Interner Bericht

Der ProLan-X - Sprachreport

Eine objekt-orientierte Sprache für die Modellierung
von Software-Prozessen

P. Knauber
M. Verlage
W. Schramm

220 / 92

Fachbereich Informatik

Universität Kaiserslautern · Postfach 3049 · D-6750 Kaiserslautern

Der ProLan-X - Sprachreport

Eine objekt-orientierte Sprache für die Modellierung
von Software-Prozessen

P. Knauber
M. Verlage
W. Schramm

220 / 92

Herausgeber: AG Programmiersprachen und Compilerbau
Leiter: Prof. Dr.-Ing. H.-W. Wippermann

Kaiserslautern, April 1992

Zusammenfassung

Bei der Realisierung großer Software-Projekte treten immer wieder Probleme auf, was die Koordination der Mitarbeiter, die Ausnutzung der vorhandenen Ressourcen und nicht zuletzt die Qualität der erzeugten Produkte angeht. Um die Vorgänge bei der Produktion von Software durchschaubarer und verständlicher zu machen, versucht man, diese aus der Sicht von Meta-Modellen zu beschreiben. Dabei fließen die individuellen Rahmenbedingungen einer jeden Entwicklungsumgebung ein; die vorhandenen Ressourcen werden ebenso modelliert wie die durchzuführenden Tätigkeiten und ihre Abhängigkeiten.

Die Beschreibungssprache für den Software-Prozeß ProLan-X dient der (konkreten) Beschreibung der Bestandteile des Meta-Modells MoMo, das ebenfalls in dieser Arbeitsgruppe entwickelt wurde [Schramm]. Die am Projekt beteiligten Personen, Hardware- und Software-Ressourcen und ihre Aufgaben werden in möglichst natürlicher Weise verhaltensorientiert beschrieben. Aus dieser Beschreibung kann eine Ablaufumgebung generiert werden, die die Durchführung des Projekts unterstützt und protokolliert.

Der vorliegende Bericht faßt die Eigenschaften der Sprache ProLan-X zusammen und erläutert ihre Verwendung. Er setzt das MoMo-Modell als bekannt voraus.

1. Einleitung 1

2. Die Umgebung von ProLan-X 3
2.1 ProLan-Komponente 3
2.2 Die virtuelle Prozess Maschine (VPM) 4
2.3 Der Temporal Reasoner (TEMPO) 4
2.4 Die Analyse Komponente 4
2.5 Abschließende Bemerkungen zum System 5

3. Report von ProLan-X 6
3.1 Syntax-Notation 6
3.2 Vokabular und Repräsentation von Terminalsymbolen 7
3.3 Gültigkeitsbereiche und Vereinbarungen 9
3.3.1 Symbol- und Konstanten-Vereinbarungen 9
3.3.2 Ereignis-Vereinbarungen 10
3.3.3 Typ-Vereinbarungen 10
3.3.4 Variablen- und Attribut-Vereinbarungen 11
3.3.4.1 Variablen in Routinen 11
3.3.4.2 Attribute in Klassen 12
3.4 Ausdrücke 16
3.4.1 Operanden 16
3.4.2 Operatoren 16
3.5 Projektbeschreibung 21
3.5.1 Vererbung 21
3.5.2 Entitäten 23
3.5.3 Aktivitäten 24
3.5.3.1 Gültigkeit von Aktivitäten 25
3.5.3.2 Anweisungsteil einer Aktivität 26
3.5.4 Zusicherungen 30
3.5.5 Wächter 30
3.5.6 Intervalle 31
3.5.7 Meilensteine 32
3.6 Schnittstelle zum System 34
3.6.1 Vereinbarungen von Funktionen und Tools 34
3.6.2 Anweisungen in Routinen 35
3.7 Projekte 39
3.8 Zusammenfassung der Syntaxregeln 40
3.9 Liste der Operatoren und Schlüsselwörter 46
3.10 Standardfunktionen von ProLan-X 47
3.10.1 Allgemeines 47
3.10.2 Trigger-Funktionen 47

3.10.3 Laufzeit-spezifische Standard-Funktionen 48
3.10.4 Quantoren 51
3.10.5 ProLan-spezifische Standard-Funktionen 52
3.10.6 Sonstige Standard-Funktionen 53

4. Literatur **54**

1. Einleitung

Die Herstellung großer Software-Systeme und deren Wartung bzw. Weiterentwicklung verläuft nicht immer zur vollen Zufriedenheit des Kunden und der Entwickler. Die Gründe für die unbefriedigende Qualität der ausgelieferten Produkte sind mannigfaltig. Bei der Produktion von Software handelt es sich zwar um eine Tätigkeit, bei der Kreativität benötigt wird, da ja immer neue Probleme gelöst werden müssen, die geforderten Eigenschaften des Ergebnisses erfordern jedoch eine ingenieurmäßige Behandlung dieses Bereiches. Es wurden und werden daher Modelle und Ansätze entwickelt, mit denen die Entwicklung von Software systematisiert wird. Dies bezieht sich sowohl auf die Verwendung von Techniken (z.B. objektorientiertes Design) wie auch auf die hier betrachtete Schematisierung der Abläufe.

In der Vergangenheit wurde jedoch der Fehler gemacht, die untersuchten Abläufe verschiedener Umgebungen einheitlich zusammenzufassen und dann als allgemeingültig zu propagieren. Man übersah dabei, daß jede Produktionsstätte subjektive Anforderungen stellt. Auch der Unwille der Organisationen, von außen diktierte Abläufe zu übernehmen, trägt sicherlich dazu bei. Basierend auf diesen Erkentnissen kehrt man nun die Richtung um: es wird ein Meta-Modell bereitgestellt, mit dem die bestehende Umgebung und die darin ablaufenden Prozesse beschrieben werden können. Basierend auf dieser Beschreibung versucht man, die Fehler oder Unzulänglichkeiten zu diagnostizierten und zu beheben. Dadurch ist es möglich die Abläufe zu verbessern, sie werden quantitativ vergleichbar und einer Wiederverwendung zugänglich gemacht. Die Tätigkeit der Beschreibung nennt man Software-Prozess-Modellierung (im folgenden SPM abgekürzt).

Von Projekt zu Projekt werden immer neue Elemente des Software-Prozesses hinzukommen oder bereits bestehende verbessert. Es findet gewissermaßen eine Evolution der Beschreibungen einzelner Elemente über mehrere Projekte statt, an deren Ende abstrakte Komponenten stehen. Somit entsteht eine Bibliothek von Beschreibungen, die mit Erfahrungen aus durchgeführten Projekten attributiert werden können. Dadurch wird es möglich, Projektteile leichter zu identifizieren und wiederzuverwenden.

Da die Beschreibung dokumentiert, analysiert und übermittelt werden soll, benutzt man selbstverständlich eine Sprache. Sie ist für das Problemfeld geeignet, wenn sie auf einem entsprechenden Abstraktionsniveau eine möglichst fehlerfreie Kommunikation der Projektmitglieder erlaubt. Die Sprache muß daher Mittel zur Verfügung stellen, um mittels einer problemspezifischen Terminologie die Abläufe in einem Projekt darstellbar zu machen. Man spricht hier auch von einer "natürlichen Sicht". Es wird daher klar, daß man keine übliche Programmiersprache verwenden kann, da sie zu allgemein gehalten ist. Das Kommunikationsmittel muß aus Gründen der Effizienz dem Inhalt der Nachrichten angepaßt werden. Darüber hinaus wird der Anwender die Methode wesentlich früher und besser verwenden können, wenn er die Fakten in seiner Sprache beschreiben kann.

Erst in zweiter Linie ist die Ausführbarkeit der Beschreibung von Interesse. Man muß dabei berücksichtigen, daß auch hier die Projekte besondere Anforderungen stellen. So ist die Entwicklung von Software gekennzeichnet durch hohe Dynamik und Nicht-Determinismus.

Es besteht also der Bedarf nach einer formalisierten Beschreibung von Software-Prozessen. Diese Beschreibung muß vor allem folgenden Gesichtspunkten Rechnung tragen:

1. Die Beziehungen der am Projekt beteiligten Personen und Objekte sollen anwendungsunabhängig, aber dennoch realitätsnah formulierbar sein.

2. Die Sprache soll für den Benutzer lesbar sein, ohne dabei an Aussagefähigkeit zu verlieren.

3. Die beschriebenen Sachverhalte sollen durch einen Übersetzungsvorgang in eine, der jeweiligen Problemstellung angepaßte, integrierte Entwicklungsumgebung transferiert werden können.

4. Bewährte Elemente der Beschreibung von aktuellen oder bereits beendeten Projekten sollen einfach in neue Projekte integriert werden können.

5. Es muß die Möglichkeit bestehen, zur Laufzeit eines Projekts dynamisch Änderungen an seiner Beschreibung zu vorzunehmen, die sofort wirksam werden.

Aus den oben genannten Gründen haben wir die Sprache ProLan entwickelt; sie stellt ein Mittel zur Verfügung, welches dem Benutzer die Beschreibung von Software Prozessen aus der Sicht des Meta-Modells MoMo ermöglicht. MoMo wurde auch am Lehrstuhl *Compilerbau und Programmiersprachen* bei Prof. Wippermann von W. Schramm entwickelt. Die Konzepte dieses Meta-Modells haben das Aussehen der Sprache ProLan natürlich stark beeinflußt. Die schnelle prototypische Implementierung des ausführenden Systems erlaubte eine intensive Rückkopplung mit Benutzern der Sprache. Die Erkenntnisse bei der Beschreibung von Beispielprojekten haben an einigen Stellen der Sprache zu Änderungen geführt. Deshalb heißt die aktuelle Version unserer Sprache ProLan-X (X steht für engl. *extended*).

Der vorliegende Bericht faßt die Eigenschaften der Sprache ProLan-X zusammen und erläutert ihre Verwendung. Er setzt das MoMo-Modell als bekannt voraus. Für weitergehende Betrachtungen zu dem Thema Software-Prozess-Modellierung verweisen wir auf die Arbeiten. Technische Details der Sprache und des Systems werden in [Schramm], [Knauber], [Verlage] und [Verlage, Knauber] beschrieben.

2. Die Umgebung von ProLan-X

Dem Benutzer steht zur Beschreibung von Software-Projekten nicht nur die Sprache selbst zur Verfügung, sondern er wird bei seiner Arbeit von einem ganzen System unterstützt. Die Teile des Systems enthalten alle von uns für relevant erachteten Mechanismen zur Modellierung der Software-Prozesse.

Die Anforderungen an die einzelnen Komponenten ergaben sich aus der Analyse bereits existierender Systeme aus dem Bereich der Prozessmodellierung (siehe [Verlage], Kapitel 2). Im einzelnen sollte ein zukünftiges System folgende Eigenschaften besitzen:

1. Organisation der Daten als Objekte im Sinne des objektorientierten Ansatzes.
2. Repräsentation von den Elementen (Prozesse, Aktivitäten, Funktionen, ...), die einen Projektzustand in einen anderen überführen.
3. Dynamische Änderung, Ergänzung und Erweiterung dieser Elemente und ihrer Typen.
4. Hierarchischer Aufbau der Repräsentationen.
5. Darstellung verschiedenartigster Beziehungen zwischen den Repräsentationen.
6. Starke und dennoch flexible Einbindung der Umgebung.
7. Angemessene Einbeziehung der (des) Benutzer(s).

Unser System besteht im einzelnen aus den folgenden Komponenten:.

2.1 ProLan-Komponente

Die Beschreibung der Umgebung und die einzelnen Prozesse werden mittels ProLan-X dem System bekannt gemacht. Der ProLan-Compiler überprüft die syntaktische und teilweise die semantische Korrektheit der spezifizierten Projektdaten. Übersetzt wird nach CLOS (Common Lisp Object System) und CPD (Correct Project Description, s. [Knauber]), einer selbst entworfenen Zwischendarstellung, die Listen und Token zur Darstellung der unterschiedlichen Ablaufstrukturen innerhalb eines Prozesses verwendet. Vom Compiler werden Klassen und Methoden generiert, die z.T. alte Definitionen ersetzen oder ergänzen können. Weitere Aufgaben der ProLan-Komponente sind die Archivierung der ProLan-Beschreibungen (auch als *ProLan-Dokumente* bezeichnet) und die Dokumentation der Beziehungen zwischen den einzelnen Projekten.

2.2 Die virtuelle Prozess Maschine (VPM)

Hauptaufgabe der VPM ist die Interpretation der übersetzten ProLan-Dokumente. Grundlage sind Objekte der Klassen ENTITY und ACTIVITY. Basierend auf den vom Benutzer spezifizierten Klassen werden Objekte erzeugt und die Inhalte der Aktionsteile dann ausgeführt. An die Klassen können weitere Informationen gebunden werden, die nicht unbedingt zur Instanziierung benötigt werden; es kann sich hierbei um extrapolierte Daten aus anderen Projekten handeln, z.B. voraussichtliche Zeitdauer oder geschätzte anfallende Kosten.

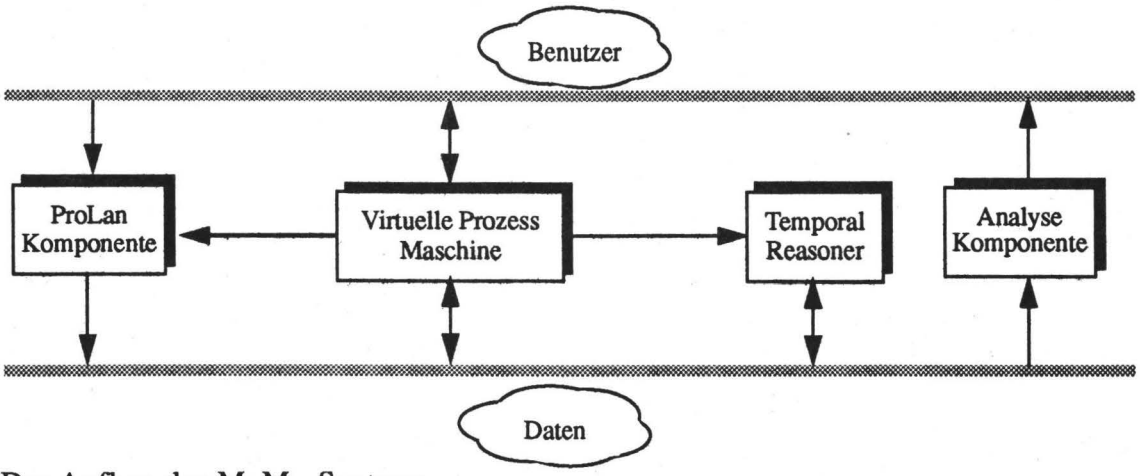
Die VPM veranlaßt auch die nachträgliche Übersetzung von Teilen der Projektbeschreibung, die erst zur Laufzeit ergänzt werden. Nachdem der Compiler im Rahmen seiner Möglichkeiten die Richtigkeit der Angaben verifiziert hat, stehen sie der Ablaufumgebung zur Verfügung.

2.3 Der Temporal Reasoner (TEMPO)

Bei TEMPO handelt es sich um ein Werkzeug zur Behandlung von zeitlichen Aspekten bei Software-Entwicklungs-Prozessen. Es wird von der VPM dann aktiviert, wenn genauere zeitliche Daten entweder durch den Benutzer oder eine Auswertung bekannt geworden sind. TEMPO arbeitet ebenso wie die VPM auf Klassen, vor allem aber auf den Objekten. Besonders wichtig sind hier die aus dem ProLan-Programm extrahierten Beschränkungen bezüglich der Auswertungsreihenfolge (s. Abschnitt 3.5.3.2 auf Seite 26). Auftretende Inkonsistenzen werden u.U. mit Hilfe des Benutzers aufgelöst. Die bisherige Anbindung an das MoMo-System muß als rudimentär bezeichnet werden, es zeigt sich jedoch, daß der Ansatz vielversprechend ist.

2.4 Die Analyse Komponente

Dieser Teil des MoMO-Systems befindet sich noch in der Planungsphase, wir wollen ihn jedoch hier erwähnen, da er ein wichtiger Bestandteil einer realen Umgebung sein muß. Ausgehend von den Objekten des momentan laufenden Projekts sollen genau wie bei TEMPO dem Benutzer abgeleitete Daten zur Steuerung der Prozesse vermittelt werden. Außerdem sind die den Klassen assoziierten Informationen an Hand der aktuellen Erkenntnisse zu verifizieren und eventuell zu verbessern.



Der Aufbau des MoMo-Systems

2.5 Abschließende Bemerkungen zum System

In der aktuellen Realisierung handelt es sich nur um eine erste Version. Bei der Entwicklungen des Prototypen ging es in erster Linie um eine schnelle Verfügbarkeit zum Test der Einsatzfähigkeit der Sprache und der Tragfähigkeit des Modells. Es war nicht Ziel der Arbeiten einen vollständigen und voll funktionsfähigen Generator für Entwicklungsumgebungen bereit zu stellen. Dennoch waren die ersten Erfahrungen mit der ProLan-Komponente positiv. Die Eigenschaften der die Prozesse beschreibenden Objekte versprechen einen günstigen Ansatz zur Untersuchung von Software-Entwicklungs-Prozessen.

3. Report von ProLan-X

3.1 Syntax-Notation

Für die Notation der Syntax wird die erweiterte Backus-Naur-Form eingesetzt. Im einzelnen werden folgende Konstruktoren verwendet:

=	trennt den Namen eines Nonterminals von seiner Definition.
[]	Optionalklammern.
{}	Repetition 0 bis beliebig oft.
$\{\}^+$	Repetition mindestens 1 mal.
{ ... //" "}	Bei Wiederholung werden die vor den Schrägstrichen stehenden Symbole durch den in Hochkommata eingeschlossenen String getrennt.
|	Alternative.
()	Gruppierung.
Capital	Nonterminale werden mit Großbuchstaben begonnen und nicht fett gedruckt.
Fettdruck	kennzeichnet Terminalsymbole; werden die hier definierten Metasymbole als Terminalsymbole der Sprache verwendet, sind sie zusätzlich in einfache oder doppelte Hochkommata (‘’ oder “”) eingeschlossen.
.	schließt eine Produktion ab.

3.2 Vokabular und Repräsentation von Terminalsymbolen

Terminale Symbole der Sprache werden mit ASCII-Zeichen notiert. Folgende lexikalischen Einschränkungen müssen beachtet werden:

1. Groß- und Kleinbuchstaben werden unterschieden.
2. Leerzeichen und Zeilenumbruch trennen aufeinanderfolgende Terminalsymbole, außer Leerzeichen in Zeichenketten (Strings).

Terminale Symbole von ProLan-X werden durch folgende Syntax-Regeln beschrieben:

1. Zahlen

Zahlen sind ganzzahlig (Integer) oder rational (Real) mit oder ohne Vorzeichen und werden im Dezimalsystem notiert. Eine rationale Zahl besteht aus einem Dezimalpunkt und mindestens einer Ziffer davor und danach.

```
Integer = { digit }+.
Real =    { digit }+ "." { digit }+.
digit =   "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
```

2. Zeichenketten

Zeichenketten (Strings) werden in einfache (') oder doppelte (") Hochkommata eingeschlossen. Das öffnende und das schließende Hochkomma muß übereinstimmen und darf nicht innerhalb der Zeichenkette verwendet werden. Zeichenketten beginnen mit dem Index 1.

```
String =         '"' { character } '"' | "'" { character } "'".
character =      alpha_char | digit | special_char.
alpha_char =     "A" | ... | "Z" | "a" | ... | "z".
special_char =   "_" | "-" | "!" | "@" | "$" | "%" | ...
```

3. Zeitangaben

In der aktuellen Version von ProLan-X sind die Angaben von Zeitpunkt und -dauer in den vorgegebenen Einheiten am gregorianischen Kalender fixiert. Die Integerwerte müssen innerhalb des im Kalender-System spezifizierten Wertebereichs liegen.

```
Time =       ( [ HH [ ":" MM [":" SS] ] ] DD ".: MM "." YYYY ) |
             ( MM "." YYYY ) |
             YYYY.
YYYY =       MM = DD = HH = SS = Integer.
Duration =   ( [HH "h"] [MM "min"] [S "s"]
               [DD "d"] [MM "m"] [YYYY "y"] ).
```

4. Bezeichner (Namen)

Bezeichner bestehen aus Folgen von Buchstaben, Zahlen, dem Trennstrich ("-") und dem Unterstrich ("_") und müssen mit einem Buchstaben beginnen.

```
project_object_name =   assertion_name | guard_name | field_name.
```

```
field_name =            attribute_name [ Actual_Parameters ] |
                        reference_name.
reference_name =        entity_name | activity_name | interval_name |
                        milestone_name.
routine_name =          tool_name | function_name | method_name |
                        std_function_name | std_trigger_name.
.._name =               Ident.
Ident =                 alpha_char { name_char }.
name_char =             alpha_char | digit | "_" | "-".
```

Die Unterscheidung in einzelne Bezeichnergruppen ist nur zur besseren Lesbarkeit der Syntax vorgenommen worden.

Nur dann, wenn es sich beim field_name um ein Methoden-Attribut (s. Abschnitt 3. auf Seite 13) handelt, darf eine Liste aktueller Parameter angegeben werden.

5. Namenskonstanten

Die Bezeichner TRUE und FALSE sind mit den entsprechenden Bool'schen Werten *wahr* und *falsch* vordefiniert und dürfen nicht verändert werden. NIL ist die Zeigerkonstante für den undefinierten Wert.

```
Boolean =           TRUE | FALSE
Link =              NIL.
```

6. Operatoren und Schlüsselwörter

Die Operatoren und Schlüsselwörter der Sprache ProLan-X sind in Kapitel 3.9 auf Seite 46 aufgelistet. Die Schlüsselwörter müssen großgeschrieben werden und können nicht als Bezeichner verwendet werden.

7. Kommentare

Kommentare werden durch 2 Trennstriche ("--") eingeleitet und enden mit einem Zeilenumbruch.

3.3 Gültigkeitsbereiche und Vereinbarungen

Alle Namen, die in einem ProLan-X - Programm verwendet werden, müssen deklariert werden (Ausnahme sind Standard-Funktionen). Der Gültigkeitsbereich eines Bezeichners ist der Block seiner Deklaration. In eingeschachtelten Blöcken kann die (globale) Bedeutung eines Bezeichners jedoch durch eine lokale Bedeutung überdeckt werden. Innerhalb des gleichen Gültigkeitsbereichs dürfen Bezeichner nicht mehrfach vereinbart werden.

Zur Sprache ProLan-X gehören eine ganze Anzahl von Standard-Funktionen. Sie können zur Abfrage von Zuständen und Ereignissen oder zur Ablaufsteuerung eingesetzt werden. Standard-Funktionen sind im gesamten Programm gültig. Eine genaue Beschreibung der vordefinierten Funktionen befindet sich in Kapitel 3.10

Die allgemeine Form eines Bezeichners (Identifier) ist ein qualifizierter Bezeichner (Qualident). Diese werden später näher erläutert (s. Abschnitt 3.4 auf Seite 16).

3.3.1 Symbol- und Konstanten-Vereinbarungen

Eine Symbol- und Konstanten-Vereinbarungen binden einen Namen an einen Wert.

```
Symbol_Declaration =     { symbol_name // "," }+ ":" SYMBOL.
Constant_Declaration =  constant_name "=" Const_Expr.
```

Symbole dienen der besseren Lesbarkeit eines Programmes. Sie entsprechen etwa den Aufzählungstypen anderer Programmiersprachen (z.B. PASCAL [Jensen, Wirth], MODULA-2 [Wirth]), es ist jedoch keine Ordnungsrelation auf ihnen definiert. Symbole erhalten eindeutige, vom System generierte Werte, die nur auf Gleichheit und Ungleichheit getestet werden können. Wegen der großen Bedeutung bzgl. der Lesbarkeit eines Software-Prozeßmodells wurde für diese Art der symbolischen Konstanten ein eigenes syntaktisches Konstrukt gewählt.

Konstanten erhalten Werte, die zum Zeitpunkt ihrer Definition auswertbar sein müssen. Sie werden bei der Übersetung in der Reihenfolge ihres Auftretens berechnet, d.h. bereits definierte Konstanten können zur Berechnung anderer Konstanten verwendet werden, ebenso wie einige vordefinierte Funktionen. Die Werte von Konstanten können zur Laufzeit eines Projekts nicht mehr verändert werden.

Beispiel:

```
MODULA-2, C,
undefined       : SYMBOL;              -- Symbol-Vereinbarung
False           = (MODULA-2 = C);      -- Konstanten-Vereinbarung
Start-State     = undefined;           -- Konstanten-Vereinbarung
```

Schlüsselwörter und Objektnamen bzw. Namen von Benutzerklassen sind nicht als Konstanten- oder Symbolnamen erlaubt; jedoch sind die Namen der System- und benutzerdefinierten Klassen als Symbole zu verwenden.

3.3.2 Ereignis-Vereinbarungen

Zu den elementaren Eigenschaften unseres Projektplanungs- und -abwicklungssystems gehört die Fähigkeit, auf spezielle Ereignisse (Events) zu reagieren.

```
Event_Declaration = { event_name // "," }+ ":" EVENT.
```

Da der Event-Mechanismus auch sinnvoll auf Benutzereingriffe in das System anwendbar ist, können bestimmte Ereignistypen definiert werden, sowie die Reaktion auf diese Ereignisse (s. Abschnitt 3.5.5 auf Seite 30). Die so definierten Ereignisse kann der Benutzer über ein Menü auslösen. Wir sprechen bei dieser Art von Ereignissen auch von Benutzerereignissen (Benutzerevents).

Beispiel:

```
Start-Project, Stop-Project, Calculate-Used-Money : EVENT;
```

Schlüsselwörter und Objektnamen bzw. Namen von Benutzerklassen sind nicht als Eventnamen erlaubt.

3.3.3 Typ-Vereinbarungen

Der Typ einer Variablen bestimmt, welche Werte sie annehmen kann und welche Operatoren auf sie anwendbar sind. In ProLan-X können keine neuen Typen definiert werden. Es ist jedoch möglich, die vorgegebenen Systemklassen

ENTITY	(Entität, s. Abschnitt 3.5.2 auf Seite 23),
ACTIVITY	(Aktivität, s. Abschnitt 3.5.3 auf Seite 24),
ASSERTION	(Zusicherung, s. Abschnitt 3.5.4 auf Seite 30),
GUARD	(Wächter, s. Abschnitt 3.5.5 auf Seite 30) und
INTERVAL	(Interval, s. Abschnitt 3.5.6 auf Seite 31)
MILESTONE	(Meilenstein, s. Abschnitt 3.5.7 auf Seite 32)

zu konkretisieren bzw. (zu Benutzerklassen) zu erweitern.

Außerdem gibt es bestimmte Grundtypen.

```
Simple_Type =     INTEGER | REAL | BOOLEAN | STRING | SYMBOL | STATUS
                  | Temporal_Type.
Temporal_Type =   TIME | DURATION.
```

Diese Grundtypen werden durch Schlüsselworte gekennzeichnet und beinhalten folgende Wertebereiche:

1. INTEGER
   Alle ganzzahligen Werte.

2. REAL
   Rationale Zahlen, deren Genauigkeit von der Implementierung abhängt.

3. BOOLEAN
   Die Wahrheitswerte *wahr* (TRUE) und *falsch* (FALSE).

4. STRING
   Zeichenketten, die alle Zeichen mit Ordinalwerten von 1 bis 255 enthalten dürfen (vgl. Definition 2. auf Seite 7).

5. SYMBOL
   Namen, die (vom Benutzer) als Symbol deklariert wurden (s. Abschnitt 3.3.1 auf Seite 9).

6. STATUS
   Eine Menge beliebiger Kardinalität von Symbolen.

7. TIME
   Zeitpunkte des gregorianischen Kalendersystems (s. Abschnitt 3. auf Seite 7).

8. DURATION
   Zeitintervalle zwischen 2 Zeitpunkten des gregorianischen Kalendersystems (s. Abschnitt 3. auf Seite 7).

3.3.4 Variablen- und Attribut-Vereinbarungen

Variablen in Routinen (Funktionen, Werkzeugen (Tools) und Methoden) sind lokal zu diesen Routinen, d.h. von außen nicht sichtbar. Sie sind zur Aufnahme von Werten bestimmt, die bei der Abarbeitung dieser Routinen entstehen bzw. benötigt werden.

Attribute in Aktivitäten (s. Abschnitt 3.5.3 auf Seite 24) sind ebenfalls zu diesem Zweck bestimmt; Attribute in anderen Objekten (Entitäten bzw. Intervallen) dienen primär der Datenspeicherung. Da Objekte referenziert werden können, ist es möglich, von außen auf Attribute anderer Objekte zuzugreifen oder sie zu verändern.

3.3.4.1 Variablen in Routinen

```
Declaration_Part =   VAR { Ident_List ":" Simple_Type
                           [INIT Expression] ";"}+.
Ident_List =         { Ident // "," }+.
```

Variablen in Routinen können Werte der Grundtypen (s. Abschnitt 3.3.3 auf Seite 10) annehmen sowie Referenzen auf Objekte enthalten. Der optionale Initialisierungsausdruck wird unmittelbar nach dem Aufruf einer Routine ausgewertet. Die Auswertungsreihenfolge der Initialisierungsausdrücke ist nicht festgelegt.

3.3.4.2 Attribute in Klassen

Aktivitäten, Entitäten und Intervalle besitzen Attribute zur Speicherung lokaler Daten. In Aktivitäten dienen diese Daten hauptsächlich der Abarbeitung des Aktionsteils. Instanzen von Aktivitäten werden nach ihrer Abarbeitung meistens nicht mehr benötigt; sie müssen jedoch zu Planungs- und Analysezwecken aufbewahrt werden. In den anderen Klassen werden die Daten dauerhaft gespeichert; entweder weil sie keiner speziellen Aktivität zuzuordnen sind oder weil sie zur späteren Auswertung noch benötigt werden.

```
Attribute_Part =          [ PUBLIC ":" { Attribute }+ ]
                          [ PRIVATE ":" { Attribute }+ ].
Attribute =               Declarative_Attribute | Active_Value |
                          Method_Declaration | Link_Attribute.
Declarative_Attribute =   attribute_name ":" Simple_Type
                          [ INIT Init_Expr ] [ MONITOR ] ";".
Active_Value =            attribute_name ":=" Expression [MONITOR]";".
Method_Declaration =      METHOD method_name
                          [ Heading ]
                          Routine_Body
                          END_METHOD method_name ";"
Link_Attribute =          ( Reference_Link | Structural_Link ) [ "!" ].
Reference_Link =          attribute_name "->" (   Single_Link |
                                                  Multiple_Link )";".
Structural_Link =         attribute_name "=" (   Single_Link |
                                                 Multiple_Link )";".
Single_Link =             reference_name [ "*" ].
Multiple_Link =           "(" { reference_name // "," }+ ")" [ "*" ].
Milestone_Links =         MILESTONES ":"
                          { attribute_name "->" milestone_name ";" }+.
```

Im Unterschied zu Variablen in Routinen können Instanzen von Aktivitäten, Entitäten, Intervallen und Meilensteinen referenziert werden (u.U. über mehrere Stufen); daher ist es möglich, von außen auf Attribute einer anderen Klasse bzw. deren Instanzen zuzugreifen. Es gibt einen Public- und einen Private-Teil bei der Attribut-Deklaration.

Attribute, die im Public-Teil beschrieben werden, bilden die Schnittstelle zu der jeweiligen Instanz. Sie können von anderen Objekten gelesen und - außer bei operationalen Attributen - verändert werden.

Attribute, die im Private-Teil deklariert werden, sind für andere Objekte nicht sichtbar. Sie können jedoch von operationalen Attributen des Public-Teils gelesen und von Methoden-Attributen verändert werden. In Aktivitäten können private Attribute natürlich auch durch Anweisungen des Aktionsteils gelesen und verändert werden.

Es stehen folgende Attribut-Arten zur Verfügung:

1. Deklarative Attribute

Deklarative Attribute (Declarative_Attribute) können Werte aus dem Wertebereich der Grunddatentypen (s. Abschnitt 3.3.3 auf Seite 10) aufnehmen. Sie bestehen aus einem Attribut-Namen und eines Attribut-Wert, der vom angegebenen Wert ist.

Bei der Instanziierung einer Entität oder eines Intervalls werden die vorhandenen Initialisierungsausdrücke in beliebiger Reihenfolge ausgewertet und zugewiesen. Die Initialisierungsausdrücke dürfen nicht (lesend) auf andere Attribute zugreifen.

In Aktivitäten werden die Initialisierungsausdrücke unmittelbar vor Beginn der Ausführung einer Aktivität ausgewertet und zugewiesen. Wird eine Aktivität bei der Instanziierung mit Parametern versorgt, werden eventuell vorhandene Initialisierungsausdrücke ignoriert [Verlage, Knauber].

Bei deklarativen Attributen kann zusätzlich das Schlüsselwort MONITOR angegeben werden. Dadurch wird deutlich gemacht, daß das Attribut wesentlich zum Zustand des Projekts beiträgt. Dieser Zustand kann dann zur Laufzeit durch die Standardfunktion MONITOR (s. Abschnitt 3.10 auf Seite 47) abgefragt werden.

Beispiel:

```
ENTITY Module;
   PUBLIC:
      Name    : STRING INIT ReadLn("Modul-Name: ");
      Status  : SYMBOL INIT undefined MONITOR;
   ...
END_ENTITY Module;
```

2. Aktive Werte

Aktive Werte (Active_Value) repräsentieren Daten, auf die nur lesend zugegriffen werden kann. Sie dienen der Berechnung von sich häufig ändernden Werten sowie dem Nur-Lese-Export von Attributen aus dem Private-Teil. Aktive Werte können durch MONITOR gekennzeichnet werden und werden dann ebenso behandelt wie deklarative Attribute.

Beim Zugriff auf Aktive Werte (Active_Values) wird der Wert jedesmal neu berechnet. Es dürfen andere Attribute zur Berechnung des Wertes verwendet werden.

Beispiel:

```
ACTIVITY Make_Program;
   PUBLIC:
      Start-Date         : TIME INIT ACTUAL-DATE();
      Time-of-Work       := ACTUAL-DATE() - Start-Date;
      Get-Program-Name   := Name;
   PRIVATE:
      Name : STRING;
   ...
END_ACTIVITY Make_Program;
```

3. Methoden-Attribute

Methoden-Attribute (Method_Declaration) dienen der Modellierung von Polymorphismus. Im Sinne der Datenkapselung bei der objektorientierten Programmierung können Attribute des Private-Teils von fremden Objekten nur mittels Methoden-Attributen modifiziert werden.

Beim Zugriff auf Methoden-Attribute wird der Wert jedesmal neu berechnet. Es dürfen

andere Attribute zur Berechnung des Wertes verwendet werden.

Es ist nicht unterscheidbar, ob es sich um einen aktiven Wert oder um ein Methoden-Attribut ohne aktuelle Parameter mit Rückgabewert handelt. Methoden-Attribute können jedoch mit Parametern versorgt werden.

Beispiel:

```
ACTIVITY Make_Program;
   PUBLIC:
      METHOD Set-Program-Name ( filename : STRING );
      BEGIN
         Name := filename;
      END_METHOD Set-Program-Name;
   PRIVATE:
      Name : STRING;
   ...
END_ACTIVITY Make_Program;
```

Zur Vererbung von Methoden-Attributen siehe s. Abschnitt 3.5.1 auf Seite 21.

4. Referenz-Attribute

Ein Referenz-Attribut (Reference_Link) enthält einen Verweis auf ein anderes Objekt oder einen Objekt-Tupel. Er besteht aus einem Attribut-Namen und einem Verweis-Typ; der Typ ist hier der Name einer anderen Benutzerklasse, die referenziert werden soll oder ein Tupel von Namen. Referenziert werden können Entitäten, Aktivitäten, Intervalle und Meilensteine. Der Wert des Attributs bleibt solange undefiniert, bis eine entsprechende Wertzuweisung erfolgt.

Mit dem Ausrufezeichen "!" wird eine obligatorische Referenz bezeichnet, d.h. es muß mindestens ein Objekt (-tupel) referenziert werden. Dem Benutzer werden zum Instanziierungszeitpunkt die existierenden Instanzen angeboten; wird davon keine ausgewählt, werden die entsprechenden Objekte neu kreiert. Die Kreierung von obligatorischen Links setzt sich u.U. über mehrere Ebenen hin fort; jedoch wird der Benutzer nur in der obersten Ebene nach Angaben gefragt, der Rest wird "blind" kreiert. Das bedeutet, daß Entitäten unter Verwendung der Initialisierungsausdrücke instanziiert werden, sofern diese vorhanden sind.[1]

Der Mengenkonstruktor "*" zeigt an, daß eine (ungeordnete) Menge von Objekten bzw. Tupeln referenziert werden soll.

Die Kombination eines obligatorischen Links mit dem Mengenkonstruktor, bedeutet, daß mindestens eine einelementige Menge kreiert wird.

Für Aktivitäten ist ein Referenz-Attribut mit dem Namen SUPER vordefiniert. SUPER enthält eine Referenz auf das instanziierende Objekt, falls dieses eine Aktivität ist (d.h. kein

1. Sollen an dieser Stelle "sinnvolle" Referenzen, d.h. Referenzen auf bereits existierende Instanzen eingesetzt werden, so ist ein entsprechender Baum von unten nach oben zu kreieren; dies ist Aufgabe des ProLan-X - Programmierers und kann nicht automatisch vom System übernommen werden.

Wächter, keine Zusicherung und kein Meilenstein), sonst enthält SUPER den Wert NIL. Der ProLan-X-Compiler überprüft, daß an SUPER ist keine Zuweisung erfolgt.

Beispiel:

```
ENTITY Module;
   PUBLIC:
      Des-Doc      -> Design-Document !;
      Programmers -> Programmer-Person *!;
   ...
END_ENTITY Module;
```

5. Mehrfachreferenz-Attribute

Mehrfach-Referenzen (Structural_Link) sind nur für Entitäten zulässig; sie geben an, aus welchen Entitäten eine Entität zusammengesetzt ist .[1] Strukturelle Verweise unterscheiden sich von einfachen Referenzen in ihrer Semantik bzgl. des Kopierens von Objekt-Aggregaten. Durch strukturelle Verweise aggregierte Objekte werden stets tief kopiert, während bei Referenzen lediglich die Verweise kopiert werden (flache Kopie).

Handelt es sich beim aktuellen Objekt um eine Entität, so verbirgt sich unter dem Namen SUPER eine Mehrfach-Referenz, d.h. eine Liste von Entitäten, die einen strukturellen Link auf das aktuelle Objekt besitzen. Die VPM sorgt zur Laufzeit dafür, daß diese Referenzen immer aktuell sind. Ist die Liste leer, enthält das SUPER den Wert NIL. Der ProLan-X-Compiler überprüft, daß der Benutzer keine Zuweisung an SUPER vornimmt.

Beispiel:

```
ENTITY Program;
   ATTRIBUTE:
      Modules      -> Module *;
   ...
END_ENTITY Program;
```

6. Referenz auf Meilensteine

Mit einer Entität als Arbeitsergebnis können mehrere Meilensteine (s. Abschnitt 3.5.7 auf Seite 32) assoziiert sein, die zum Zeitpunkt der Entitäten-Instanziierung ebenfalls instanziiert werden.

1. In Aktivitäten wird die Zusammensetzung aus Subaktivitäten durch den Aktionsteil beschrieben.

3.4 Ausdrücke

Ausdrücke werden gebraucht, um Regeln zur Berechnung von Werten zu beschreiben. Die Bausteine dieser Regeln sind Konstanten und Variablen (Operanden); sie werden durch Operatoren und Funktionen verknüpft und die Ergebnisse der Berechnung können wieder Variablen zugewiesen werden.

3.4.1 Operanden

Operanden werden entweder durch (Wert-) Konstanten beschrieben oder durch Namen. Die Namen unterscheiden sich wieder in zwei Gruppen: konstante Werte (Symbole, Konstanten, Ereignisse) und Variablen oder Attribute (Variablen in Routinen, s. Abschnitt 3.3.4.1 auf Seite 11 und Attribute in Klassen, s. Abschnitt 3.3.4.2 auf Seite 12). Im Fall von Variablen und Attributen (im folgenden einfach als Variable bezeichnet) können qualifizierende Ausdrücke folgen.

```
Qualident =   { field_name [ "[" Qualifier "]" ] // "." }.
Qualifier =   Bool_Expr.
```

1. Stammt eine Variable aus dem Wertebereich eines einfachen Typs, darf sie nicht näher spezifiziert werden.

2. Bezeichnet eine Variable l einen Referenz-Link (s. Abschnitt 4. auf Seite 14) auf ein Objekt, so kann mit l.f das Attribut f innerhalb dieses Objekts bezeichnet werden (f muß im Public-Teil deklariert sein). Besteht der qualifizierte Bezeichner aus n, durch Punkte getrennten Bezeichnern, so bezeichnen die ersten n-1 Bezeichner Referenzen oder sind qualifizierende Klassennamen (s. Abschnitt 3.5.1 auf Seite 21), der n-te den Namen eines Attributs des referierten Objekts. Als Bezeichner sind auch Methoden-Attribute erlaubt.

3. Bezeichnet eine Variable s einen strukturellen Link (s. Abschnitt 5. auf Seite 15), so sind dadurch im allgemeinen mehrere Objekte referenziert. Durch Angabe von s[expr] kann mit Hilfe eines qualifizierenden Bool'schen Ausdrucks *expr* eines oder mehrere dieser Objekte näher bezeichnet werden.

Beispiel:

```
Name
Lines_of_Code("ActModule.MOD")
Module.Designer.Name
Program.Modules[completed IN Status]
```

3.4.2 Operatoren

Es sind vier verschiedene Bindungsstärken von Operatoren definiert: am stärksten bindet der einstellige Operator NOT, gefolgt von der Multiplikation, der Addition und schließlich

den Relationsoperatoren. Diese Bindungen werden durch syntaktische Konstruktionen erreicht. Gleichstarke Operatoren werden von links nach rechts ausgewertet.

Zur besseren Lesbarkeit der Syntaxregeln wurden verschiedene Produktionen für verschiedene Typen von Ausdrücken eingeführt. Diese werden jedoch nicht syntaktisch unterschieden.

```
Const_Expr =    Expression.
Init_Expr =     Expression.
Bool_Expr =     Expression.
```

Konstante Ausdrücke müssen zur Übersetzzeit auswertbar sein (das kann auch während der Durchführung eines Projekts sein!); Bool'sche Ausdrücke müssen einen Wahrheitswert ergeben und werden verkürzt ausgewertet.

Beispiel:

```
   FALSE AND f(x)      -- f(x) wird nicht berechnet
   TRUE OR g(x)        -- g(x) wird nicht berechnet

Expression =          Simple_Expr [ Relop Simple_Expr ].
Relop =               "=" | "<>" | "<" | "<=" | ">=" | ">" | IN.
Simple_Expr =         [ "+" | "-" ] Term { Addop Term }.
Addop =               "+" | "-" | OR.
Term =                Factor { Mulop Factor }.
Mulop =               "*" | "/" | DIV | MOD | AND.
Factor =              Status | Routine_Call | Qualident | NOT Factor |
                      "(" Expression ")" | Type_Conversion | Integer |
                      Real | Boolean | String | Link | Time | Duration |
                      constant_name | event_name.
Status =              "{" { symbol_name // "," } "}".
Type_Conversion =     "(" Simple_Type Expression ")".
```

1. Logische Operatoren

OR	logische Disjunktion
AND	logische Konjunktion
NOT	logische Negation

Diese Operationen werden auf Bool'sche Operanden angewendet und liefern ein Resultat vom Typ BOOLEAN.

2. Arithmetische Operatoren

+	Summe
-	Differenz
*	Produkt
/	rationaler Quotient
DIV	ganzzahliger Quotient
MOD	Rest bei ganzzahliger Division

Diese Operationen sind alle auf numerische Operanden anwendbar (DIV und MOD nur auf ganzzahlige Operanden) und liefern ein numerisches Ergebnis. Ist mindestens einer der

Operanden eine rationale Zahl, so ist das Ergebnis rational (Typ REAL), sonst ganzzahlig (Typ INTEGER), ausgenommen den Divisionsoperator "/", der immer ein rationales Ergebnis liefert. Bei einstelliger Verwendung bezeichnet "-" die Vorzeichenumkehr.

Eine eventuelle Division durch 0 wird zur Laufzeit abgefangen; es wird eine Warnung für den Benutzer erzeugt und ein Defaultwert zurückgegeben.

3. Stringoperatoren

+	Stringkonkatenation

Die Operation wird auf Zeichenketten angewendet und liefert als Resultat wieder eine Zeichenkette.

4. Mengenoperatoren

+	Vereinigung
-	Mengendifferenz
*	Schnittmenge
/	symmetrische Differenz: A / B = (A-B) + (B-A)

Mengenoperatoren sind auf Operanden vom Typ STATUS anwendbar und liefern ein Resultat vom Typ STATUS. Elemente einer Menge sind immer vom Typ SYMBOL.

5. Zeitoperatoren

Die Behandlung von Zeiten im Bereich der Projektplanung ist sehr wichtig, um den zeitlichen Rahmen für verschiedene Aktivitäten abzugrenzen. ProLan-X bietet neben den elementaren Zeitangaben in Ausdrücken außerdem die Behandlung von Zeiten auf einer höheren Ebene (s. Abschnitt 2.3 auf Seite 4, s. Abschnitt 3.5.6 auf Seite 31).

*, /	Skalierung eines Zeitintervalls
+	Verschiebung eines Zeitpunktes
-	Verschiebung eines Zeitpunktes, Differenz zweier Zeitpunke

Ein Zeitintervall (DURATION) kann durch Multiplikation ("*") oder Division ("/") mit/ durch eine ganze oder rationale Zahl vergrößert und verkleinert werden. Ein Zeitpunkt (TIME) kann durch Addition ("+") oder Subtraktion ("-") eines Zeitintervalles nach vorne oder hinten verschoben werden; es ergibt sich dann ein neuer Zeitpunkt. Es ist möglich, die Differenz zweier Zeitpunkte zu bilden; das Ergebnis dieser Operation ist dann ein Zeitintervall.

6. Vergleichsoperatoren

=	gleich
<>	ungleich
<	kleiner
<=	kleiner oder gleich
>=	größer oder gleich
>	größer
IN	Element von

Vergleichsoperatoren geben immer ein Bool'sches Ergebnis zurück.

Der Test auf Gleichheit/Ungleichheit ist anwendbar auf Zahlen, Bool'sche Operanden, Zeichenketten, Symbole (SYMBOL), Mengen (STATUS) sowie Zeitpunkte (TIME) und Zeitintervalle (DURATION).

Die Ordnungsrelationen ("<", "<=", ">=", ">") sind anwendbar auf Zahlen, Zeichenketten (lexikographische Ordnung laut ASCII-Tabelle), Mengen (echte, unechte Teil- oder Obermenge), Zeitpunkte und Zeitintervalle.

Der IN-Operator testet, ob ein Symbol in einer Menge enthalten ist.

7. Typumwandlung

Die Typumwandlung ist nur zwischen folgenden Grundtypen erlaubt:

von	nach
INTEGER	REAL
INTEGER	STRING
REAL	STRING
BOOLEAN	STRING
STRING	INTEGER
STRING	REAL
STRING	BOOLEAN
STRING	SYMBOL
STRING	TIME
STRING	DURATION
SYMBOL	STRING
TIME	STRING
DURATION	STRING

Der Zieltyp und der zu konvertierende Ausdruck werden in runden Klammern zusammengefaßt. Die Zeichenketten, die konvertiert werden sollen müssen entsprechend der Syntax der Zieltypen notiert sein.

Beispiel:

```
str := (STRING 53);
i := (INTEGER "99");
```

Bei einer fehlerhaften Konvertierung wird eine Warnung erzeugt und ein Defaultwert zurückgegeben.

8. Funktions-, Tools- und Methodenaufrufe

Es steht der Name einer Standard-Funktion, einer benutzerdefinierten Funktion oder eines Tools oder der Name einer Methode. Die entsprechende Routine wird mit den aktuellen Parametern versorgt, sofern vorhanden, und ausgeführt. Bei Methodenaufrufen wird die jeweils speziellste Methode, die auf das angegebene Objekt anwendbar ist, ausgeführt.

Beispiele für Ausdrücke:

`4712`	`INTEGER`
`17 + 4.0`	`REAL`
`"Finished" + (STRING date)`	`STRING`
`designed IN { implemented, tested }`	`STATUS`
`designed IN Program.Finished`	`BOOLEAN`
`Module[NOT (tested IN Status)]`	`List of Module`
`{ designed } + implemented`	`STATUS`
`(BOOLEAN "TRUE")`	`BOOLEAN`
`23.12.1991 7:02 < 23.12.1991 8:04`	`BOOLEAN`
`26.8.1991 - 1.10.1985`	`DURATION`
`faculty( 13 )`	`INTEGER`

3.5 Projektbeschreibung

Die Projektbeschreibung ist der Teil, der das Projekt und seine Lösung beschreibt. Dieser Teil ist (durch Verwendung eines einheitlichen Interfaces, s. Abschnitt 3.6 auf Seite 34), unabhängig von dem benutzten Betriebssystem und der Wirtsumgebung.

```
Project_Description =   PROJECT_DESCRIPTION
                        { IMPORT project_name ";" }
                        { Project_Object ";" }.
```

Mit der Import-Anweisung können ganze Projekt-Beschreibungen aus anderen Projekten übernommen werden; daran können sich speziell auf das aktuelle Projekt zugeschnittene Projekt-Klassen anschließen. Die importierten Definitionen können dabei durch nachfolgend importierte oder lokale überlagert werden. In diesem Fall wird eine Warnung an den Benutzer ausgegeben.

```
Project_Object =  Symbol_Declaration | Constant_Declaration |
                  Event_Declaration | Entity | Activity | Assertion |
                  Milestone | Activity_Guard | Interval.
```

Symbole, Konstanten (s. Abschnitt 3.3.1 auf Seite 9) und Ereignisse (Events, s. Abschnitt 3.3.2 auf Seite 10) wurden bereits beschrieben.

Die im folgenden definierten Projektobjekte sind jeweils als benutzerdefinierte Klasse zu sehen. Z.B. kann der Begriff Aktivität sowohl die Klasse als auch eine Instanz der entsprechenden Klasse bezeichnen. Sofern es aus dem Zusammenhang eindeutig hervorgeht, wird nicht explizit zwischen der Klasse des Objekts und dem Objekt selbst unterschieden.

3.5.1 Vererbung

```
Inheritance =  INHERIT FROM { reference_name // "," }+ ";".
```

Von den Prozeßbeschreibungen werden Exemplare (Prozesse) gebildet, die als Grundlage für die Ausführung dienen. Man kann sagen, daß die Beschreibungen die Typen der Prozesse sind. Verfolgt man die Evolution der Beschreibungen über mehrere Projekte, so wird man feststellen, daß 1. Beschreibungen modifiziert werden und 2. Beschreibungen Gemeinsamkeiten teilen. Aus diesen Gründen wurde das Klassenkonzept eingeführt. Man ist somit in der Lage, inkrementell Programmbeschreibungen zu entwickeln und abstrakte Klassen zu bilden um Redundanzen zu vermeiden.

Entitäten dürfen nur von Entitäten erben, Aktivitäten nur von Aktivitäten.
Die Klassen erben die aufgelisteten Definitionen von ihrer Vaterklasse:

1. Entitäten: Attribute, d.h. deren Namen, Typ, Initialisierungsausdruck und Zugriffsberechtigung (Private oder Public), sowie Referenzen auf Meilensteine, d.h. deren Namen und Typ;

2. Aktivitäten: Attribute, d.h. deren Namen, Typ, Initialisierungsausdruck und

Zugriffsberechtigung (Private oder Public), sowie
Referenzen auf Intervalle, d.h. deren Namen und Typ;

Ererbte Definitionen können in der aktuellen Klasse nicht überlagert werden; auch die Zugehörigkeit von Attributen zum Public- oder Private-Teil kann nicht verändert werden. Eine Ausnahme bilden Methoden-Attribute (s. Abschnitt 3. auf Seite 13): Methoden-Attribute dürfen auf die aktuelle Klasse spezialisiert werden, auch wenn die aktuelle Klasse bereits ein Methoden-Attribut gleichen Namens erbt. Zur Laufzeit wird die speziellste Methode ausgewählt, die auf das aktuelle Objekt und eventuell vorhandene zusätzliche Parameter anwendbar ist.

Beispiel:

```
ENTITY EmptyNumber;
   PUBLIC:
      METHOD Print; -- Default-Methode
         Write( "###" );
      END_METHOD Print;
END_ENTITY EmptyNumber;

ENTITY ComplexNumber;
   PUBLIC:
      METHOD Print;
         Write( real, "+ i", im );
      END_METHOD Print;
   PRIVATE:
      real   : REAL;
      im     : REAL INIT 0;
END_ENTITY ComplexNumber;
```

Bei der Verarbeitung von Attributen ist der Zeitpunkt der Definition (z.B. durch Ergänzung) von Attributen in Superklassen ohne Bedeutung. Werden jedoch konkrete Initialisierungswerte vererbt, so ändern sich diese nur in neuen, nicht aber in bereits existierenden Instanzen.

Für die Vererbung gilt folgende Regel:
Jede Klasse hat Priorität über ihre Superklassen.

Da Mehrfach-Vererbung erlaubt ist, ist zusätzlich folgendes zu beachten:

Es ist möglich, daß eine Klasse ein Attribut gleichen Namens von mehreren Superklassen erbt. In diesem Fall muß in der aktuellen Klasse (sowie deren Subklassen) qualifizierend auf dieses Attribut zugegriffen werden. Als Qualifikator wird der Name der direkten Superklasse benutzt.

Beispiel:

```
ENTITY K;
   PUBLIC: a : INTEGER;
END_ENTITY K;

ENTITY K1;      INHERIT FROM K;
END_ENTITY K1;
```

```
ENTITY K2;
   PUBLIC: a : CHAR;
END_ENTITY K2;

ENTITY K3;      INHERIT FROM K1, K2;
   -- impliziert die Attribute
   -- K1.a : INTEGER;
   -- K2.a : CHAR;
END_ENTITY K3;
```

Besteht eine Referenz auf ein Objekt einer Superklasse, das aktuell referenzierte Objekt ist jedoch eine Instanz einer Subklasse (s. Abschnitt 1. auf Seite 36), wird automatisch das Attribut referenziert, das der ursprünglich referenzierten Klasse entspricht.

Beispiel:

```
ACTIVITY A;
   PUBLIC:
      ref -> K1;
   EXCUTE:
      ref := CREATE( K3 );
      -- es bezeichnet ref.a das Attribut vom Typ INTEGER:
      ref.a := 5;
   END;
END_ACTIVITY A;
```

Methoden-Attribute dürfen nicht durch einen Klassennamen qualifiziert werden!

3.5.2 Entitäten

```
Entity =        ENTITY entity_name ";"
                Entity_Body
                END_ENTITY entity_name.
Entity_Body =   [ Inheritance ]
                [ Attribute_Part ]
                [ Milestone_Links ].
```

Eine Entität beschreibt die Struktur eines Datenobjekts. Sie besteht aus verschiedenen Attributen, die verschiedene Datentypen oder Referenzen auf andere Objekte (nicht nur Entitäten) beinhalten können. Entitäten können von anderen Entitäten Attribute erben und bei Bedarf an Meilensteine gebunden werden. Alle Entitätenklassen erben von der Standardklasse ENTITY.

Der Name am Anfang und am Ende der Deklaration muß übereinstimmen; er bezeichnet den Namen der Entitätenklasse.

Entitäten werden explizit mit CREATE (s. Abschnitt 3.10.3 auf Seite 48) kreiert oder bei der Kreierung eines Objekts, das eine Referenz auf ein solches Objekt verlangt, miterzeugt. Sie bleiben bestehen, auch wenn keine Referenz mehr darauf verweist bis sie explizit durch DELETE bzw. DELETE-COMPLETE (s. Abschnitt 3.10.3 auf Seite 48) gelöscht werden.

Beispiel:

```
ENTITY Program;
   PUBLIC:
      Name     : STRING;
      METHOD AddModule( new : Module );
         INCLUDE( new, Modules );
      END_METHOD AddModule;
   PRIVATE:
      Modules = Module;
      Date-of-Creation : TIME INIT ACT-TIME();
END_ENTITY Program;

...

   link := CREATE( Program );
   DELETE-COMPLETE( link );
```

3.5.3 Aktivitäten

```
Activity =            ACTIVITY activity_name ";"
                      Activity_Body
                      END_ACTIVITY activity_name.
Activity_Body =       [ Inheritance ]
                      [ Super_Activities ]
                      [ Attribute_Part ]
                        Interval_Link
                      [ Validation ]
                      [ Precondition ]
                      [ Action_Sequence ]
                      [ Postcondition ].
Super_Activities =    SUPER_ACTIVITIES ":" {activity_name // ","}+ ";".
Interval_Link =       INTERVAL ":" attribute_name
                      [ "->" interval_name ] ";".
```

Aktivitäten beschreiben die Zustands-Übergänge des Projekt-Modells. Das können Veränderungen an den Objekten oder an den Benutzerklassen sein. Alle Aktivitätenklassen erben von der Standardklasse ACTIVITY.

Die Super-Aktivitäten beschreiben die möglichen Aktivitäten (-klassen), welche die aktuelle Aktivität instanziieren dürfen. Das können mehrere sein; zur Laufzeit ist es jedoch immer eine konkrete Aktivität (aus der Menge von Klassen der angegebenen Super-Aktivitäten), die die Instanziierung auslöst. Aktivitäten können auch durch Wächter (s. Abschnitt 3.5.5 auf Seite 30), Zusicherungen (s. Abschnitt 3.5.4 auf Seite 30) oder Meilensteine (s. Abschnitt 3.5.7 auf Seite 32) instanziiert werden. In diesem Fall ist das Standard-Attribut SUPER mit NIL initialisiert, ansonsten kann die aktuelle Super-Aktivität über SUPER referenziert werden [Knauber].

Aktivitäten können Attribute (s. Abschnitt 3.3.4.2 auf Seite 12) besitzen, die die notwendigen Daten zur Abarbeitung des Aktionsteils zur Verfügung stellen.

Mit einem zugeordneten Intervall (s. Abschnitt 3.5.6 auf Seite 31) wird der zeitliche Rahmen für den Ablauf einer Aktivität bestimmt bzw. protokolliert. Defaultwert für den Typ des zugeordneten Intervalls ist das leere Intervall.

Der Name am Anfang und am Ende der Deklaration muß übereinstimmen; er bezeichnet den Namen der Aktivitätenklasse.

3.5.3.1 Gültigkeit von Aktivitäten

```
Validation =      VALID ":" Bool_Expr ";".
Precondition =    PRECOND ":" Bool_Expr ";".
Postcondition =   POSTCOND ":" Bool_Expr ";".
```

In ProLan-X ist es möglich, das Verhalten von Aktivitäten durch Vor- und Nachbedingungen zu beschreiben.

Die Vorbedingung (Precondition) wird vor der Ausführung einer Aktivität überprüft. Sie muß den Bool'schen Wert *wahr* liefern, damit die Aktivität gestartet werden kann. Defaultwert für die Vorbedingung ist *wahr.*

Die Nachbedingung (Postcondition) ist als Kommentar für den Benutzer gedacht. Sie beschreibt den angestrebten Folgezustand, der durch die Aktivität erreicht werden soll. Da das System nicht garantieren kann, daß der Aktionsteil der Aktivität Aktionen enthält, die unter allen Umständen dazu führen, daß die Nachbedingung gilt, wird die Nachbedingung nach Ausführung der Aktivität überprüft und eine Warnung erzeugt, falls sie den Wert *falsch* liefert. Defaultwert für die Nachbedingung ist *wahr.*

In einer Kaskade (s. Abschnitt 8. auf Seite 28) erhält die Nachbedingung eine besondere Bedeutung.

Es ist möglich, daß die Durchführung einer bereits instanziierten Aktivität keinen Sinn (mehr) ergibt. Es muß daher möglich sein, ein solches Aktivitätenobjekt wieder zu löschen, solange seine Ausführung noch nicht begonnen wurde.
Die Valid-Form gibt die Gültigkeit einer Aktivität an. Liefert der angegebene Ausdruck für eine Instanz *falsch,* so wird das Objekt wieder vom Plan genommen. Defaultwert für die Valid-Form ist *wahr.*

Beispiel:

```
ACTIVITY Edit_Module;
   PUBLIC:
      mod     -> Module;
      Editor  : STRING INIT "vi";
   PRIVATE:
      name    := mod.Name;
      state   := mod.Status;
   PRECOND: designed IN state;
   VALID: mod <> NIL;
   ...
```

```
    POSTCOND: edited IN state;
END_ACTIVITY Edit_Module;
```

3.5.3.2 Anweisungsteil einer Aktivität

Der Anweisungsteil einer Aktivität beschreibt die Modifikationen, die die Objekte einer Klasse von Aktivitäten am Projektzustand vornimmt. Es handelt sich um eine sequentielle Auflistung von Aktionen.

```
Action_Sequence =   { Action ";" }+.
Action =            Subactivity_Call | Sequence | Set | Selection |
                    Loop | Exit | M_Set | Cascade |
                    Conditional_Action | Elementary_Action.
```

Die einzelnen Aktionen spezifizieren auf der momentanen Abstraktionsebene der Projektbeschreibung einzelne Schritte. Die möglichen Arten von Aktionen werden im folgenden beschrieben.

Die Aktionsfolge (Action_Sequence) kann zur Übersetzungszeit leer bleiben. Damit wird angezeigt, daß die Notwendigkeit zur Berücksichtigung dieser Aktivität bereits bekannt ist, daß die Aktivität aber erst später (zur Laufzeit) genauer spezifiziert werden soll bzw. kann. Bevor eine solche Aktivität zur Ausführung gebracht werden kann, erfolgt ein entsprechender Hinweis an den Benutzer, der die Aktionsfolge nun konkretisieren muß.

1. Subaktivitäts-Aufruf

Der Aufruf einer Subaktivität bewirkt die Instanziierung eines neuen Objekts der Klasse ACTIVITY. Vergleichbar mit einer Funktion wird die Bearbeitung eines Projektschrittes in einem feineren Aktionsgranulat dargestellt. Beim Aufruf einer Aktivität können Daten mittels der aktuellen Parameter an Attribute der Subaktivitäten weitergegeben werden. Die Zuweisung erfolgt qualifizierend.

```
Subactivity_Call =      activity_name [ Instantiation_List ].
Instantiation_List =    "(" { Instantiation_Expression // "," }+ ")".
Instantiation_Expression = Ident "=" Expression.
```

Der Attribut-Name muß im neu zu schaffenden Objekt existieren, das Attribut muß im Public-Teil (s. Abschnitt 3.3.4.2 auf Seite 12) deklariert sein und der Ausdruck muß zuweisungskompatibel zum angegebenen Attribut-Typ sein.

Wird der Wert eines formalen Parameters (Attributs) nicht spezifiziert, so wird der betreffende Initialwert verwendet (s. Abschnitt 3.3.4.2 auf Seite 12).

Beispiel:

```
ACTIVITY Translate_Module;
   PRIVATE:
      actModule -> Module;
   ...
   Edit_Module( mod = actModule );
```

```
    -- s.o.: 'mod' wird durch den Parameter gesetzt;
    --       'Editor' mit dem INIT-Wert versorgt
    Compile_Module( actModule.SourceName );
    ...
END_ACTIVITY Translate_Module;
```

2. Folge von Aktionen

```
Sequence = SEQUENCE Action_Sequence END.
```

Das Schlüsselwort SEQUENCE leitet eine Reihe von Aktionen ein, die streng sequentiell abgearbeitet werden müssen. Erst nachdem die vorherige Aktion innerhalb der Aktionsfolge bearbeitet wurde, kann mit der Auswertung der nachfolgenden Aktion begonnen werden.

3. Menge von Aktionen

```
Set = SET Action_Sequence END.
```

Das Schlüsselwort SET kennzeichnet eine Menge von Aktionen, deren Ausführungsreihenfolge keinerlei Beschränkung unterworfen ist. Es ist hier eine echt parallele Abarbeitung erlaubt.

4. Auswahl von Subaktivitäten

```
Selection = SELECTION { Subactivity_Call ";" }+ END.
```

Eine Auswahl bezeichnet alternative Abarbeitungen von Aktivitäten. Der Benutzer kann unter den angegebenen Subaktivitäten eine auswählen, die ausgeführt werden soll. Diese Auswahl muß spätestens dann erfolgen, wenn die Auswertung einer Vorbedingung der Subaktivitäten erfolgen könnte. Als Repräsentant für diese Aktivität existiert ein Objekt der allgemeinen Klasse ACTIVITY, welches nach der Auswahl durch den Benutzer in ein entsprechendes Objekt der betreffenden Klasse umgewandelt wird.
Eine direkte Auswertung einer Auswahl, d.h. ohne den Benutzer, ist nicht möglich.

5. Wiederholte Folge von Aktionen

```
Loop = LOOP Action_Sequence END.
```

Das Schlüsselwort LOOP kennzeichnet den Beginn einer Schleife. Gelangt die Auswertung an das abschließende END, so wird sofort an den Anfang der Schleife zurückgesprungen, d.h. die Auswertung setzt mit der ersten Aktion nach LOOP fort. Eine Schleife wird durch eine Exit-Aktion innerhalb der eingeschachtelten Aktionen verlassen.

6. Exit-Aktion

```
Exit = EXIT.
```

Mit der Ausführung einer Exit-Anweisung wird die direkt umgebende Schleife verlassen. Die Interpretation fährt mit der ersten Anweisung nach dem abschließenden END fort. Es besteht also nicht die Möglichkeit mehrere geschachtelte Schleifen simultan zu verlassen. Die Auswertung einer Exit-Anweisung ohne eine umschließende Schleife führt zu einem Laufzeitfehler.

Beispiel:

```
ACTIVITY Translate_Module;
   PUBLIC:
      mod -> Module;
   LOOP
      Edit_Module(...);
      Compile_Module(...);
      IF compiled IN mod.Status THEN EXIT END;
   END;
END_ACTIVITY Translate_Module;
```

7. Multiple Menge von Aktionen

```
M_Set =              M_SET Cardinality_Expr ":" Action_Sequence END.
Cardinality_Expr =   [ Ident ] OVER Qualident.
```

Im Gegensatz zur statischen Menge von Aktionen (s. Abschnitt 3. auf Seite 27) ist die Mächtigkeit der multiplen Menge, d.h. die Anzahl der Subaktionen nicht statisch berechenbar. Über den Kardinalitätsausdruck wird zum Zeitpunkt der Ausführung der M_Set-Anweisung die Mächtigkeit der Menge berechnet und die entsprechende Anzahl von Subaktivitäten instanziiert. Der Qualident bezeichnet entweder eine (Benutzer-) Klasse von Objekten oder eine Referenz, die (sinnvollerweise) eine Menge von Objekten bezeichnen kann.

Mit Ident wird eine lokale Variable definiert, über die in der Aktionsfolge auf Elemente der ausgewählten Menge zugegriffen werden kann. Ihr Name muß im aktuellen Kontext eindeutig sein und ihr Gültigkeitsbereich ist auf die Schleife beschränkt.

Beispiel:

```
ACTIVITY Translate_Program;
   PRIVATE:
      modules = Modules;
   M_SET actModule OVER modules:
      Translate_Module( mod = actModule );
   END;
   Link_Modules( modules=modules );
END_ACTIVITY Translate_Program;
```

8. Kaskade

```
Cascade =   CASCADE { Subactivity_Call
            ["=>" {activity_name // ","}+] ";" } END.
```

Die Kaskade ist eine Mischung zwischen einer Folge von Aktivitäten und einer Schleife. Generell werden die angegebenen Aktivitäten (nicht Aktionen!) sequentiell abgearbeitet. Wird jedoch eine Nachbedingung zu *falsch* ausgewertet, so erfolgt ein rückwärtiger Sprung zu einer der in der Liste angegebenen Aktivitäten. Bei mehreren spezifizierten Aktivitäten muß der Benutzer ein Sprungziel auswählen. (Nur) Sprungzielnamen müssen in der Kaskade eindeutig sein.

Beispiel:

```
ACTIVITY Perform_Project;
   ...
   CASCADE
      System_Analysis( ... );
      Design( ... )          => System_Analysis;
      Implementation( ... )  => Design;
      Test( ... )            => System_Analysis, Design;
      -- tritt ein Fehler auf, ist er in der Design- oder
      -- Analysephase entstanden
   END;
   ...
END_ACTIVITY Perform_Project;
```

9. Bedingte Aktivitäten

```
Conditional_Action =  IF Bool_Expr THEN Action_Sequence
                      { ELSIF Bool_Expr THEN Action_Sequence }
                      [ ELSE Action_Sequence ] END.
```

Die Bool'schen Ausdrücke werden in der Reihenfolge ihres Auftretens ausgewertet, bis einer den Wert *wahr* ergibt. Dann wird die mit dem Ausdruck assoziierte Aktionsfolge ausgeführt. Die Aktionsfolge des Else-Zweigs wird, sofern spezifiziert, dann ausgeführt, wenn alle Bool'schen Ausdrücke zu *falsch* ausgewertet wurden.

10. Elementare Aktionen

```
Elementary_Action =  EXECUTE Statement_Sequence END.
```

Elementare Aktionen enthalten eine Folge von Anweisungen, die Werte ändern, um das Ziel der Aktivität, die Überführung in einen Folgezustand und damit die Erfüllung der Postcondition, zu erreichen.

Als elementare Aktionen können alle Anweisungen von Routinen (s. Abschnitt 3.6.2 auf Seite 35) verwendet werden außer der CALL-Anweisung. Dies ist semantisch zu überprüfen.

Im Unterschied zu anderen Kombinationen von Aktionen (z.B. Folge von Aktionen, Sequence) können elementare Aktionen nicht unterbrochen werden; Ausnahme bei **TRANSFER** (s. Abschnitt 3.6.1 auf Seite 34).

Beispiel:

```
ACTIVITY Edit_Module;
   ...
   EXECUTE
      Edit( mod.SourceName );
      IF compiled IN mod.Status THEN
         EXCLUDE( compiled, mod.Status );
      END;
      INCLUDE( edited, mod.Status );
   END;
END_ACTIVITY Edit_Module;
```

3.5.4 Zusicherungen

```
Assertion =              ASSERTION assertion_name ";"
                         Assertion_Body
                         END_ASSERTION assertion_name.
Assertion_Body =         UNLESS Constraint DO Exception_Handling.
Constraint =             Bool_Expr.
Exception_Handling =     [ Action_Sequence ].
```

Durch Zusicherungen ist der Benutzer in der Lage, globale Bedingungen für ein Projekt anzugeben. Alle Zusicherungen zusammen beschreiben den Zustand, der garantiert werden soll. Die Bedingungen (Constraints) werden vor jeder Ausführung einer (Teil-) Aktivität überprüft. Da während einer elementaren Aktion kurzfristig Inkonsistenzen auftreten können, werden die Bedingungen nicht ständig getestet.

Die Ausnahme-Behandlung (Exception_Handling) beschreibt die Aktionen, die ausgeführt werden sollen, wenn die Bedingung verletzt wird. Sie kann ebenso wie die Aktionsfolge von Aktivitäten (s. Abschnitt 3.5.3.2 auf Seite 26) zunächst unspezifiziert bleiben.

Von Zusicherungen wird genau eine Instanz kreiert; dies geschieht automatisch beim Start eines übersetzten Projektes. Diese Instanz kann durch DELETE (s. Abschnitt 3.10.3 auf Seite 48) gelöscht werden.

Der Name am Anfang und am Ende der Deklaration muß übereinstimmen; er bezeichnet den Namen der Zusicherungsklasse.

Beispiel:

```
ASSERTION Limit_untested_Modules;
   UNLESS NUMBER-OF( Module[NOT (tested IN Status)] ) < 10 DO
   EXECUTE
      Warning( "More than 10 untested Modules!" );
   END;
END_ASSERTION Limit_untested_Modules;
```

3.5.5 Wächter

```
Activity_Guard =   GUARD guard_name ";"
                   Guard_Body
                   END_GUARD guard_name.
Guard_Body =       ON Guard
                   [ IF Bool_Expr ]
                   DO Subactivity_Call.
Guard =            Routine_Call.
```

Ein Wächter reagiert auf bestimmte Änderungen am Zustand des Systems und löst in Abhängigkeit vom Typ des Ereignisses (Events) bestimmte Aktionen aus, die in einer Aktivität zusammengefaßt werden.

Die Wächter-Funktion (Guard) beschreibt das Ereignis, auf das hin die angegebene

Aktivität ausgeführt wird. Es dürfen nur Trigger-Funktionen (s. Abschnitt 3.10.2 auf Seite 47) verwendet werden. Werden nicht die Trigger-Funktionen USER-EVENT oder TIME-EVENT verwendet, bezeichnet der Name ACTIVATOR innerhalb eines Wächters eine vordefinierte Variable von Typ Referenz-Link. Diese Variable ist eine Referenz auf das Objekt, das für das Auslösen des Events verantwortlich ist. So können identifizierende Attribute dieses Objekts (oder das gesamte Objekt) in der Instanziierungs-Liste an die zu startende Aktivität übergeben werden.

Die If-Klausel schränkt die Gültigkeit des Wächters ein.

Der Name am Anfang und am Ende der Deklaration muß übereinstimmen; er bezeichnet den Namen der Wächterklasse.

Beispiel:

```
GUARD Start_Project;
   ON CREATE-EVENT( Project-Object )
   DO Perform_Project( description = ACTIVATOR );
END_GUARD Start_Project;
```

3.5.6 Intervalle

```
Interval =              INTERVAL interval_name ";"
                        Interval_Body
                        END_INTERVAL interval_name ";".
Interval_Body =         [ EARLIEST_START ":" Expression ";" ]
                        [ LATEST_START ":" Expression ";" ]
                        [ EARLIEST_END ":" Expression ";" ]
                        [ LATEST_END ":" Expression ";" ]
                        [ MIN_DURATION ":" Expression ";" ]
                        [ MAX_DURATION ":" Expression ";" ]
                        { Temporal_Attribute ";" }.
Temporal_Attribute =    attribute_name ":" Temporal_Type
                        [ INIT Init_Expr ].
```

Die Ausdrücke (Earliest_Start bis Latest_End), die Zeitpunkte spezifizieren, müssen vom Typ TIME (s. Abschnitt 7. auf Seite 11) sein. Die Dauer-Ausdrücke (Min_Duration und Max_Duration) müssen vom Typ DURATION (s. Abschnitt 8. auf Seite 11) sein. Der Benutzer kann den vordefinierten Attributen eigene hinzufügen.

Die Kombination qualitativen und quantitativen temporalen Schließens wurde durch die Arbeit von R. Bleisinger (s. [Bleisinger]) möglich. Die Umsetzung dieser Konzepte wurde in Form einer TEMPO (Temporal Reasoner) genannten Komponente an das MoMo-System angebunden. Der Temporal Reasoner überprüft die Konsistenz der temporalen Angaben.

Qualitative Angaben (Allen-Relationen (s. [Allen]) und Dauer-Relationen) können nicht auf Klassenebene spezifiziert werden. Sie werden erst zur Laufzeit für Intervall-Instanzen angegeben.

In der Regel hat es sich nicht als praktisch herausgestellt, temporale Angaben auf Klassenebene anzugeben, da diese erst zur Ausführungszeit, d.h. auf Objektebene, (tatsächlich) quantifiziert werden können.

Spezifizierter Typ und Typ des Initialisierungs-Ausdrucks müssen übereinstimmen.

Der Name am Anfang und am Ende der Deklaration muß übereinstimmen; er bezeichnet den Namen der Intervallklasse.

Beispiel:

```
INTERVAL Module_Implementation_Int;
   EARLIEST_END:      01.01.1991;
   LATEST_END:        24.06.1991;
   MIN_DURATION:      2 m; -- 2 Monate
   MAX_DURATION:      5 m; -- 5 Monate
END_INTERVAL Module_Implementation_Int;

ACTIVITY Module_Implementation;
   ...
   INTERVAL: Time_Constraints -> Module_Implementation_Int;
   ...
END_ACTIVITY Module_Implementation;
```

3.5.7 Meilensteine

```
Milestone =          MILESTONE milestone_name ";"
                     Milestone_Body
                     END_MILESTONE milestone_name.
Milestone_Body =     [ Date ]
                     [ Condition ]
                     [ Action_Sequence ].
Date =               DATE ":" Expression ";".
Condition =          ON Bool_Expr DO.
```

Ein Meilenstein kennzeichnet einen Zeitpunkt, zu welchem bestimmte Arbeitsergebnisse vorliegen sollen. Da Arbeitsergebnisse als Entitäten modelliert werden, sind Meilensteine deshalb immer mit Entitäten verknüpft (s. Abschnitt 3.5.2 auf Seite 23) und werden zusammen mit den Entitäten instanziiert.

In Abhängigkeit vom Wert des Bool'schen Ausdrucks der Bedingung (Condition) werden die angegebenen Aktionen veranlaßt. Die Condition muß in jedem Fall spezifiziert werden, sei es indirekt durch Vererbung oder direkt. Über den Standard-Bezeichner RESULT kann immer die zugehörige Entität als Arbeitsergebnis referenziert werden. Die Action_Sequence kann ebenso wie bei Aktivitäten (s. Abschnitt 3.5.3.2 auf Seite 26) zunächst unspezifiziert bleiben.

DATE (als vordefinierter Bezeichner vom Typ TIME (s. Abschnitt 7. auf Seite 11) spezifiziert den Termin, zu dem der Meilenstein erreicht sein soll. Für die Bearbeitung von Meilensteinen kann DATE in der Meilenstein-Bedingung verwendet werden. Meilensteine

sind in der Regel mit Wächtern vergleichbar. Sobald die Meilenstein-Bedingung (Constraint) erfüllt ist (z.B. der angegebene Termin (Date) überschritten ist), wird die spezifizierte Aktionsfolge ausgeführt (z.B. Meldung wegen Terminüberschreitung an den Projektleiter).

Die mit dem Meilenstein assoziierten Aktionen werden dann ausgeführt, wenn der angegebene Bool'sche Ausdruck den Wert *wahr* liefert.

Der Name am Anfang und am Ende der Deklaration muß übereinstimmen; er bezeichnet den Namen der Meilensteinklasse.

Beispiel:

```
MILESTONE Design-MS;
   DATE: 24.6.1991;
   ON (ACTUAL-DATE() > DATE) AND (RESULT.status # done) DO
         Warning( "Time-Constraint violated for ", RESULT.name );
END_MILESTONE Design-MS;

ENTITY Design-Document;
   PUBLIC:
      name              : STRING INIT "";
      status            : SYMBOL INIT undefines;
   MILESTONES:
      Termination-Constraint-> Design-MS;
END_ENTITY Design-Document;
```

3.6 Schnittstelle zum System

```
System_Interface =    SYSTEM_INTERFACE
                      { IMPORT project_name ";" }
                      { Function_Declaration | Tool_Declaration }.
```

Um die eigentliche Projektbeschreibung von der jeweiligen (System-) Umgebung zu entkoppeln, gibt es Funktionen und Tools (im folgenden zusammenfassend Routinen genannt), die in der Schnittstellenbeschreibung (System_Interface) beschrieben werden. Damit ist es möglich, eine Projektbeschreibung nur durch den Austausch der Schnittstelle in eine andere Umgebung zu portieren.

Es ist möglich, Bibliotheken mit diversen Routinen zu erstellen und diese dann mit der Import-Anweisung in die entsprechenden Projekte zu importieren. So kommt man einfach zu Standard-Schnittstellen, an die sich dann projektspezifische Routinen anschließen können. Importierte Routinen können dabei von nachfolgend importierten oder lokalen überlagert, d.h. verdeckt werden, wenn diese dieselben Typen für die formalen Parameter besitzen. In diesem Fall wird bei der Übersetzung eine Warnung an den Benutzer ausgegeben. Ansonsten sind auch diese Routinen polymorph (z.B. Funktion Print, s. Abschnitt 8. auf Seite 37).

Der Anweisungsteil von Funtionen, Tools und Methoden-Attributen (s. Abschnitt 3. auf Seite 13) stimmt überein. In Abschnitt 3.6.2 wird deshalb keine weitere Unterscheidung zwischen den verschiedenen Routinen-Arten getroffen, falls nicht notwendig.

3.6.1 Vereinbarungen von Funktionen und Tools

```
Function_Declaration =  FUNCTION function_name
                        [ Heading ]
                        Routine_Body
                        END_FUNCTION function_name ";".
Tool_Declaration =      TOOL tool_name
                        [ Heading ]
                        Routine_Body
                        END_TOOL tool_name ";".
Heading =               ( WAIT | TRANSFER |  NO_WAIT )
                        [ Formal_Parameter_List ]
                        [(":" Simple_Type)|("->" reference_name)]";".
Formal_Parameter_List =
          "(" { [VAR] Ident_List
                (":" Simple_Type) | ("->" reference_name) // ";" } ")".
```

Eine Routinen-Vereinbarung besteht aus einem Routinen-Kopf und einem Routinen-Rumpf. Der Kopf spezifiziert den Namen der Routine, die formalen Parameter sowie ein Kennzeichen bzgl. der Parallelität der Routine. Der Rumpf spezifiziert die Anweisungen, die beim Aufruf der Routine ausgeführt werden. Der Name am Anfang und am Ende der Vereinbarung von Routinen muß übereinstimmen.

Routinen können bei Bedarf parallel zum laufenden Projekt ausgeführt werden; es ist z.B.

nicht sinnvoll, mit allen Projektaktivitäten zu warten, bis ein Editiervorgang ausgeführt wurde. Es muß aber möglich sein, auf einen Testlauf zu warten, um zu entscheiden, ob ein Programm korrekte Ergebnisse liefert. Diese Steuerung wird durch drei verschiedene Schlüsselwörter in der Kopfzeile ausgedruckt:

1. Das Schlüsselwort WAIT drückt aus, daß auf die Ausführung dieser Routine gewartet werden soll.

2. Mit TRANSFER wird ein paralleler Prozeß erzeugt. Die nachfolgenden Anweisungen werden frühestens dann ausgeführt, wenn sich der Parallelprozeß beim Laufzeitsystem zurückmeldet; inzwischen können aber andere (parallele) Aktivitäten zum Laufen kommen.

3. Mit NO_WAIT wird ein (quasi-) paralleler Prozeß erzeugt; nach seiner Beendigung erfolgt keine Rückmeldung beim Laufzeitsystem.

Formale Parameter von Routinen sind nur innerhalb der Routine bekannt; sie werden beim Aufruf der Routine mit aktuellen Parametern versorgt. Es gibt Wert- und Referenzparameter; Referenzparameter werden durch das Schlüsselwort VAR gekennzeichnet. Sie erhalten beim Aufruf einen Zeiger auf den Speicherplatz des aktuellen Parameters, der eine Variable oder ein Attribut sein muß. Wertparameter werden wie lokale Variablen behandelt (s. Abschnitt 3.3.4.1 auf Seite 11), denen der Wert des aktuellen Parameters als Initialisierungsausdruck zugewiesen wurde. Eine Ausnahme bilden Referenzattribute: wird ein Referenzattribut an einen Wertparameter zugewiesen, so zeigen beide auf das gleiche Objekt; nur der Zeiger (Link) wird kopiert. Soll des Objekt kopiert werden, muß dazu die Standard-Funktion COPY (s. Abschnitt 3.10.3 auf Seite 48) verwendet werden. Wird ein Referenzattribut an eine Referenzparameter zugewiesen, erhält der Parameter einen Zeiger auf den Speicherplatz des Referenzattributs. Die Zuweisungsverträglichkeit von Parametern ist analog zur Wertzuweisung (s. Abschnitt 1. auf Seite 36) geregelt.

Routinen können einen Wert oder eine Referenz zurückliefern; der Typ des Resultats wird dann in der Kopfzeile angegeben. Wird kein Ergebnis-Typ angegeben, so handelt es sich um eine echte Prozedur; es wird kein Ergebnis zurückgegeben.

3.6.2 Anweisungen in Routinen

```
Routine_Body =          [ Declaration_Part ]
                        BEGIN Statement_Sequence.
Statement_Sequence =    { Statement ";" }+.
Statement =             Assignment | Routine_Call |
                        Conditional_Statement | Loop_Statement |
                        Exit_Statement | Iteration_Statement |
                        Return_Statement | System_Call.
```

In Routinen ist nur die Vereinbarung (lokaler) Variablen möglich (s. Abschnitt 3.3.4.1 auf Seite 11). Der Rumpf besteht aus einer Folge von Anweisungen, die auch (in elementaren Aktionen) im Aktionsteil von Aktivitäten verwendet werden können.

1. Wertzuweisung

```
Assignment =   Qualident ":=" Expression.
```

Die Zuweisung von Werten eines Grundtyps (s. Abschnitt 3.3.3 auf Seite 10) ist nur an Variable des gleichen Typs möglich. Ausnahmen:

1. Zeichenketten beliebiger Länge dürfen an String-Variable zugewiesen werden.
2. Ganze Zahlen dürfen an Variablen für rationale Zahlen zugewiesen werden.

Referenzen auf Objekte dürfen an Referenz-Attribute, -Variable und -Parameter zugewiesen werden, wenn diese als Referenzen auf Objekte der gleichen Klasse oder auf Objekte einer Super-Klasse deklariert wurden (vgl. Oberon [Mössenböck, Wirth]).

2. Routinen-Aufruf

```
Routine_Call =          [Qualident "."] routine_name Actual_Parameters.
Actual_Parameters =     "(" { Expression // "," } ")".
```

Es wird der Aufruf einer Routine ohne Rückgabewert erwartet. Die Parameter-Kompatibilität wird dynamisch getestet. Ist der Qualident angegeben, wird eine Methode in einem - u.U. mehrstufig - referenzierten Objekt aufgerufen.

3. Bedingte Anweisung

```
Conditional_Statement =   IF Bool_Expr THEN Statement_Sequence
                          { ELSIF Bool_Expr THEN Statement_Sequence }
                          [ ELSE Statement_Sequence ] END.
```

Die Bool'schen Ausdrücke werden in der Reihenfolge ihres Auftretens ausgewertet, bis einer den Wert *wahr* ergibt. Dann wird die mit dem Ausdruck assoziierte Anweisungsfolge ausgeführt. Die Anweisungsfolge des Else-Zweigs wird, sofern spezifiziert, dann ausgeführt, wenn alle Bool'schen Ausdrücke zu *falsch* ausgewertet wurden.

4. Wiederholungs-Anweisung

```
Loop_Statement =    LOOP Statement_Sequence END.
```

Eine Loop-Anweisung spezifiziert die wiederholte Ausführung der eingeschachtelten Anweisungsfolge. Sie wird durch eine Exit-Anweisung innerhalb dieser Folge beendet.

5. Exit-Anweisung

```
Exit_Statement =    EXIT.
```

Eine Exit-Anweisung zeigt die Beendigung der umgebenden Loop-Schleife an. Sie darf nur in einer solchen stehen und das Programm wird unmittelbar nach dem Ende der Schleife fortgesetzt.

6. Iterations-Anweisung

```
Iteration_Statement =   ITER  Ident OVER Qualident ":"
                              Statement_Sequence END.
```

Mit der Iterations-Anweisung wird über eine Menge von Objekten gleichen Typs iteriert.

Da keine Ordnung auf den Objektmengen definiert ist, reicht die Wiederholungs-Anweisung nicht aus. Mit Ident wird eine lokale Variable definiert, der nacheinander die Objekte der mittels Qualident beschriebenen Klasse zugewiesen werden. Ihr Gültigkeitsbereich ist auf die Schleife beschränkt. Der Qualident bezeichnet entweder eine (Benutzer-) Klasse von Objekten oder eine Referenz, die (sinnvollerweise) eine Menge von Objekten bezeichnen kann.

Beispiel:

```
FUNCTION Compile_All( prog -> Program );
BEGIN
   ITER module OVER prog.Modules:
      IF Compile( module.SourceName ) THEN
         INCLUDE( CorrectCompiled, module.Status );
      ELSE
         INCLUDE( IncorrectCoompiled, module.Status );
      END;
      EXCLUDE( Modified, module.Status );
   END;
END_FUNCTION Compile_All;
```

7. Return-Anweisung

```
Return_Statement =   RETURN [ "(" Expression ")" ].
```

Eine Return-Anweisung zeigt die Beendigung einer Routine an. Dann und nur dann, wenn im Kopf der Routine spezifiziert wurde, daß sie ein Ergebnis liefern soll, steht ein Ausdruck, dessen Wert zurückgeliefert wird. Der Ausdruck muß vom spezifizierten Typ sein.

In Routinen ohne spezifizierten Ergebnistyp impliziert das Ende des Routinen-Rumpfs ein Return-Statement ohne Resultatswert.

8. System-Aufruf

```
System_Call =  CALL [ Qualident ":=" ] Expression.
```

Es stehen zwei unterschiedliche Routinen-Konstrukte zur Verfügung: Funktionen und Tools. In beiden hat das Call-Konstrukt eine unterschiedliche Bedeutung. Der Rumpf von Methoden besteht aus den gleichen Anweisungen wie der Rumpf von Funktionen und Tools; das Call-Konstrukt ist dort jedoch nicht zulässig. Die Überprüfung muß semantisch getroffen werden.

Funktionen (Function_Declaration) geben die Befehle des Call-Konstrukts an die Ablaufumgebung weiter. So können z.B. Ein- und Ausgabe-Routinen formuliert werden oder es kann auf den Objektraum zugegriffen werden. Die meisten Standardfunktionen (s. Abschnitt 3.10 auf Seite 47) können sehr einfach als Funktionen implementiert werden. In Funktionen kann der Typ des Rückgabewerts erst zur Laufzeit bestimmt werden. Daher wird auch die Korrektheit des Ausdrucks erst zur Laufzeit überprüft; im Fehlerfall erfolgt ein entsprechender Hinweis an den Benutzer und der Fehler wird – sofern möglich – durch Einsetzen eines Defaultwertes abgefangen.

Tools (Tool_Declaration) geben die Befehle des Call-Konstrukts an das zugrundeliegende

Betriebssystem weiter. Sie beziehen sich auf Modifikationen an den tatsächlichen Objekten der Projekt-Umgebung (Dateien usw.). Auch Werkzeuge der Entwicklungsumgebung (z.B. Editoren und Compiler) werden über Tools aufgerufen. Es wird ein Resultat vom Typ INTEGER erwartet, das aber nicht unbedingt ausgewertet werden muß.

In beiden Fällen werden die Befehle als Zeichenketten an die zugrundeliegenden Systeme übergeben. Die Zeichenketten können aus konstanten Teilen, globalen String-Konstanten, Parametern und lokalen Variablen zusammengesetzt werden. Sie werden an geeigneter Stelle direkt in den Quellcode der Ablaufumgebung eingebunden und entweder mit dem restlichen System übersetzt oder einem interpretierenden System kenntlich gemacht.

Beispiel:

```
FUNCTION Print( int : INTEGER );
BEGIN
   CALL "prin1 "+(STRING int)+")";
END_FUNCTION Print;

FUNCTION Print( str : STRING );
BEGIN
   CALL "princ "+str+")";
END_FUNCTION Print;

TOOL Compile( source : STRING ) : BOOLEAN;
VAR success : INTEGER;
BEGIN
   CALL success := "cc "+source;
   RETURN success = 0;
END_TOOL Compile;
```

3.7 Projekte

```
Project = PROJECT project_name ";"
          [ System_Interface ]
          [ Project_Description ]
          END_PROJECT project_name "."
```

Ein Projekt besteht aus einem Systemteil und einem Problemteil. Jeder dieser Teile wiederum besteht (sofern er nicht leer ist) aus einer Folge von Importanweisungen und Deklarationen. Üblicherweise wird eine (ausgezeichnete) Aktivtät durch ein Benutzerereignis mittels Wächter aktiviert und initiiert so die Abwicklung des Projekts.

Der Name am Anfang und am Ende der Projektbeschreibung muß übereinstimmen; er bezeichnet den Namen des Projekts. Der Name der Datei, in der die Beschreibung des Projekts gespeichert wird, ist gleich dem Namen des Projekts und wird durch den Anhang ".ProLan-X" gekennzeichnet.

Der Inhalt dieser Datei ist nicht statisch, da die Projektbeschreibung interaktiv zur Laufzeit des Projekts geändert oder ergänzt werden kann.

3.8 Zusammenfassung der Syntaxregeln

```
(1)  Project =                  PROJECT project_name ";"
                                [ System_Interface ]
                                [ Project_Description ]
                                END_PROJECT project_name "."

(2)  System_Interface =         SYSTEM_INTERFACE
                                { IMPORT project_name ";" }
                                { Function_Declaration | Tool_Declaration }.

(3)  Function_Declaration =     FUNCTION function_name
                                [ Heading ]
                                Routine_Body
                                END_FUNCTION function_name ";".

(4)  Tool_Declaration =         TOOL tool_name
                                [ Heading ]
                                Routine_Body
                                END_TOOL tool_name ";".

(5)  Heading =                  ( WAIT | TRANSFER | NO_WAIT )
                                [ Formal_Parameter_List ]
                                [ ( ":" Simple_Type ) |
                                  ( "->" reference_name ) ] ";".

(6)  Formal_Parameter_List =    "(" { [VAR] Ident_List
                                      ( ":" Simple_Type ) |
                                      ( "->" reference_name ) // ";" } ")".

(7)  Routine_Body =             [ Declaration_Part ]
                                BEGIN Statement_Sequence.

(8)  Declaration_Part =         VAR { Ident_List ":" Simple_Type
                                      [INIT Expression] ";"}+.

(9)  Statement_Sequence =       { Statement ";" }+.

(10) Statement =                Assignment | Routine_Call |
                                Conditional_Statement | Loop_Statement |
                                Exit_Statement | Iteration_Statement |
                                Return_Statement | System_Call.

(11) Assignment =               Qualident ":=" Expression.

(12) Routine_Call =             [Qualident "."] routine_name
                                Actual_Parameters.

(13) Actual_Parameters =        "(" { Expression // "," } ")".

(14) Conditional_Statement =    IF Bool_Expr THEN Statement_Sequence
                                { ELSIF Bool_Expr THEN Statement_Sequence }
                                [ ELSE Statement_Sequence ] END.

(15) Loop_Statement =           LOOP Statement_Sequence END.

(16) Exit_Statement =           EXIT.

(17) Iteration_Statement =      ITER  Ident OVER Qualident ":"
                                      Statement_Sequence END.

(18) Return_Statement =         RETURN [ "(" Expression ")" ].
```

```
(19) System_Call =          CALL [ Qualident ":=" ] Expression.
```

```
(20) Project_Description =  PROJECT_DESCRIPTION
                            { IMPORT project_name ";" }
                            { Project_Object ";" }.

(21) Project_Object =       Symbol_Declaration | Constant_Declaration |
                            Event_Declaration | Entity | Activity |
                            Assertion | Milestone | Activity_Guard |
                            Interval.

(22) Symbol_Declaration =   { symbol_name // "," }+ ":" SYMBOL.

(23) Constant_Declaration = constant_name "=" Const_Expr.

(24) Event_Declaration =    { event_name // "," }+ ":" EVENT.
```

```
(25) Entity =               ENTITY entity_name ";"
                            Entity_Body
                            END_ENTITY entity_name.

(26) Entity_Body =          [ Inheritance ]
                            [ Attribute_Part ]
                            [ Milestone_Links ].

(27) Inheritance =          INHERIT FROM { reference_name // "," }+ ";".

(28) Attribute_Part =       [ PUBLIC ":" { Attribute }+ ]
                            [ PRIVATE ":" { Attribute }+ ].

(29) Attribute =            Declarative_Attribute | Active_Value |
                            Method_Declaration | Link_Attribute.

(30) Declarative_Attribute = attribute_name ":" Simple_Type
                            [ INIT Init_Expr ] [ MONITOR ] ";".

(31) Active_Value =         attribute_name ":=" Expression [MONITOR]";".

(32) Method_Declaration =   METHOD method_name
                            [ Heading ]
                            Routine_Body
                            END_METHOD method_name ";"

(33) Link_Attribute =       ( Reference_Link | Structural_Link ) [ "!" ].

(34) Reference_Link =       attribute_name "->" (   Single_Link |
                                                    Multiple_Link )";".

(35) Structural_Link =      attribute_name "=" (    Single_Link |
                                                    Multiple_Link )";".

(36) Single_Link =          reference_name [ "*" ].

(37) Multiple_Link =        "(" { reference_name // "," }+ ")" [ "*" ].

(38) Milestone_Links =      MILESTONES ":"
                            { attribute_name "->" milestone_name ";" }+.
```

---

(39) Activity = **ACTIVITY** activity_name ";"
Activity_Body
END_ACTIVITY activity_name.

(40) Activity_Body = [Inheritance]
[Super_Activities]
[Attribute_Part]
Interval_Link
[Validation]
[Precondition]
[Action_Sequence]
[Postcondition].

(41) Super_Activities = **SUPER_ACTIVITIES** ":"
{ activity_name // "," }$^{+}$ ";".

(42) Interval_Link = **INTERVAL** ":" attribute_name
["->" interval_name] ";".

(43) Validation = **VALID** ":" Bool_Expr ";".

(44) Precondition = **PRECOND** ":" Bool_Expr ";".

(45) Postcondition = **POSTCOND** ":" Bool_Expr ";".

(46) Action_Sequence = { Action ";" }$^{+}$.

(47) Action = Subactivity_Call | Sequence | Set | Selection
| Loop | Exit | M_Set | Cascade |
Conditional_Action | Elementary_Action.

(48) Subactivity_Call = activity_name [Instantiation_List].

(49) Instantiation_List = "(" { Instantiation_Expression // "," }$^{+}$ ")".

(50) Instantiation_Expression = Ident "=" Expression.

(51) Sequence = **SEQUENCE** Action_Sequence **END**.

(52) Set = **SET** Action_Sequence **END**.

(53) Selection = **SELECTION** { Subactivity_Call ";" }$^{+}$ **END**.

(54) Loop = **LOOP** Action_Sequence **END**.

(55) Exit = **EXIT**.

(56) M_Set = **M_SET** Cardinality_Expr ":" Action_Sequence
END.

(57) Cardinality_Expr = [Ident] **OVER** Qualident.

(58) Cascade = **CASCADE** { Subactivity_Call
["=>" {activity_name // ","}$^{+}$] ";" } **END**.

(59) Conditional_Action = **IF** Bool_Expr **THEN** Action_Sequence
{ **ELSIF** Bool_Expr **THEN** Action_Sequence }
[**ELSE** Action_Sequence] **END**.

(60) Elementary_Action = **EXECUTE** Statement_Sequence **END**.

```
(61) Assertion =              ASSERTION assertion_name ";"
                              Assertion_Body
                              END_ASSERTION assertion_name.

(62) Assertion_Body =         UNLESS Constraint DO Exception_Handling.

(63) Constraint =             Bool_Expr.

(64) Exception_Handling =     [ Action_Sequence ].
```

```
(65) Milestone =              MILESTONE milestone_name ";"
                              Milestone_Body
                              END_MILESTONE milestone_name.

(66) Milestone_Body =         [ Date ]
                              [ Condition ]
                              [ Action_Sequence ].

(67) Date =                   DATE ":" Expression ";".

(68) Condition =              ON Bool_Expr DO.
```

```
(69) Activity_Guard =         GUARD guard_name ";"
                              Guard_Body
                              END_GUARD guard_name.

(70) Guard_Body =             ON Guard
                              [ IF Bool_Expr ]
                              DO Subactivity_Call.

(71) Guard =                  Routine_Call.
```

```
(72) Interval =               INTERVAL interval_name ";"
                              Interval_Body
                              END_INTERVAL interval_name ";".

(73) Interval_Body =          [ EARLIEST_START ":" Expression ";" ]
                              [ LATEST_START ":" Expression ";" ]
                              [ EARLIEST_END ":" Expression ";" ]
                              [ LATEST_END ":" Expression ";" ]
                              [ MIN_DURATION ":" Expression ";" ]
                              [ MAX_DURATION ":" Expression ";" ]
                              { Temporal_Attribute ";" }.

(74) Temporal_Attribute =     attribute_name ":" Temporal_Type
                              [ INIT Init_Expr ].

(75) Temporal_Type =          TIME | DURATION.
```

```
(76) Const_Expr =             Expression.

(77) Init_Expr =              Expression.

(78) Bool_Expr =              Expression.

(79) Expression =             Simple_Expr [ Relop Simple_Expr ].
```

(80) Relop = "=" | "<>" | "<" | "<=" | ">=" | ">" | **IN**.

(81) Simple_Expr = ["+" | "-"] Term { Addop Term }.

(82) Addop = "+" | "-" | **OR**.

(83) Term = Factor { Mulop Factor }.

(84) Mulop = "*" | "/" | **DIV** | **MOD** | **AND**.

(85) Factor = Status | Routine_Call | Qualident | **NOT** Factor | "(" Expression ")" | Type_Conversion | Integer | Real | Boolean | String | Link | Time | Duration | constant_name | event_name.

(86) Status = "{" { symbol_name // "," } "}".

(87) Qualident = { field_name ["[" Qualifier "]"] // "." }.

(88) Qualifier = Bool_Expr.

(89) Type_Conversion = "(" Simple_Type Expression ")".

(90) Simple_Type = **INTEGER** | **REAL** | **BOOLEAN** | **STRING** | **SYMBOL** | **STATUS** | Temporal_Type.

(91) Integer = { digit }$^{+}$.

(92) Real = { digit }$^{+}$ "." { digit }$^{+}$.

(93) Boolean = **TRUE** | **FALSE**.

(94) String = '"' { character } '"' | "'" { character } "'".

(95) Link = **NIL**.

(96) Time = ([HH [":" MM [":" SS]]] DD "." MM "." YYYY) | (MM "." YYYY) | YYYY.

(97) YYYY = MM = DD = HH = SS = Integer.

(98) Duration = ([HH "h"] [MM "min"] [S "s"] [DD "d"] [MM "m"] [YYYY "y"]).

---

(99) project_object_name = assertion_name | guard_name | field_name.

(100) field_name = attribute_name [Actual_Parameters] | reference_name.

(101) reference_name = entity_name | activity_name | interval_name | milestone_name.

(102) routine_name = tool_name | function_name | method_name | std_function_name | std_trigger_name.

(103) .._name = Ident.

(104) Ident_List = { Ident // "," }$^{+}$.

(105) Ident = alpha_char { name_char }.

```
(106)character =          alpha_char | digit | special_char.
(107)name_char =          alpha_char | digit | "_" | "-".
(108)alpha_char =         "A" | ... | "Z" | "a" | ... | "z".
(109)digit =              "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7"
                          | "8" | "9".
(110)special_char =       "_" | "-" | "!" | "@" | "$" | "%" | ...
```

3.9 Liste der Operatoren und Schlüsselwörter

;
.
:
(
)
:=
,
=
!
->
=>
<>
<
<=
>=
>
+
-
*
/
{
}
[
]
“
‘
h
min
s
d
m
y

ACTIVITY
AND
ASSERTION
BEGIN
BOOLEAN
CALL
CASCADE
DATE
DIV
DO
DURATION
EARLIEST_END
EARLIEST_START
ELSE
ELSIF
END
END_ACTIVITY
END_ASSERTION
END_ENTITY
END_FUNDTION
END_GUARD
END_INTERVAL
END_METHOD
END_MILESTONE
END_PROJECT
END_TOOL
ENTITY
EVENT
EXECUTE
EXIT
FALSE
FROM
FUNCTION
GUARD
IF
IMPORT
IN
INHERIT
INIT
INTEGER
INTERVAL
ITER
LATEST_END
LATEST_START
LOOP
MAX_DURATION
METHOD
MILESTONE
MILESTONES
MIN_DURATION
MOD
MONITOR
M_SET
NIL
NOT
NO_WAIT
ON
OR
OVER
POSTCOND
PRECOND
PRIVATE
PROJECT
PROJECT_DESCRIPTION
PUBLIC
REAL
RETURN
SELECTION
SET
SEQUENCE
STATUS
STRING
SUPER_ACTIVITIES
SYMBOL
SYSTEM_INTERFACE
TIME
THEN
TRUE
TOOL
TRANSFER
UNLESS
VALID
VAR
WAIT

3.10 Standardfunktionen von ProLan-X

3.10.1 Allgemeines

Im folgenden bezeichnet reference_name einen Verweis auf die Klassen, die in der Syntax angegeben werden (s. Regel (101)). Der Typ SYMBOL in der Parameterliste bedeutet, daß der Name der entsprechenden Klasse wie ein Symbol (z.B. ENTITY als Systemklasse, Design-Document als benutzerdefinierte Entität, Edit-Module als benutzerdefinierte Aktivität) angegeben wird. Die Instantiatiierungs-Liste bezieht sich ebanfalls auf die entsprechende Syntaxregel (s. Regel (49)).

3.10.2 Trigger-Funktionen

Create-Event

Syntax: CREATE-EVENT(reference_name : SYMBOL;
[Instantiation_List]) : BOOLEAN

Semantik: Liefert *wahr*, wenn eine Instanz einer Benutzerklasse kreiert wurde, für die die Attribute der Instanziierungs-Liste den angegebenen Wert besitzen.

Delete-Event

Syntax: DELETE-EVENT (reference_name : SYMBOL) : BOOLEAN

Semantik: Liefert *wahr*, wenn ein Objekt einer Benutzerklasse vom Plan entfernt wurde.

Read-Event

Syntax: READ-EVENT (reference_name : SYMBOL) : BOOLEAN

Semantik: Liefert *wahr* bei einem lesenden Zugriff auf ein Objekt.

Write-Event

Syntax: WRITE-EVENT (reference_name : SYMBOL) : BOOLEAN

Semantik: Liefert *wahr* bei einem schreibenden Zugriff auf ein Objekt.

Modify-Event

Syntax: MODIFY-EVENT(reference_name : SYMBOL) : BOOLEAN

Semantik: Liefert *wahr*, wenn eine Instanz einer Benutzerklasse modifiziert wurde. Im Unterschied zum WRITE-EVENT muß ein Wert verändert worden sein.

Time-Event

Syntax: TIME-EVENT (event_time : TIME) : BOOLEAN

Semantik: Liefert *wahr* beim Eintreffen einer bestimmten Uhrzeit.

User-Event

Syntax: USER-EVENT (event_name : SYMBOL) : BOOLEAN

Semantik: Liefert *wahr* beim Auslösen eines benutzer-generierten Events.

3.10.3 Laufzeit-spezifische Standard-Funktionen

Is-Activ

Syntax: IS-ACTIV (activity_name : SYMBOL) : BOOLEAN

Semantik: Liefert *wahr*, falls eine Aktivität dieser Klasse zur Zeit Element des Plans (also instanziiert) ist und noch nicht beendet oder gelöscht wurde.

Is-Present

Syntax: IS-PRESENT (reference_name : SYMBOL) : BOOLEAN

Semantik: Liefert *wahr*, falls eine Instanz der entsprechenden Klasse zur Zeit existiert.

Is-Bound (A)

Syntax: IS-BOUND (attribute_name : SYMBOL) : BOOLEAN

Semantik: Liefert *wahr*, falls das Attribut einen Wert hat (gebunden ist).

Is-Bound (B)

Syntax: IS-BOUND (link : Reference_Link;
attribute_name : SYMBOL) : BOOLEAN

Semantik: Liefert *wahr*, falls das Attribut im referenzierten Objekt einen Wert hat (gebunden ist).

Is-Equal

Syntax: IS-EQUAL (link : Reference-Link; link : Reference-Link) :BOOLEAN

Semantik: Liefert *wahr* genau dann, wenn beide Referenzen dieselbe Instanz bezeichnen.

Monitor

Syntax: MONITOR ({ reference_name : SYMBOL // "," }$^{+}$)

Semantik: Zeigt alle mit MONITOR gekennzeichneten deklarativen Attribute (s. Abschnitt 1. auf Seite 12) und Aktivwerte (s. Abschnitt 2. auf Seite 13) aller Instanzen der angegebenen Benutzerklassen an.

Act-Time

Syntax: ACT-TIME () : TIME

Semantik: Liefert das aktuelle Datum mit der aktuellen Uhrzeit.

Act-Date

Syntax: ACT-DATE () : TIME

Semantik: Liefert das aktuelle Datum.

Start-Time

Syntax: START-TIME (interval : DURATION) : TIME

Semantik: Liefert den Startzeitpunkt eines Intervalls (nicht zu verwechseln mit der Systemklasse INTERVAL).

End-Time

Syntax: END-TIME (interval : DURATION) : TIME

Semantik: Liefert den Endzeitpunkt eines Intervalls (nicht zu verwechseln mit der Systemklasse INTERVAL).

Trunc

Syntax: TRUNC (number : REAL) : INTEGER

Semantik: Die gegebene rationale Zahl wird auf die, dem Betrag nach, nächst kleinere ganze Zahl abgeschnitten.

Round

Syntax: ROUND (number : REAL) : INTEGER

Semantik: Die gegebene rationale Zahl wird auf die, dem Betrag nach, nächst kleinere ganze Zahl gerundet.

Substr

Syntax: SUBSTR (str : STRING; start, end : INTEGER) : STRING

Semantik: Liefert den Substring aus dem gegebenen String zurück; der Start ist inklusiv, das Ende exklusiv. Strings beginnen mit dem Index 1.

Length

Syntax: LENGTH (str : STRING) : INTEGER

Semantik: Liefert die Länge des gegebenen Strings.

Number-Of (A)

Syntax: NUMBER-OF (reference_name : SYMBOL) : INTEGER

Semantik: Liefert die Anzahl der momentan existierenden Instanzen des Prototyps.

Number-Of (B)

Syntax: NUMBER-OF (link : Structural_Link) : INTEGER

Semantik: Liefert die aktuelle Anzahl der Komponenten.

Include (A)

Syntax: INCLUDE (element : SYMBOL; set : STATUS)

Semantik: Nimmt das neue Symbol in die gegebene Menge von Symbolen auf.

Include (B)

Syntax: INCLUDE (link : Reference_Link; link : Structural_Link)

Semantik: Nimmt das neue Objekt zu den bisherigen Komponenten hinzu.

Exclude (A)

Syntax: EXCLUDE (element : SYMBOL; set : STATUS)

Semantik: Schließt das neue Symbol aus der gegebenen Mengen von Symbolen aus.

Exclude (B)

Syntax: EXCLUDE (link : Reference_Link; link : Structural_Link)

Semantik: Schließt das angegebene Objekt von den bisherigen Komponenten aus.

Create

Syntax: CREATE (reference_name : SYMBOL; [Instantiation_List])
: Reference_Link

Semantik: Kreiert eine neue Instanz der angegebenen Benutzerklasse und liefert eine Referenz darauf zurück. Steht die Instanziierungs-Liste, werden die entsprechenden Attribute bei der Kreierung mit den angegebenen Werten versorgt. Die Attribute müssen im Public-Teil deklariert sein.
Es dürfen keine Namen von Aktivitäten, Meilensteinen oder Intervallen angegeben werden!

Delete (A)

Syntax: DELETE (reference_name : SYMBOL;
link : Reference_Link) : BOOLEAN

Semantik: Löscht die referenzierte Instanz. Die Funktion liefert *wahr*, wenn die Löschaktion erfolgreich verlaufen ist. Andere Verweise auf dasselbe Objekt werden auf NIL gesetzt.
Aktivitäten dürfen nur gelöscht werden, wenn keine zugehörige Super-Aktivität existiert! Dieser Test wird zur Laufzeit durchgeführt.
Das Löschen von Meilensteinen oder Intervallen ist nicht möglich.

Delete (B)

Syntax: DELETE (type : TypeSet; name : SYMBOL) : BOOLEAN
TypeSet = { ASSERTION, GUARD }

Semantik: Die Zusicherung bzw. der Wächter mit dem angegebenen Namen wird im Speicher (nicht in der ProLan-X-Datei!) gelöscht. Ist die Operation erfolgreich, liefert die Funktion *wahr* zurück.

Delete-Complete

Syntax: DELETE-COMPLETE (link : Reference_Link)

Semantik: Es darf nur eine Entität referenziert werden! Die referenzierte Entität und alle eventuellen Komponenten, die durch Mehrfach-Referenzattribute (s. Abschnitt 5. auf Seite 15) bezeichnet werden und auf die sonst keine Verweise bestehen, werden gelöscht.

Copy

Syntax: COPY (link : Reference_Link) : Reference_Link

Semantik: Liefert eine Kopie des referenzierten Objekts zurück. In Entitäten werden bei enthaltenen Mehrfach-Referenzen (s. Abschnitt 5. auf Seite 15) "tiefe" Kopien gemacht, d.h. es werden auch Kopien der in der Mahrfachreferenz enthaltenen Objekte gemacht.

Select-Object

Syntax: SELECT-OBJECT (reference_name : SYMBOL;
{attribute_name : SYMBOL // ","}$^{+}$) : Reference_Link

Semantik: Eröffnet ein Menü zur Wahl einer der vorhandenen Instanzen der angegebenen Klasse, aufgelistet nach den angegebenen Attribut-Werten. Auch Objekte die Instanz einer Subklasse der angegebenen Klasse sind, werden berücksichtigt.

3.10.4 Quantoren

For-All

Der Allquantor steht nicht als Standardfunktion zur Verfügung, da er durch das M_SET-Konstrukt nachgebildet werden kann.

Exists (A)

Syntax: EXISTS (link : Structural_Link; expr : BOOLEAN) : BOOLEAN

Semantik: Liefert *wahr*, wenn mindestens für eine Komponente der Bool'sche Ausdruck gültig ist.

Exists (B)

Syntax: EXISTS (reference_name : SYMBOL;
expr : BOOLEAN) : BOOLEAN

Semantik: Liefert *wahr*, wenn bei mindestens einer Instanz der angegebenen Benutzerklasse der Bool'sche Ausdruck gültig ist.

3.10.5 ProLan-spezifische Standard-Funktionen

Compile-Project

Syntax: COMPILE-PROJECT (project_name : SYMBOL) : BOOLEAN

Semantik: Compiliert ein Projekt mit dem Namen *project_name*, dessen Beschreibung in der Datei *project_name*.ProLan-X erwartet wird. Die Funktion liefert *wahr*, wenn die Aktion erfolgreich abgeschlossen wurde.

Compile

Syntax: COMPILE (project_name : SYMBOL;
type : TypeSet; { name : SYMBOL }$^{+}$) : BOOLEAN
TypeSet = { PROJECT, FUNCTION, TOOL, SYMBOL, CONSTANT, EVENT, ENTITY, ACTIVITY, ASSERTION, GUARD, INTERVAL, MILESTONE }

Semantik: Es wird der angegebene Teil des Projektes *project_name* übersetzt. Die Beschreibung wird in der Datei *project_name*.ProLan-X erwartet. Die Funktion liefert *wahr*, wenn die Aktion erfolgreich abgeschlossen wurde; dann wird das neue Objekt dem aktuellen Projekt hinzugefügt und kann ab sofort benutzt werden. Sonst bleibt die aktuelle Beschreibung des aktuellen Projekts gültig.

Replace

Syntax: REPLACE (type : TypeSet; name : SYMBOL) : BOOLEAN
TypeSet = { ACTIVITY, ENTITY, ASSERTION, GUARD, INTERVAL, MILESTONE }

Semantik: Zuerst wird ein Editor-Fenster an der angegebenen Stelle der Datei mit der aktuellen Projektbeschreibung, die in der Datei *project_name*.ProLan-X erwartet wird, geöffnet. Dann wird die Benutzerklasse der angegebenen Klasse mit dem Namen *name* compiliert. Im aktuellen Projekt muß eine Klasse dieses Namens existieren. Tritt beim Übersetzen ein Fehler auf, bleibt die alte Beschreibung erhalten. Die Funktion liefert *wahr*, wenn die Aktion erfolgreich abgeschlossen wurde.

Handelt es sich bei der neu zu definierenden Klasse um eine Aktivität, so wird die Übersetzung abgelehnt, falls Instanzen dieser Klasse existieren, deren Abarbeitung bereits begonnen und noch nicht abgeschlossen wurde. In diesem Fall liefert die Funktion den Wert *falsch* zurück.

Ein Ersetzen einer Benutzerklasse durch eine andere (gleichen Namens) zeigt erst Auswirkungen, wenn neue Instanzen kreiert werden. Auf bestehende Instanzen, gleich, ob aktive oder passive (bei Aktivitäten), wirken sich die Veränderungen nicht aus.

3.10.6 Sonstige Standard-Funktionen

Actual-Project

Syntax: ACTUAL-PROJECT () : STRING

Semantik: Liefert einen String mit den Namen des aktuellen Projekts zurück.

Loaded-Projects

Syntax: LOADED-PROJECTS ()

Semantik: Gibt die aktuell geladenen Projekte auf dem Bildschirm aus.

Load-Project

Syntax: LOAD-PROJECT (project_name : SYMBOL)

Semantik: Lädt das Projekt mit dem Namen *project_name*, dessen Beschreibung in der Datei project_name.ProLan-X erwartet wird. Die Funktion liefert *wahr*, falls die Aktion erfolgreich verlaufen ist.

Freeze-Project

Syntax: FREEZE-PROJECT () : BOOLEAN

Semantik: Friert das aktuelle Projekt im aktuellen Zustand ein. Die Funktion liefert *wahr*, falls die Aktion erfolgreich verlaufen ist.

Continue-Project

Syntax: CONTINUE-PROJECT (project_name : SYMBOL) : BOOLEAN

Semantik: Übernimmt das Projekt mit dem Namen *project_name*, wie es (mit FREEZE-PROJECT) eingefroren wurde. Die Funktion liefert *wahr*, falls die Aktion erfolgreich verlaufen ist.

4. Literatur

[Allen] James F. Allen
"Maintaining Knowledge about Temporal Intervals"
in: Communications of the ACM November 1983, Vol. 26, No. 11, Seite 832 - 843

[Bleisinger] Rainer Bleisinger
"TEMPO - Ein integrierter Ansatz zur Modellierung qualitativer und quantitativer zeitlicher Informationen",
Proceedings der 15. Fachtagung für Künstliche Intelligenz, GWAI - 91, Bonn

[Jensen, Wirth] K. Jensen, N. Wirth
"PASCAL - User Manual and Report",
Springer Verlag, 1974

[Knauber] Peter Knauber
"Entwurf und Implementierung einer Beschreibungssprache für den Software-Entwicklungsprozeß",
Diplomarbeit, Universität Kaiserslautern, 1991

[Mössenböck, Wirth] Hanspeter Mössenböck, Niklaus Wirth
"The Programming Language Oberon-2",
ETH Zürich

[Schramm] Wolfgang Schramm
"Ein objektorientiertes Metamodells für die Software-Entwicklung",
Dissertation, Universität Kaiserslautern, 1991

[Verlage] Martin Verlage
"Entwurf und Implementierung eines Projektinterpreters",
Diplomarbeit, Universität Kaiserslautern, 1991

[Verlage, Knauber] Martin Verlage, Peter Knauber
"Just-in-Time Initialization of Objects Representing Software Processes",
Proceedings of the acm Symposium on Applied Computing, Kansas, March 1992

[Wirth] Niklaus Wirth
"Programming in MODULA-2",
Springer Verlag, 1983