Ladder Diagrams (LD) werden im deutschen Sprachraum auch als *Kontaktpläne (KOP)* bezeichnet. Sie sind an die Darstellung in elektromechanischen Relaissystemen angelehnt und werden zur Realisierung von boolschen Schaltungen verwendet. Der Metamodellausschnitt zu Kontaktplänen ist in Abbildung 14 angegeben.

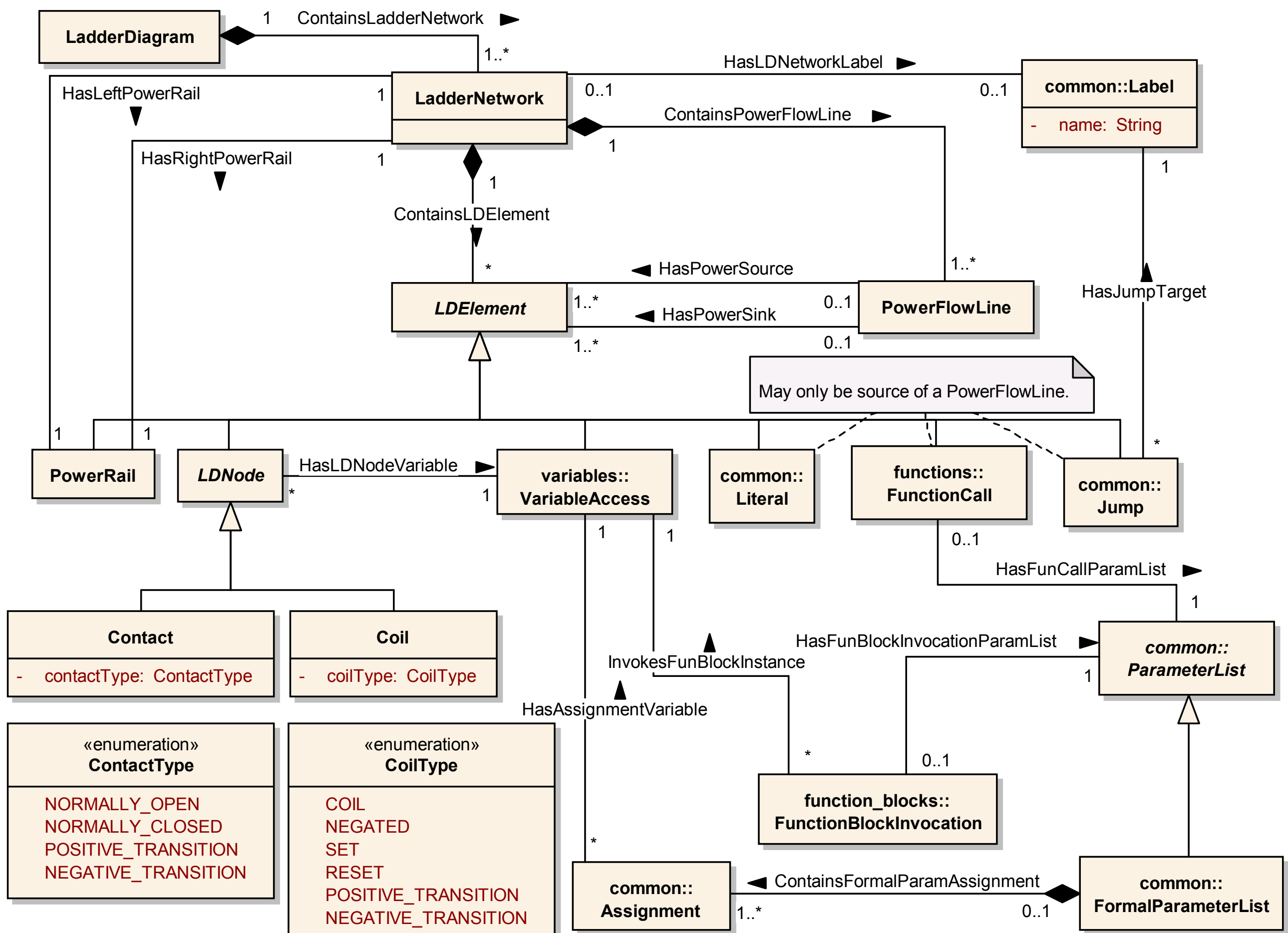

Abbildung 14: Metamodellausschnitt zu Ladder Diagrams

Ein Kontaktplan (`LadderDiagram`) besteht aus Ladder-Netzwerken (`LadderNetwork`). Jedes Netzwerk kann mit einem Label (`Label`) versehen sein.

Ein Netzwerk wird begrenzt durch eine linke und eine rechte Stromschiene (`PowerRail`), die Einstiegs- und Austiegspunkt des Netzwerks darstellen.

Kontaktplan-Elemente (`LDElement`) werden durch Stromflusslinien (`PowerFlowLine`) miteinander verbunden, welche den boolschen Wert ihres Quellelements an das Zielelement übermitteln. Eine Stromflusslinie kann aber auch mehrere Quell- und Zielelemente besitzen. Dabei wirken sich mehrere Quellelemente wie ein logisches ODER aus, d.h. die Zielelemente bekommen `TRUE` geliefert, wenn der Wert mindestens eines Quellelements `TRUE` ist.

Kontakte (`Contact`) führen ein logisches UND des Wertes des linken horizontalen Links mit einem durch eine Variable bestimmten boolschen Wert durch und leiten das Ergebnis an den rechten horizontalen Link weiter. Es lassen sich die Kontakttypen aus Tabelle 5 unterscheiden.

Symbol	Bedeutung
`A` `--| |--`	Der rechte Link wird genau dann `TRUE`, wenn der linke Link und der Wert von `A` `TRUE` ist. (`NormallyOpenContact`)
`A` `--|/|--`	Der rechte Link wird genau dann `TRUE`, wenn der linke Link den Wert `TRUE` und `A` den Wert `FALSE` hat. (`NormallyClosedContact`)
`A` `--|P|--`	Der rechte Link wird genau dann für eine Evaluation `TRUE`, wenn der linke Link `TRUE` ist und die Variable `A` einen positiven Flankenwechsel erfährt. (`PositiveTransitionContact`)
`A` `--|N|--`	Der rechte Link wird genau dann für eine Evaluation `TRUE`, wenn der linke Link `TRUE` ist und die Variable `A` einen negativen Flankenwechsel erfährt. (`NegativeTransitionContact`)

Tabelle 5: Kontakttypen

Durch eine Reihenschaltung von Kontakten kann somit eine UND-Verknüpfung der assoziierten Variablen realisiert werden. Eine Parallelschaltung realisiert hingegen eine ODER-Verknüpfung.

Während Kontakte zum Auslesen von boolschen Variablen dienen, können mit Spulen (`Coil`) boolsche Variablen gesetzt werden. Zusätzlich kopiert eine Spule ihren Eingangswert auf ihren Ausgang. Tabelle 6 führt alle Spulentypen auf.

Ausführungssemantik. Ladder Diagrams sind wie Function Block Diagrams datenflussorientiert und haben dementsprechend dieselbe Ausführungssemantik. Wiederum erfolgen Aufrufe von Funktionen und Funktionsbausteinen, indem man die Ein- und Ausgabevariablen mit Stromflusslinien in das Netzwerk integriert. Da ein `FunctionCall` selbst wieder einen Wert liefert, kann dieser auch selbst als Quelle einer `PowerFlowLine` dienen.

Ein Beispiel zu Ladder Diagrams. In Abbildung 15 ist ein Beispiel für einen Kontaktplan angeben. Es besteht aus drei Ladder-Netzwerken, von denen zwei mit Labels versehen sind.

Gilt `(X OR Y) AND Z`, so wird zu `LABEL1` gesprungen.

Das Netzwerk von `LABEL1` addiert die Werte der Variablen `V1` und `V2` und speichert das Resultat in der Variablen `SUM`. Der Funktionsaufruf ist jedoch bedingt durch den Wert der Variablen `EN_ADD`. Ob die Funktion tatsächlich aufgerufen wurde, kann über die Variable `ENO_ADD` abgefragt werden.

Danach geht die Kontrolle zurück an das erste Netzwerk. Gilt `NOT Z`, so wird zu `LABEL2` gesprungen.

Symbol	Beschreibung
`A` `--( )--`	**Spule**: Der Wert des linken Links wird zum Wert der Variablen `A`. (`Coil`)
`A` `--(/)--`	**Negierte Spule**: `A` wird auf den negierten Wert des linken Links gesetzt. (`NegatedCoil`)
`A` `--(S)--`	**Setz-Spule**: Ist der linke Link `TRUE`, so wird `A` auf `TRUE` gesetzt und behält diesen Wert bis ihn eine Rücksetz-Spule löscht. (`SetCoil`)
`A` `--(R)--`	**Rücksetz-Spule**: Ist der linke Link `TRUE`, so wird `A` auf `FALSE` zurückgesetzt und behält diesen Wert bis er von einer Setz-Spule gesetzt wird. (`ResetCoil`)
`A` `--(P)--`	**Positive Flankenerkennung**: Wird am linken Link eine positive Flanke erkannt, so wird `A` für eine Evaluation auf `TRUE` gesetzt. (`PositiveTransitionCoil`)
`A` `--(N)--`	**Negative Flankenerkennung**: Wird am linken Link eine negative Flanke erkannt, so wird `A` für eine Evaluation auf `TRUE` gesetzt. (`NegativeTransitionCoil`)

Tabelle 6: Spulentypen

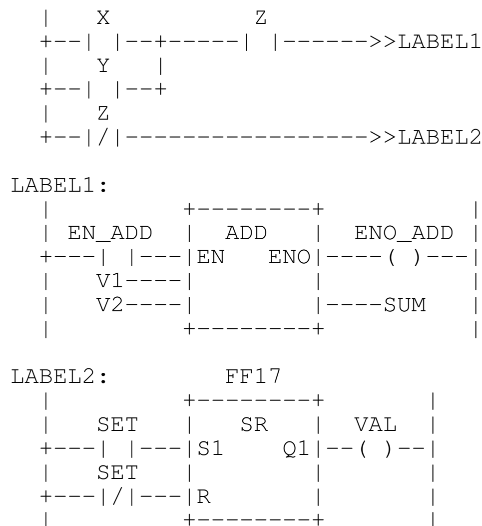

Abbildung 15: Beispiel Ladder-Diagramm

Im zugehörigen Netzwerk wird das Flipflop `FF17` gesetzt, falls die Variable `SET` den Wahrheitswert `TRUE` besitzt. Ansonsten wird es zurückgesetzt. In jedem Fall wird der aktuelle Wert in der Variablen `VAL` gespeichert.

3.5 Sequential Function Charts

Die sowohl visuell als auch textuell darstellbaren *Sequential Function Charts (SFC)* (*Ablaufsprache, AS*), welche wie die *State Machines* der UML [Obj07] zustandsorientiert sind, werden zur Strukturierung von Funktionsblöcken (Abschnitt 2.4.2) und Programmen (Abschnitt 2.4.3) eingesetzt.

In der Regel ist ein Programm in mehrere Teilaufgaben unterteilbar. Manche Teilaufgaben bedingen sich gegenseitig während andere voneinander unabhängig sind. Die Ablaufsprache bietet Mittel, um solche Sachverhalte möglichst einfach abzubilden.

In Abbildung 16 ist der Ausschnitt des Metamodells angegeben, welcher SFCs beschreibt.

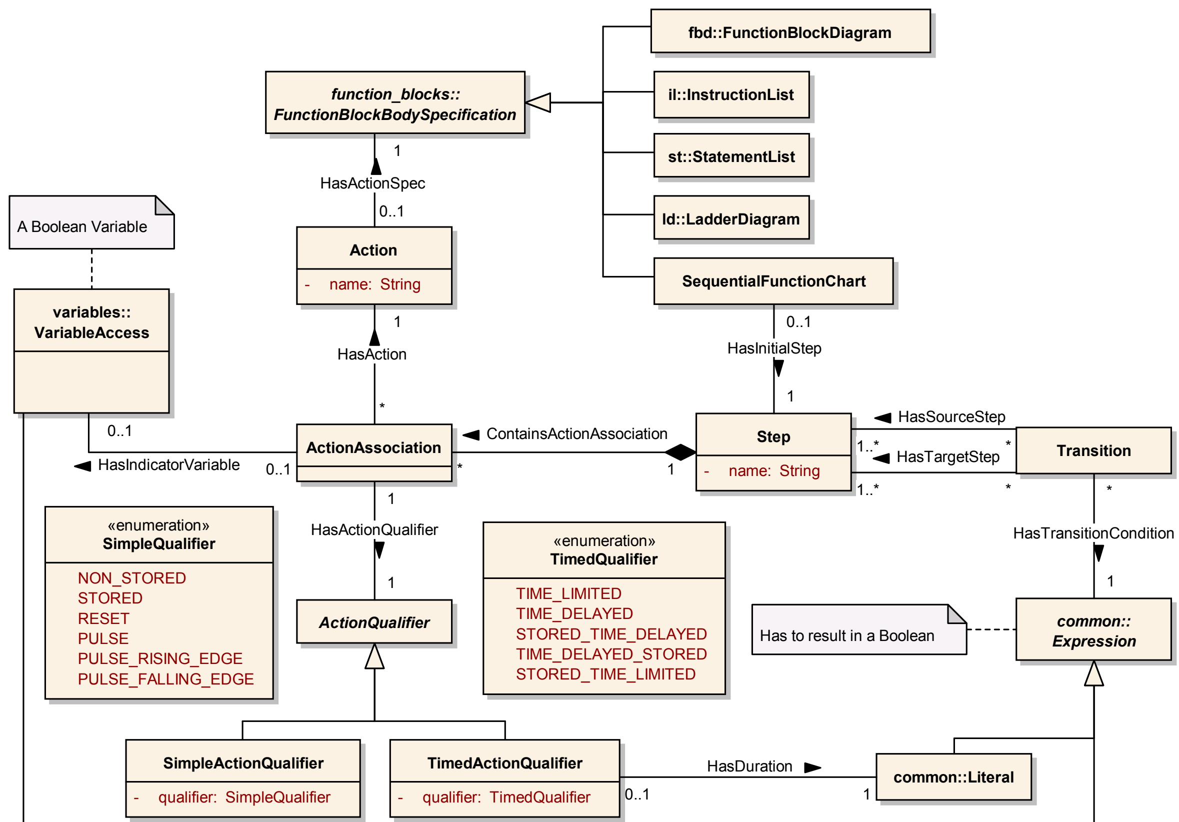

Abbildung 16: Metamodellausschnitt zur Ablaufsprache

Ein SFC besteht aus *Schritten* (`Step`, vergleichbar mit einem *State* einer UML State Machine). Genau einer davon ist als initialer Schritt gekennzeichnet und stellt den Eintrittspunkt ins Programm dar. Ein Schritt hat einen Status, der innerhalb eines SFCs über spezielle Variablen abgefragt werden kann. Beispielsweise gibt `step1.X` an, ob der Schritt 1 zur Zeit aktiv ist. `step1.T` gibt die Zeitdauer seit Aktivierung von Schritt 1 an.

Über gerichtete Linien, sogenannte *Transitionen* (`Transition`), wird die Kontrolle von einem Schritt an den nächsten übergeben. Eine Transition hat immer eine *Transitionsbedingung* (`TransitionCondition`), die bestimmt, wann eine Transition schaltet. Diese wird als boolscher Ausdruck in einer der drei Sprachen Structured Text, Ladder Diagram oder Function Block Diagram formuliert.

Eine Transition darf mehrere Nachfolgerschritte bzw. mehrere Vorgängerschritte haben. Dadurch wird eine Parallelverzweigung bzw. eine Zusammenführung einer parallelen Ausführungssequenz erreicht.

Wenn mehrere Transitionen aus einem Schritt herausführen, so ist darauf zu achten, dass die entsprechenden Transitionsbedingungen disjunkt formuliert sind.

Ist ein Schritt aktiv, so werden die assoziierten *Aktionen* (`Action`) ausgeführt. Im Gegensatz zu Funktionen enthält eine Aktion keine Variablendeklarationen und kann somit nur auf globale bzw. in der umgebenden Programmorganisationseinheit deklarierte Variablen zugreifen. Eine Aktion kann mit jeder der in IEC 61131-3 definierten Sprachen formuliert werden (`FunctionBlockBodySpecification`). Insbesondere ist eine Schachtelung mehrerer SFCs erlaubt.

Eine Aktion wird einem Schritt mittels einer *Aktionsassoziation* (`ActionAssociation`) zugeordnet. Diese legt die Ausführungsart mittels eines *Aktionsqualifizierers* (`ActionQualifier`) fest. Beispielsweise kann eine Aktion `ACTION1` im dritten Schritt nur einmal bei Aktivierung ausgeführt werden, in Schritt 17 aber kontinuierlich. Tabelle 7 listet alle möglichen Qualifizierer auf.

Aktionsqualifizierer	**Erklärung**
Einfache Qualifizierer	
`N` (non-stored)	Aktion wird solange kontinuierlich ausgeführt, wie der Schritt aktiviert ist. Ein weggelassener Qualifizierer ist gleichbedeutend mit `N`.
`S` (set)	Ausführung der Aktion wird beim Betreten des Schritts begonnen. Wird der Schritt verlassen, so wird die Aktion weiterhin ausgeführt.
`R` (reset)	Beendet die Ausführung einer mit `S` gestarteten Aktion.
`P` (pulse)	Einmalige Ausführung der Aktion.
`P1` (pulse - rising edge)	Einmalige Ausführung der Aktion beim Betreten des Schritts.
`P0` (pulse - falling edge)	Einmalige Ausführung der Aktion beim Verlassen des Schritts.
Zeitliche Qualifizierer (haben ein Argument von Typ `TIME`)	
`L` (limited)	Die Aktion wird bei Aktivierung des Schritts gestartet und nach der gegebenen Zeitdauer beendet. Wird der Schritt schon vorher verlassen, so wird auch die Aktion beendet.
`D` (delayed)	Nach Aktivierung des Schritts wird die Aktion nach Ablauf der angegebenen Verzögerungszeit gestartet. Bei Deaktivierung des Schritts wird auch die Aktion beendet.
`SD` (set and delayed)	Kombination aus `S` und `D`. Die Verzögerungszeit beginnt bei Aktivierung des Schritts und läuft auch bei dessen Deaktivierung weiter.
`DS` (delayed and set)	Wie `SD`, jedoch wird die Aktion nicht ausgeführt, wenn der Schritt vor Ablauf der Verzögerungszeit deaktiviert wird.
`SL` (set and limited)	Aktion wird bei Aktivierung des Schritts gestartet und nach der gegebenen Zeitspanne oder bei einem entsprechenden Reset (`R`) wieder beendet.

Tabelle 7: Aktionsqualifizierer

Weiterhin kann der Aktion mit der Aktionsassoziation eine boolsche Indikatorvariable zugewiesen werden. Diese kann von der Aktion zur Signalisierung von Fehlern oder Zeitüberschreitungen verwendet werden.

Ausführungssemantik. Bei der Initialisierung eines SFCs wird der initiale Schritt aktiviert.

Eine Transition wird aktiviert (*enabled*), wenn alle Vorgängerschritte aktiviert wurden. Ab diesem Zeitpunkt wird die Transitionsbedingung kontinuierlich geprüft, und sobald diese erfüllt ist, wird die Transition ausgelöst (*clearing*), wodurch die Vorgängerschritte deaktiviert und die Nachfolgerschritte aktiviert werden.

Da die Aktionen der Vorgängerschritte einer Transition Auswirkungen auf das Ergebnis der Auswertung der Transitionsbedingung haben können, soll laut Standard die erste Überprüfung der Transitionbedingung erst dann erfolgen, wenn die Effekte der Aktivierung der Schritte durch die umgebenden Programmorganisationseinheit propagiert wurden.

Ein Beispiel zu Sequential Function Charts. Ein Beispiel für einen SFC ist in Abbildung 17 angegeben.

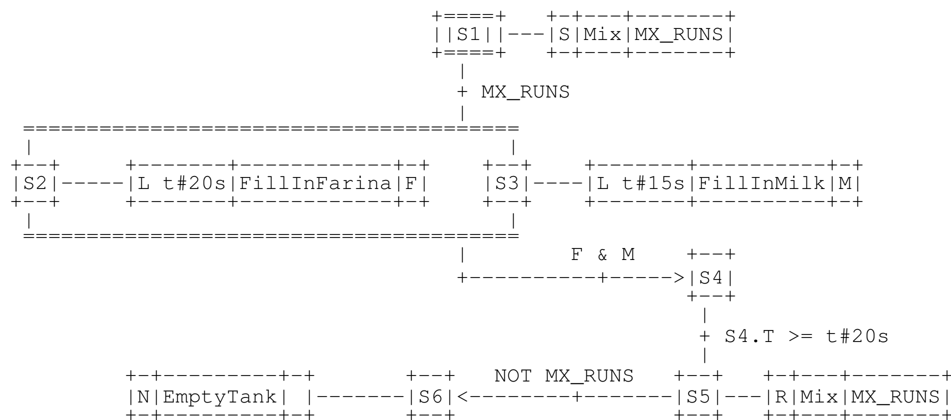

Abbildung 17: Ablaufsteuerung mit einem Sequential Function Chart

Mit dem SFC wird die Steuerung einer Maschine implementiert, die einen Kuchenteig mischt. Im initialen Schritt `S1` wird die Aktion `Mix` gestartet. Der Qualifizierer `S` bedeutet, dass die Aktion auch nach dem Verlassen des Schrittes weiterhin ausgeführt werden soll. Das Anlaufen des Mixers wird mit der Indikatorvariable `MX_RUNS` signalisiert.

Sobald `MX_RUNS` den Wert `TRUE` annimmt, kann die Transition zu `S2` und `S3` schalten. Parallel werden Milch (`FillInMilk`) und Mehl (`FillInFarina`) eingefüllt. Der Qualifizierer `L` lässt die zugehörige Aktion für die jeweils angegebene Zeitspanne ausführen, es sei denn, der Schritt wird zuvor verlassen. Da die beiden zugehörigen Aktionen aber mittels der Indikatorvariablen `F` und `M` signalisieren, wann die Zutaten in den Kessel eingefüllt wurden und die nächste Transition erst dann schalten kann, werden hier die spezifizierten Zeiten eingehalten.

Die parallele Verarbeitung wird mit der Transition zu `S4` wieder zusammengeführt. Dieser Schritt hat keine assoziierte Aktion, er realisiert nur eine Wartefunktion. Das wird dadurch erreicht, dass die Transition nach `S5` erst dann schalten kann, wenn `S4` mindestens 20 Sekunden lang aktiv war.

Im fünften Schritt wird mittels des Qualifizierers `R` die dauerhafte Ausführung der Aktion `Mix` beendet, die in Schritt `S1` gestartet wurde.

Nachdem der Mixer angehalten wurde (`MX_RUNS = FALSE`) wird der Kessel geleert.

Der SFC ist in Listing 23 noch einmal in der äquivalenten textuellen Repräsentation angegeben.

```
1   INITIAL_STEP S1 : Mix(S, MX_RUNS);   END_STEP
2   STEP S2 : FillInFarina(L, t#20s, F); END_STEP
3   STEP S3 : FillInMilk(L, t#15s, M);   END_STEP
4   STEP S4                              END_STEP
5   STEP S5 : Mix(R, MX_RUNS);           END_STEP
6   STEP S6 : EmptyTank(N);              END_STEP
7
8   TRANSITION FROM S1       TO (S2, S3) := MX_RUNS;        END_TRANSITION
9   TRANSITION FROM (S2, S3) TO S4       := F & M;          END_TRANSITION
10  TRANSITION FROM S4       TO S5       := S4.T >= t#20s;  END_TRANSITION
11  TRANSITION FROM S5       TO S6       := NOT MX_RUNS;    END_TRANSITION
12
13  ACTION Mix: (* Action Body *)            END_ACTION
14  ACTION FillInFarina: (* Action Body *)   END_ACTION
15  ACTION FillInMilk: (* Action Body *)     END_ACTION
16  ACTION EmptyTank: (* Action Body *)      END_ACTION
```

Listing 23: Beispiel: Der SCF aus Abbildung 17 in textueller Darstellung

4 Zusammenfassung

Im vorliegenden Bericht wurde der Versuch unternommen, die fünf Sprachen der Norm IEC 61131-3 durch ein integriertes Metamodell zu beschreiben, welches präzise die abstrakte Syntax der Sprachen erfasst. Dieses Metamodell ist als Vorschlag für eine Referenzbeschreibung anzusehen und kann bei Bedarf auf die im Einsatz befindlichen Varianten dieser Sprachen angepasst werden.

Zudem wurde um der Verständlichkeit willen eine knappe Einführung in die Grundzüge von speicherprogrammierbaren Steuerungen und deren Programmierung gegeben, wobei ebenfalls die Semantik der eingeführten Sprachen Berücksichtigung fand. Um die Inhalte nachvollziehbar zu halten, wurde zu den meisten Sprachelementen und jeder Sprache mindestens ein einfaches Beispiel angegeben.

Das vorgeschlagene Referenzschema orientiert sich an den Festlegungen der Norm und lässt herstellerspezifische Erweiterungen außer Acht. Daher können Abweichungen vom Standard in verschiedenen Implementationen der IEC-Sprachen mit Hilfe dieser kompakten und präzisen Beschreibung leichter fassbar und explizit darstellbar gemacht werden.

Literatur

[ERSB08] EBERT, JÜRGEN, VOLKER RIEDIGER, HANNES SCHWARZ und DANIEL BILDHAUER: *Using the TGraph Approach for Model Fact Repositories*. In: *Proceedings of the International Workshop on Model Reuse Strategies (MoRSe 2008)*, Seiten 9–18, 2008.

[ERW08] EBERT, JÜRGEN, VOLKER RIEDIGER und ANDREAS WINTER: *Graph Technology in Reverse Engineering, The TGraph Approach*. In: GIMNICH, RAINER, UWE KAISER, JOCHEN QUANTE und ANDREAS WINTER (Herausgeber): *10th Workshop Software Reengineering (WSR 2008)*, Band 126 der Reihe *GI Lecture Notes in Informatics*, Seiten 67–81, Bonn, 2008. GI.

[EWD+98] EBERT, JÜRGEN, ANDREAS WINTER, PETER DAHM, ANGELIKA FRANZKE und ROGER SÜTTENBACH: *Graph-Based Modeling and Implementation with EER/GRAL*, Kapitel 3, Seiten 33–50. Koblenzer Schriften zur Informatik. Föllbach, 1998.

[IEC00] IEC - INTERNATIONAL ELECTROTECHNICAL COMMISSION, Genf, Schweiz: *Programmable Controllers - Part 5: Communications*, 1. Auflage, 11. 2000.

[IEC03a] IEC - INTERNATIONAL ELECTROTECHNICAL COMMISSION, Genf, Schweiz: *Programmable Controllers - Part 1: General information*, 2. Auflage, 5. 2003.

[IEC03b] IEC - INTERNATIONAL ELECTROTECHNICAL COMMISSION, Genf, Schweiz: *Programmable Controllers - Part 3: Programming languages*, 2. Auflage, 1. 2003.

[Obj07] OBJECT MANAGEMENT GROUP, Needham, USA: *OMG Unified Modeling Language, Superstructure*, 2.1.2 Auflage, 11. 2007.

[Win00] WINTER, ANDREAS: *Referenz-Metaschema für visuelle Modellierungssprachen*. Deutscher Universitätsverlag, Wiesbaden, 2000. zugl. Dissertation, Institut für Informatik. Universität Koblenz-Landau.

Bisher erschienen

Arbeitsberichte aus dem Fachbereich Informatik

(http://www.uni-koblenz.de/fb4/publikationen/arbeitsberichte)

Tassilo Horn, Jürgen Ebert, Ein Referenzschema für die Sprachen der IEC 61131-3, Arbeitsberichte aus dem Fachbereich Informatik 13/2008

Thomas Franz, Ansgar Scherp, Steffen Staab, Does a Semantic Web Facilitate Your Daily Tasks?, Arbeitsberichte aus dem Fachbereich Informatik 12/2008

Norbert Frick, Künftige Anfordeungen an ERP-Systeme: Deutsche Anbieter im Fokus, Arbeitsberichte aus dem Fachbereicht Informatik 11/2008

Jürgen Ebert, Rüdiger Grimm, Alexander Hug, Lehramtsbezogene Bachelor- und Masterstudiengänge im Fach Informatik an der Universität Koblenz-Landau, Campus Koblenz, Arbeitsberichte aus dem Fachbereich Informatik 10/2008

Mario Schaarschmidt, Harald von Kortzfleisch, Social Networking Platforms as Creativity Fostering Systems: Research Model and Exploratory Study, Arbeitsberichte aus dem Fachbereich Informatik 9/2008

Bernhard Schueler, Sergej Sizov, Steffen Staab, Querying for Meta Knowledge, Arbeitsberichte aus dem Fachbereich Informatik 8/2008

Stefan Stein, Entwicklung einer Architektur für komplexe kontextbezogene Dienste im mobilen Umfeld, Arbeitsberichte aus dem Fachbereich Informatik 7/2008

Matthias Bohnen, Lina Brühl, Sebastian Bzdak, RoboCup 2008 Mixed Reality League Team Description, Arbeitsberichte aus dem Fachbereich Informatik 6/2008

Bernhard Beckert, Reiner Hähnle, Tests and Proofs: Papers Presented at the Second International Conference, TAP 2008, Prato, Italy, April 2008, Arbeitsberichte aus dem Fachbereich Informatik 5/2008

Klaas Dellschaft, Steffen Staab, Unterstützung und Dokumentation kollaborativer Entwurfs- und Entscheidungsprozesse, Arbeitsberichte aus dem Fachbereich Informatik 4/2008

Rüdiger Grimm: IT-Sicherheitsmodelle, Arbeitsberichte aus dem Fachbereich Informatik 3/2008

Rüdiger Grimm, Helge Hundacker, Anastasia Meletiadou: Anwendungsbeispiele für Kryptographie, Arbeitsberichte aus dem Fachbereich Informatik 2/2008

Markus Maron, Kevin Read, Michael Schulze: CAMPUS NEWS – Artificial Intelligence Methods Combined for an Intelligent Information Network, Arbeitsberichte aus dem Fachbereich Informatik 1/2008

Lutz Priese,Frank Schmitt, Patrick Sturm, Haojun Wang: BMBF-Verbundprojekt 3D-RETISEG Abschlussbericht des Labors Bilderkennen der Universität Koblenz-Landau, Arbeitsberichte aus dem Fachbereich Informatik 26/2007

Stephan Philippi, Alexander Pinl: Proceedings 14. Workshop 20.-21. September 2007 Algorithmen und Werkzeuge für Petrinetze, Arbeitsberichte aus dem Fachbereich Informatik 25/2007

Ulrich Furbach, Markus Maron, Kevin Read: CAMPUS NEWS – an Intelligent Bluetooth-based Mobile Information Network, Arbeitsberichte aus dem Fachbereich Informatik 24/2007

Ulrich Furbach, Markus Maron, Kevin Read: CAMPUS NEWS - an Information Network for Pervasive Universities, Arbeitsberichte aus dem Fachbereich Informatik 23/2007

Lutz Priese: Finite Automata on Unranked and Unordered DAGs Extented Version, Arbeitsberichte aus dem Fachbereich Informatik 22/2007

Mario Schaarschmidt, Harald F.O. von Kortzfleisch: Modularität als alternative Technologie- und Innovationsstrategie, Arbeitsberichte aus dem Fachbereich Informatik 21/2007

Kurt Lautenbach, Alexander Pinl: Probability Propagation Nets, Arbeitsberichte aus dem Fachbereich Informatik 20/2007

Rüdiger Grimm, Farid Mehr, Anastasia Meletiadou, Daniel Pähler, Ilka Uerz: SOA-Security, Arbeitsberichte aus dem Fachbereich Informatik 19/2007

Christoph Wernhard: Tableaux Between Proving, Projection and Compilation, Arbeitsberichte aus dem Fachbereich Informatik 18/2007

Ulrich Furbach, Claudia Obermaier: Knowledge Compilation for Description Logics, Arbeitsberichte aus dem Fachbereich Informatik 17/2007

Fernando Silva Parreiras, Steffen Staab, Andreas Winter: TwoUse: Integrating UML Models and OWL Ontologies, Arbeitsberichte aus dem Fachbereich Informatik 16/2007

Rüdiger Grimm, Anastasia Meletiadou: Rollenbasierte Zugriffskontrolle (RBAC) im Gesundheitswesen, Arbeitsberichte aud dem Fachbereich Informatik 15/2007

Ulrich Furbach, Jan Murray, Falk Schmidsberger, Frieder Stolzenburg: Hybrid Multiagent Systems with Timed Synchronization-Specification and Model Checking, Arbeitsberichte aus dem Fachbereich Informatik 14/2007

Björn Pelzer, Christoph Wernhard: System Description:“E-KRHyper“, Arbeitsberichte aus dem Fachbereich Informatik, 13/2007

Ulrich Furbach, Peter Baumgartner, Björn Pelzer: Hyper Tableaux with Equality, Arbeitsberichte aus dem Fachbereich Informatik, 12/2007

Ulrich Furbach, Markus Maron, Kevin Read: Location based Informationsystems, Arbeitsberichte aus dem Fachbereich Informatik, 11/2007

Philipp Schaer, Marco Thum: State-of-the-Art: Interaktion in erweiterten Realitäten, Arbeitsberichte aus dem Fachbereich Informatik, 10/2007

Ulrich Furbach, Claudia Obermaier: Applications of Automated Reasoning, Arbeitsberichte aus dem Fachbereich Informatik, 9/2007

Jürgen Ebert, Kerstin Falkowski: A First Proposal for an Overall Structure of an Enhanced Reality Framework, Arbeitsberichte aus dem Fachbereich Informatik, 8/2007

Lutz Priese, Frank Schmitt, Paul Lemke: Automatische See-Through Kalibrierung, Arbeitsberichte aus dem Fachbereich Informatik, 7/2007

Rüdiger Grimm, Robert Krimmer, Nils Meißner, Kai Reinhard, Melanie Volkamer, Marcel Weinand, Jörg Helbach: Security Requirements for Non-political Internet Voting, Arbeitsberichte aus dem Fachbereich Informatik, 6/2007

Daniel Bildhauer, Volker Riediger, Hannes Schwarz, Sascha Strauß, „grUML – Eine UML-basierte Modellierungssprache für T-Graphen“, Arbeitsberichte aus dem Fachbereich Informatik, 5/2007

Richard Arndt, Steffen Staab, Raphaël Troncy, Lynda Hardman: Adding Formal Semantics to MPEG-7: Designing a Well Founded Multimedia Ontology for the Web, Arbeitsberichte aus dem Fachbereich Informatik, 4/2007

Simon Schenk, Steffen Staab: Networked RDF Graphs, Arbeitsberichte aus dem Fachbereich Informatik, 3/2007

Rüdiger Grimm, Helge Hundacker, Anastasia Meletiadou: Anwendungsbeispiele für Kryptographie, Arbeitsberichte aus dem Fachbereich Informatik, 2/2007

Anastasia Meletiadou, J. Felix Hampe: Begriffsbestimmung und erwartete Trends im IT-Risk-Management, Arbeitsberichte aus dem Fachbereich Informatik, 1/2007

„Gelbe Reihe“

(http://www.uni-koblenz.de/fb4/publikationen/gelbereihe)

Lutz Priese: Some Examples of Semi-rational and Non-semi-rational DAG Languages. Extended Version, Fachberichte Informatik 3-2006

Kurt Lautenbach, Stephan Philippi, and Alexander Pinl: Bayesian Networks and Petri Nets, Fachberichte Informatik 2-2006

Rainer Gimnich and Andreas Winter: Workshop Software-Reengineering und Services, Fachberichte Informatik 1-2006

Kurt Lautenbach and Alexander Pinl: Probability Propagation in Petri Nets, Fachberichte Informatik 16-2005

Rainer Gimnich, Uwe Kaiser, and Andreas Winter: 2. Workshop "Reengineering Prozesse" – Software Migration, Fachberichte Informatik 15-2005

Jan Murray, Frieder Stolzenburg, and Toshiaki Arai: Hybrid State Machines with Timed Synchronization for Multi-Robot System Specification, Fachberichte Informatik 14-2005

Reinhold Letz: FTP 2005 – Fifth International Workshop on First-Order Theorem Proving, Fachberichte Informatik 13-2005

Bernhard Beckert: TABLEAUX 2005 – Position Papers and Tutorial Descriptions, Fachberichte Informatik 12-2005

Dietrich Paulus and Detlev Droege: Mixed-reality as a challenge to image understanding and artificial intelligence, Fachberichte Informatik 11-2005

Jürgen Sauer: 19. Workshop Planen, Scheduling und Konfigurieren / Entwerfen, Fachberichte Informatik 10-2005

Pascal Hitzler, Carsten Lutz, and Gerd Stumme: Foundational Aspects of Ontologies, Fachberichte Informatik 9-2005

Joachim Baumeister and Dietmar Seipel: Knowledge Engineering and Software Engineering, Fachberichte Informatik 8-2005

Benno Stein and Sven Meier zu Eißen: Proceedings of the Second International Workshop on Text-Based Information Retrieval, Fachberichte Informatik 7-2005

Andreas Winter and Jürgen Ebert: Metamodel-driven Service Interoperability, Fachberichte Informatik 6-2005

Joschka Boedecker, Norbert Michael Mayer, Masaki Ogino, Rodrigo da Silva Guerra, Masaaki Kikuchi, and Minoru Asada: Getting closer: How Simulation and Humanoid League can benefit from each other, Fachberichte Informatik 5-2005

Torsten Gipp and Jürgen Ebert: Web Engineering does profit from a Functional Approach, Fachberichte Informatik 4-2005

Oliver Obst, Anita Maas, and Joschka Boedecker: HTN Planning for Flexible Coordination Of Multiagent Team Behavior, Fachberichte Informatik 3-2005

Andreas von Hessling, Thomas Kleemann, and Alex Sinner: Semantic User Profiles and their Applications in a Mobile Environment, Fachberichte Informatik 2-2005

Heni Ben Amor and Achim Rettinger: Intelligent Exploration for Genetic Algorithms – Using Self-Organizing Maps in Evolutionary Computation, Fachberichte Informatik 1-2005