Interner Bericht

Integration temporallogischer Verarbeitungskonzepte in C^{++}

Paul Kirchberg   Otto Mayer

Dezember 1997

294/97

Fachbereich Informatik

Universität Kaiserslautern · Postfach 3049 · D-67653 Kaiserslautern

Zusammenfassung

In dieser Arbeit wird eine Integration der temporallogischen Verarbeitungskonzepte der Programmiersprache ExTeLL in die objektorientierte Wirtssprache C^{++} vorgestellt. Dabei war unser Ziel eine Schnittstelle zur komfortablen Kommunikation der Sprachkomponenten zu entwickeln, derart daß die Sprachsynthese eine homogene Gesamtsprache darstellt. Hierbei haben wir besonderen Wert auf die Nutzung der Möglichkeiten der jeweils hinzugefügten Sprachkomponente und einen syntaktisch einheitlichen Aufbau der Gesamtsprache gelegt. Dies erforderte insbesondere die Integration des Typkonzepts von C^{++} sowie der Mechanismen zur Überladung von Funktionen und Prozeduren in ExTeLL und in der zugrundeliegenden Temporallogik EITeL.

1 Einführung 3

2 Die Temporallogik EITeL 3

3 EITeL^T: Typisierung von EITeL 9

4 Von EITeL^T zur Programmiersprache ExTeLL^T 18

5 Integrationsarten 24

6 Aufbau der syntaktischen Sprachintegration 27

7 Aufbau des ExTeLL^{++}–Compilers 33

8 Ausblick 35

1 Einführung

Die in [Spi95] vorgestellte temporallogische Programmiersprache ExTeLL erlaubt die Repräsentation und Simulation zeitlicher Abläufe in einer natürlichen Form ([Kan92, Kir94, Schwi93, Spi95]). ExTeLL–Programme entsprechen dabei Formeln der intervallbasierten Logik EITeL, in der die temporalen Operatoren *always, sometimes, next* sowie deren vergangenheitsorientierte Formen zur Beschreibung temporaler Beziehungen zur Verfügung stehen. Weiterhin existiert in EITeL der Operator *chop* zur Verkettung von Zeitintervallen.

Da für viele Anwendungen die Behandlung zeitlicher Aspekte nur einen Teil der Gesamtaufgabe darstellt, wird in einem von der Deutschen Forschungsgemeinschaft geförderten Projekt ein Konzept entwickelt, mit dem sich ExTeLL in verbreitete Programmiersprachen integrieren läßt. Damit können dann Ergebnisse temporaler Berechnungen für Auswertungen in Form von Analysen und Ansteuerungen externer Komponenten, deren Behandlung in ExTeLL oft unnatürlich und umständlich erscheint, mit Hilfe der in der Wirtssprache vorhandenen Mechanismen durchgeführt werden. Dies ermöglicht eine Trennung zwischen der temporalen Bearbeitung und der darauf aufbauenden Wissensbewertung. Als Wirtssprache wird hierbei C^{++} verwendet.

Für die Integration ist eine Schnittstelle zwischen den beiden Sprachkomponenten erforderlich, über die Formeln und Variablenbelegungen an die ExTeLL–Komponente übergeben und die ermittelten Ergebnisse an die Verarbeitungskomponente zurückgegeben werden können. Dabei soll eine homogene Gesamtsprache entstehen. Hierzu sind primär Erweiterungen am ExTeLL-Kern erforderlich. Insbesondere ist im Hinblick auf die Kommunikation von ExTeLL mit ihrer jeweiligen Umgebung die Integration eines Typkonzeptes in EITeL und ExTeLL erforderlich.

In den folgenden Kapiteln 2 und 3 werden die Logik EITeL sowie ihre typisierte Form $\text{EITeL}^{\mathcal{T}}$ vorgestellt. Darauf aufbauend wird in Kapitel 4 auf die typisierte, temporallogische Programmiersprache $\text{ExTeLL}^{\mathcal{T}}$ eingegangen. Hiernach folgt eine Beschreibung verschiedener Integrationsarten bzgl. imperativer Wirtssprachen. Nach Festlegung einer geeigneten Integrationsart wird in Kapitel 6 die syntaktische Eingliederung entwickelt. Abschließend wird der Aufbau eines Compilers für die Sprachvereinigung vorgestellt.

2 Die Temporallogik EITeL

ExTeLL basiert auf der Temporallogik EITeL, die im Hinblick auf eine algorithmische Verarbeitung mittels Transformationen entwickelt wurde, wobei ein diskretes, unendliches Zeitmodell zugrunde liegt.[1] EITeL ist eine Modallogik, d.h. eine um modale Operatoren erweiterte Prädikatenlogik erster Stufe. Während in der klassischen Logik jeweils

[1]Eine ausführliche Darstellung von EITeL findet sich in [SpMa92]. Von der dort jeweils implizit gegebenen Funktions- und Prädikatsstelligkeit gehen wir im Hinblick auf nachfolgende Erweiterungen auf eine Definition von Termen und Formeln mittels einer Signaturdarstellung über.

Eigenschaften eines Modells beschrieben werden, haben modallogische Betrachtungen das Ziel, ein ganzes System von möglichen Situationen zu charakterisieren. Zwischen diesen Situationen bestehen Übergangsmöglichkeiten, die jeweils eine Veränderung bezüglich einer neuen Situation beschreiben. Eine Veränderung kann beispielsweise ein Wechsel von Voraussetzungen oder, wie in EITeL, eine zeitliche Änderung bedeuten. Mittels der modalen Operatoren sind Aussagen über mehrere Situationen bzw. über mehrere Zeitpunkte möglich.

2.1 Syntax von EITeL

Definition 2.1 Ein *Alphabet* $\mathcal{A}$ der erweiterten Intervall–Temporal–Logik besteht aus:

- einer Menge $\mathcal{T} = \{$'(' , ')' , ',' , ':' , '$\langle$' , '$\rangle$' , '|'$\}$ von *(Trenn–) Zeichen*
- einer Menge $\mathcal{L} = \{$'$\neg$' , '$\wedge$'$\}$ von *Junktoren*
- dem *Existenzquantor* '$\exists$'
- einer Menge $\mathcal{M} = \{$'$\Box$', '$\circ$', '$\Box_p$', '$\circ_p$', ';', '$;_p$'$\}$ von *Modaloperatoren*
- einer abzählbaren Menge $\mathcal{V}$ von *Variablen*
- einer abzählbaren Menge $\mathcal{F}$ von *Funktionssymbolen*, wobei '$\langle\rangle$' (leere Liste) $\in \mathcal{F}$
- einer abzählbaren Menge $\mathcal{P}$ von *Prädikatssymbolen*, wobei '*true*', '*start*', '*more*' und '=' $\in \mathcal{P}$

Definition 2.2 Eine *Signatur sig* zu einem Alphabet $\mathcal{A}$ ist ein Quadrupel $(\mathcal{F}, \mathcal{P}, \tau_\mathcal{F}, \tau_\mathcal{P})$ mit:

- $\mathcal{F}$ ist Menge der Funktions- und $\mathcal{P}$ die Menge der Prädikatssymbole aus $\mathcal{A}$.
- $\tau_\mathcal{F}$ ist eine Funktion $\mathcal{F} \longrightarrow \mathbb{N}_0$, sie gibt für jedes $f \in \mathcal{F}$ die Stelligkeit an.
- $\tau_\mathcal{P}$ ist eine Funktion $\mathcal{P} \longrightarrow \mathbb{N}_0$, sie gibt für jedes $p \in \mathcal{P}$ die Stelligkeit an. Dabei sei $\tau_\mathcal{P}(true) = \tau_\mathcal{P}(start) = \tau_\mathcal{P}(more) = 0$ und $\tau_\mathcal{P}(=) = 2$.

Die Menge $f \in \mathcal{F}$ mit $\tau_\mathcal{F}(f) = 0$ heißt auch die Menge der *Konstanten*.

In EITeL verwendet man die Operatoren $\circ$ und $\circ_p$ um sich auf die Interpretation des Arguments im nachfolgenden bzw. vorherigen Zeitpunkt zu beziehen. Weiterhin verfügt EITeL über Konstrukte zum Umgang mit geschachtelten Listen. Variablen werden künftig durch die Buchstaben $x, y, z, ...$ bezeichnet. Für Funktionssymbole werden meist die Buchstaben $f, g, h, ...$ und für Prädikatssymbole die Buchstaben $p, q, r, ...$ verwendet. Dabei können die Bezeichner auch indiziert auftreten.

Definition 2.3 *Terme* über einer Signatur *sig* und einem Alphabet $\mathcal{A}$ sind induktiv definiert durch:

- Jede Variable $x \in \mathcal{V}$ ist ein Term.
- Ist $f \in \mathcal{F}$ ein Funktionssymbol mit $\tau_{\mathcal{F}}(f) = \text{n}$ und sind $t_1, ..., t_n$ Terme, so ist $f(t_1, ..., t_n)$ ein Term.
- Ist t ein Term, so sind $\circ t$ und $\circ_p t$ Terme.
- Listenkonstruktionen
  - *Listenaufbau:* Sind $t_1, ..., t_n$ Terme, n$\geq$0, so ist $\langle t_1, ..., t_n \rangle$ ein Term.
  - *Listenlänge:* Ist t ein Term, so ist $\mid t \mid$ ein Term.
  - *Indizierung:* Sind $t, t_1, ..., t_n$ Terme, n$\geq$1, so ist $t(t_1, ..., t_n)$ ein Term.

Die Menge $TERM(\mathcal{F}, \mathcal{V})$ bezeichnet die Menge aller nach obiger Regeln bildbaren Terme, $TERM(\mathcal{F})$ die Menge der variablenfreien Terme (*Grundterme*).

In Formeln verwenden wir die modalen Operatoren analog um zustandsübergreifende Aussagen zu formulieren:

Definition 2.4 *Formeln* über einer Signatur sig und einem Alphabet $\mathcal{A}$ sind induktiv definiert durch:

- *Atomare Formeln*: Ist $p \in \mathcal{P}$ mit $\tau_{\mathcal{P}}(p) = \text{n}$ und sind $t_1, ..., t_n$ Terme, so ist $p(t_1, ..., t_n)$ eine *atomare Formel.*
- *Negation* und *Konjunktion*: Sind w, w_1 und w_2 Formeln, so sind $\neg w$ und $w_1 \wedge w_2$ Formeln.
- *Quantifizierung*: Ist w eine Formel und $x \in \mathcal{V}$, so ist $\exists x : w$ eine Formel.
- *Next* und *Previous*: Ist w eine Formel, so sind $\circ w$ und $\circ_p w$ Formeln.
- *Always* und *Past-Always*: Ist w eine Formel, so sind $\Box w$ und $\Box_p w$ Formeln.
- *Chop* und *Past-Chop*: Sind w_1 und w_2 Formeln, so sind $w_1 ; w_2$ und $w_1 ;_p w_2$ Formeln.

Die Menge $FORM(\mathcal{P}, \mathcal{F}, \mathcal{V})$ bezeichnet die Menge aller nach obiger Regeln bildbaren Formeln.

2.2 Semantik von EITeL

2.2.1 Grundlegende Definitionen

Definition 2.5 Eine *Präinterpretation* P einer Signatur $sig = (\mathcal{F}, \mathcal{P}, \tau_{\mathcal{F}}, \tau_{\mathcal{P}})$ ist ein Trippel $P = (D, \mathcal{I}_{\mathcal{F}}, \mathcal{I}_{\mathcal{P}})$ mit:

- D ist eine nicht leere Menge, der *Bereich* oder das *Universum* der Präinterpretation P.

- $\mathcal{I}_\mathcal{F}$ ist eine Abbildung, die jedem Funktionssymbol $f \in \mathcal{F}$ mit $\tau_\mathcal{F}(f) = n$ eine Abbildung $\mathcal{I}_\mathcal{F}(f) : \mathrm{D}^n \to D$ zuordnet.
- $\mathcal{I}_\mathcal{P}$ ist eine Abbildung, die jedem Prädikatssymbol $p \in \mathcal{P}$ mit $\tau_\mathcal{P}(p) = n$ eine Abbildung $\mathcal{I}_\mathcal{P}(p) : \mathrm{D}^n \to \{true, false\}$ zuordnet.

Zur Definition der Semantik von Termen und Formeln muß auch die Bedeutung von Variablen definiert werden. In der klassischen Prädikatenlogik geschieht dies durch die Belegung der Variablen aus $\mathcal{V}$ mit Werten aus D. Die zeitliche Dimension erfassen wir mit den Begriffen *Zustand* und *Intervall*, womit die Betrachtung einer Folge von Zeitpunkten mit den jeweiligen Variablenbelegungen möglich wird.

Definition 2.6 Sei $\mathcal{V}$ die Menge der Variablen eines Alphabetes der erweiterten Intervall–Temporal–Logik, D eine nicht–leere Menge.
Ein *Zustand* σ über D ist eine Abbildung $\sigma : \mathcal{V} \to D$.
Die Menge dieser Zustände wird mit $\Sigma_{\mathcal{V},D}$ bezeichnet, also $\Sigma_{\mathcal{V},D} = \{\sigma : \mathcal{V} \to D\}$.[2]
Die Menge der endlichen Folgen von Zuständen wird mit Σ^+ bezeichnet.

Definition 2.7 Ein *Intervall* $\Im$ über D ist ein Paar (σ, i), bestehend aus einer endlichen Folge $\sigma = \sigma_0, ..., \sigma_n \in \Sigma^+$ von Zuständen über D und einem Index $i \in \mathbb{N}_0$ mit $0 \leq i \leq n$.
σ_i heißt *Gegenwartszustand* oder *aktueller Zustand* des Intervalls (σ, i); i heißt dann auch *Gegenwarts-* bzw. *aktueller Zeitpunkt.*
$\sigma_P := \sigma_0, ..., \sigma_i$ heißt *Vergangenheitsanteil* des Intervalls (σ, i),
$\sigma_F := \sigma_i, ..., \sigma_n$ heißt *Zukunftsanteil* des Intervalls (σ, i).
$|(\sigma, i)| := |\sigma| := n$ heißt die *Länge* des Intervalls (σ, i), sie entspricht der Anzahl der *Zustandsübergänge.*

Bemerkung: Wenn der jeweilige Bereich aus dem Kontext ersichtlich ist, wird meist auf den Zusatz "über D" verzichtet, man schreibt einfach Zustand bzw. Intervall.

Definition 2.8 Eine *Interpretation* $\mathcal{I}$ ist ein Paar $\mathcal{I} = (P, \Im)$ bestehend aus einer Präinterpretation P einer Signatur $sig = (\mathcal{F}, \mathcal{P}, \tau_\mathcal{F}, \tau_\mathcal{P})$ und einem Intervall $\Im$ über dem Bereich D der Präinterpretation P.

Nach dieser Definition des Begriffs der Interpretation ist es möglich, Terme und Formeln auf Intervallen zu interpretieren. Zunächst wird noch ein technischer Begriff eingeführt, der bei der Definition der Semantik von Quantoren nützlich sein wird.

Definition 2.9 Seien σ und $\sigma' \in \Sigma^+$ Folgen von Zuständen und $x \in \mathcal{V}$ eine Variable. σ und σ' sind *äquivalent modulo* x (geschrieben $\sigma \sim_x \sigma'$), falls $|\sigma|=|\sigma'|$ und $\sigma_i(y) = \sigma_i'(y)$ für alle $y \neq x$ und für alle $i \leq |\sigma|$.

[2]Wenn $\mathcal{V}$ und D aus dem Kontext ersichtlich sind, wird meist auf die Indizes verzichtet.

2.2.2 Interpretation von Termen und Formeln auf Intervallen

Definition 2.10 Sei $\mathcal{I} = (P, \Im)$ eine Interpretation einer Signatur $sig = (\mathcal{F}, \mathcal{P}, \tau_{\mathcal{F}}, \tau_{\mathcal{P}})$ mit $P = (D, \mathcal{I}_{\mathcal{F}}, \mathcal{I}_{\mathcal{P}})$ und $\Im = (\sigma, i) = (\sigma_0, ..., \sigma_{|\sigma|}, i) \in \Sigma^+ \times \mathbb{N}_0$. Die zugehörige *Interpretationsabbildung* $\mathcal{I}_{(\sigma,i)}$ von Termen und Formeln ist induktiv mit den folgenden Eigenschaften definiert:

- *auf Termen:*

  a) Ist $v \in \mathcal{V}$ eine Variable, so sei $\mathcal{I}_{(\sigma,i)}(v) := \sigma_i(v)$.

  b) Ist $f \in \mathcal{F}$ ein Funktionssymbol mit $\tau_{\mathcal{F}}(f) = n$ und sind $t_1, ..., t_n$ Terme, so sei $\mathcal{I}_{(\sigma,i)}(f(t_1, ..., t_n)) := \mathcal{I}_{\mathcal{F}}(f)(\mathcal{I}_{(\sigma,i)}(t_1), ..., \mathcal{I}_{(\sigma,i)}(t_n))$.

  c) Ist t ein Term, $|\sigma| > i$, so sei $\mathcal{I}_{(\sigma,i)}(\circ t) := \mathcal{I}_{(\sigma,i+1)}(t)$.

  d) Ist t ein Term, $i > 0$, so sei $\mathcal{I}_{(\sigma,i)}(\circ_p t) := \mathcal{I}_{(\sigma,i-1)}(t)$.

  e) Sind $t_0, ..., t_n$ Terme, so sei $\mathcal{I}_{(\sigma,i)}(\langle t_0, ..., t_n \rangle) := \langle \mathcal{I}_{(\sigma,i)}(t_0), ..., \mathcal{I}_{(\sigma,i)}(t_n) \rangle$. Die rechte Seite der Definition bezeichne die übliche Listenbildung.

  f) Sind t und t_1,...,t_n Terme, so sei $\mathcal{I}_{(\sigma,i)}(t(t_1,...,t_n)) := \mathcal{I}_{(\sigma,i)}(t)_{\mathcal{I}_{(\sigma,i)}(t_1),...,\mathcal{I}_{(\sigma,i)}(t_n)}$. Die rechte Seite der Definition bezeichne die übliche Listenindizierung, wobei die Listenelemente beginnend von links mit 0 zu indizieren sind.

  g) Ist t ein Term, so sei $\mathcal{I}_{(\sigma,i)}(|\, t \,|) := |\, \mathcal{I}_{(\sigma,i)}(t) \,|$. Die rechte Seite der Definition bezeichne die übliche Listenlänge.

- auf Formeln

  h) $\mathcal{I}_{(\sigma,i)}(\mathit{true}) :=$ true

  $$\mathcal{I}_{(\sigma,i)}(\mathit{more}) := \begin{cases} \text{true}, & \text{falls } |\sigma| > i \\ \text{false}, & \text{sonst} \end{cases}$$

  $$\mathcal{I}_{(\sigma,i)}(\mathit{start}) := \begin{cases} \text{true}, & \text{falls } i = 0 \\ \text{false}, & \text{sonst} \end{cases}$$

  i) Ist $p \in \mathcal{P}$ Prädikatssymbol mit $\tau_{\mathcal{P}} = n$ und sind $t_1, ..., t_n$ Terme, so sei $\mathcal{I}_{(\sigma,i)}(p(t_1, ..., t_n)) := \mathcal{I}_{\mathcal{P}}(p)(\mathcal{I}_{(\sigma,i)}(t_1), ..., \mathcal{I}_{(\sigma,i)}(t_n))$.

  j) Sind t_1 und t_2 Terme, so sei
  $$\mathcal{I}_{(\sigma,i)}(t_1 = t_2) := \begin{cases} \text{true}, & \text{falls } \mathcal{I}_{(\sigma,i)}(t_1) = \mathcal{I}_{(\sigma,i)}(t_2) \\ \text{false}, & \text{sonst} \end{cases}$$

  k) Ist w eine Formel, so sei
  $$\mathcal{I}_{(\sigma,i)}(\neg w) := \begin{cases} \text{true}, & \text{falls } \mathcal{I}_{(\sigma,i)}(w) = \text{false} \\ \text{false}, & \text{sonst} \end{cases}$$

  Sind w_1 und w_2 Formeln, so sei
  $$\mathcal{I}_{(\sigma,i)}(w_1 \wedge w_2) := \begin{cases} \text{true}, & \text{falls } \mathcal{I}_{(\sigma,i)}(w_1) = \text{true und } \mathcal{I}_{(\sigma,i)}(w_2) = \text{true} \\ \text{false}, & \text{sonst} \end{cases}$$

l) Ist w eine Formel und $x \in \mathcal{V}$ eine Variable, so sei

$$\mathcal{I}_{(\sigma,i)}(\exists x : w) := \begin{cases} \text{true}, & \text{falls es } \sigma' \in \Sigma^+ \text{ gibt mit } \sigma \sim_x \sigma' \\ & \text{und } \mathcal{I}_{(\sigma',i)}(w) = \text{true} \\ \text{false}, & \text{sonst} \end{cases}$$

m) Ist w eine Formel, so sei

$$\mathcal{I}_{(\sigma,i)}(\circ w) := \begin{cases} \text{true}, & \text{falls } |\sigma| > i \text{ und } \mathcal{I}_{(\sigma,i+1)}(w) = \text{true} \\ \text{false}, & \text{sonst} \end{cases}$$

n) Ist w eine Formel, so sei

$$\mathcal{I}_{(\sigma,i)}(\circ_p w) := \begin{cases} \text{true}, & \text{falls } i > 0 \text{ und } \mathcal{I}_{(\sigma,i-1)}(w) = \text{true} \\ \text{false}, & \text{sonst} \end{cases}$$

o) Ist w eine Formel, so sei

$$\mathcal{I}_{(\sigma,i)}(\Box w) := \begin{cases} \text{true}, & \text{falls für alle } j \text{ mit } i \leq j \leq |\sigma| \text{ gilt: } \mathcal{I}_{(\sigma,j)}(w) = \text{true} \\ \text{false}, & \text{sonst} \end{cases}$$

p) Ist w eine Formel, so sei

$$\mathcal{I}_{(\sigma,i)}(\Box_p w) := \begin{cases} \text{true}, & \text{falls für alle } j \text{ mit } 0 \leq j \leq i \text{ gilt: } \mathcal{I}_{(\sigma,j)}(w) = \text{true} \\ \text{false}, & \text{sonst} \end{cases}$$

q) Sind w_1 und w_2 Formeln, so sei

$$\mathcal{I}_{(\sigma,i)}(w_1; w_2) := \begin{cases} \text{true}, & \text{falls es ein } j \text{ gibt mit } i \leq j \leq |\sigma| \text{ und} \\ & \mathcal{I}_{(\sigma_0,\ldots,\sigma_j,i)}(w_1) = \text{true und} \\ & \mathcal{I}_{(\sigma_0,\ldots,\sigma_{|\sigma|},j)}(w_2) = \text{true} \\ \text{false}, & \text{sonst} \end{cases}$$

r) Sind w_1 und w_2 Formeln, so sei

$$\mathcal{I}_{(\sigma,i)}(w_1 ;_p w_2) := \begin{cases} \text{true}, & \text{falls es } k \text{ und } j \text{ gibt mit } k \leq j \leq i \text{ und} \\ & \mathcal{I}_{(\sigma_0,\ldots,\sigma_j,k)}(w_1) = \text{true und} \\ & \mathcal{I}_{(\sigma_0,\ldots,\sigma_{|\sigma|},j)}(w_2) = \text{true} \\ \text{false}, & \text{sonst} \end{cases}$$

Diese Anforderungen definieren $\mathcal{I}_{(\sigma,i)}$ nicht vollständig, so ist beispielsweise $\mathcal{I}_{(\sigma,0)}(\circ_p(t))$ nicht festgelegt. Im folgenden wird aus Gründen der Übersichtlichkeit $\mathcal{I}_{(\sigma,i)}$ mit der Bedeutung 'für alle $\mathcal{I}_{(\sigma,i)}$, die den Anforderungen aus Definition 2.10 genügen' verwendet.

Definition 2.11 Sei $\mathcal{I} = (P, \Im)$ eine Interpretation mit der Präinterpretation $P = (D, \mathcal{I}_\mathcal{F}, \mathcal{I}_\mathcal{P})$ und einem Intervall $\Im = (\sigma, i) = (\sigma_0, ..., \sigma_n, i)$. Eine Formel w wird von der Interpretation $\mathcal{I}$ *erfüllt*, falls $\mathcal{I}_\Im(w) = true$ ist. Man schreibt dann auch $\models_\Im w$. In diesem Fall nennt man $\mathcal{I}$ auch *Modell* für w.
Falls w von allen Interpretationen mit der Präinterpretation P erfüllt wird, d.h. $\mathcal{I}_\Im(w) = true$ für alle Intervalle $\Im \in \Sigma^+ \times \mathbb{N}_0$, so schreibt man $\models_P w$ und sagt *w ist gültig unter P*.
Eine Formel w *gilt in einem Zustand* σ_j, $0 \leq j \leq n$, falls $\mathcal{I}_{(\sigma,j)}(w) = true$. Dabei verwendet man für den Begriff Zustand synonym die Formulierung *Zeitpunkt eines Intervalls*.

Definition 2.12 Ist $\mathcal{I} = (P, \Im)$ eine Interpretation in der die Präinterpretation P den Symbolen $+$, $-$, $<$, $>$ usw. die übliche Bedeutung zuweist, so heißt P auch *Standard-Präinterpretation* und $\mathcal{I}$ *Standard-Interpretation.* Eine Standard-Interpretation, die Modell ist, heißt *Standard-Modell.*

Zur Verbesserung der Lesbarkeit der nachfolgenden Ausführungen werden folgende Schreibweisen eingeführt:

Eine Formel w *gilt auf einem Intervall* $\Im = (\sigma, i)$, wenn $\mathcal{I}_{(\sigma,i)} = true$ für jede Standard-interpretation $\mathcal{I} = (P, \Im)$ über $\Im$. Man sagt hierfür auch, daß die Folge σ von Zuständen die Formel w *im Zeitpunkt* i *erfüllt.*

Die Aussage "eine Folge σ von Zuständen erfüllt w im Zeitpunkt i" ist demnach gleichbedeutend mit σ erfüllt $\circ^i w$. Fehlt die explizite Angabe des Zeitpunkts i, so gilt der Zeitpunkt $i = 0$.

3 EITeL^T: Typisierung von EITeL

Die in [SpMa92] definierte Logik EITeL verwendet keine expliziten Datentypen. Eine Einbeziehung von Wertebereichen und deren Zusammenhänge vereinfacht jedoch den Umgang mit logischen Aussagen bei der Modellierung einer Problemstellung. Wird beispielsweise die Verwendung von Funktions- bzw. Prädikatssymbolen auf bestimmte Parameterarten eingeschränkt, so kann eine irreguläre Verwendung eines Symbols bereits bei der syntaktischen Überprüfung erkannt werden kann.

Der Begriff *Wert* wird in der Informatik oft recht unscharf verwendet. Wie verwenden den Begriff für eine beliebige Größe, die während einer Berechnung existiert (vgl. [Watt96]). Dazu gehören alle Variableninhalte, Funktions- und Prädikatsparameter sowie Funktionsergebnisse. In der Programmiersprache C^{++} finden sich beispielsweise folgende Arten von Werten:

- Einfache Werte (Zeichen, Aufzählungswerte, ganze und reelle Zahlen, ...)
- Zusammengesetzte Werte (Klassen, Arrays, Dateien, ...)
- Zeiger
- Referenzen auf Variablen

In Programmiersprachen werden Werte meist in Typen eingeteilt:

Definition 3.1 Ein *Datentyp* oder kurz *Typ* T ist eine Menge von Werten. Im Falle $s \in T$ sagen wir: s hat den Typ T.
In Verbindung mit logischen Spezifikationen wird anstatt des Begriffs *Typ* meist der Begriff *Sorte* verwendet.

Der Einsatz von Typen ist bei imperativen Programmiersprachen von elementarer Bedeutung. So verhindert beispielsweise die Definition von Funktionen und Prozeduren unter Angabe der Argumenttypen die Bildung von Ausdrücken und Anweisungen, in denen Konstrukte wie die Multiplikation einer Zahl mit einem Wahrheitswert oder der Vergleich einer Farbe mit einem Wochentag auftreten.

Zur Einführung von Sorten in EITeL müssen die vorangegangenen Definitionen erweitert werden:

Definition 3.2 Ein *Alphabet* $\mathcal{A}$ der typisierten erweiterten Intervall–Temporal–Logik besteht aus:

- den Mengen $\mathcal{T}$ (Trennzeichen), $\mathcal{L}$ (Junktoren), $\mathcal{M}$ (Modaloperatoren), $\mathcal{F}$ (Funktionssymbole), $\mathcal{P}$ (Prädikatssymbole) und dem Existenzquantor '$\exists$' wie in Definition 2.1.
- einer endlichen, nicht leeren Menge $\mathcal{S}$ von *Sorten.* Dabei sei $\mathcal{S}_B \subset \mathcal{S}$ eine endliche, nicht leere Menge von Basissorten. Insbesondere sei die Menge der natürlichen Zahlen $\mathbb{N}_0 \in \mathcal{S}_B$.
  Weiterhin gelte: für alle $s \in \mathcal{S}_B : list_s \in \mathcal{S}$, d.h. $\mathcal{S}$ enthalte zu jeder Basissorte ist auch Listen von Elementen dieser Sorte.
- einer abzählbaren Menge $\mathcal{V}_s$ von *Variablen* für jede Sorte $s \in \mathcal{S}$. Jede Variable $x \in \mathcal{V}_s$ heißt eine *Variable der Sorte s*. Weiterhin sei $\mathcal{V} = \bigcup_{s \in \mathcal{S}} \mathcal{V}_s$ und $\mathcal{V}_s \cap \mathcal{V}_{s'} = \emptyset$ für $s \neq s'$.

Die Integration der Sorten bedingt, daß Zustände, also die Belegung der Variablen, und damit auch Intervalle *sortentreu* sind.

Definition 3.3 Sei $\mathcal{V}$ die Menge der Variablen eines Alphabets der typisierten erweiterten Intervall–Temporal–Logik, $D = \bigcup_{s \in \mathcal{S}} \mathcal{D}_s$ mit $D_s \neq \emptyset$ für alle $s \in \mathcal{S}$.
Ein *Zustand* σ über D ist eine Abbildung $\sigma : \mathcal{V} \to D$, wobei für alle $x_s \in \mathcal{V}_s : \sigma(x_s) \in D_s$, d.h. die Abbildung ist sortentreu. Die Menge dieser Zustände wird mit $\Sigma_{\mathcal{V},D}$ oder kurz Σ bezeichnet: $\Sigma_{\mathcal{V},D} = \{\sigma : \mathcal{V} \to D\}$.

3.1 Sortenhierarchien

Die einfachsten Typisierungsansätze verwenden disjunkte Sorten, zwischen denen keine Beziehungen bestehen. Die meisten Programmiersprachen, wie auch C^{++}, unterstützen auch Hierarchien von Sorten mit Ober– und Untersorten. So lassen sich beispielsweise ganze Zahlen in positive und negative Bereiche untergliedern (siehe Abbildung 1).

Die Integration hierarchischer Beziehungen bietet für die Modellierung folgende Vorteile (vgl. [GoMe89]):

- Untersorten erlauben eine natürlichere Beschreibung vieler Problemstellungen.

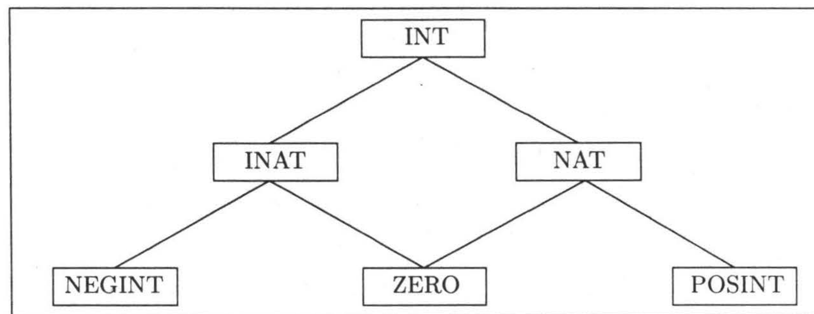

Abbildung 1: Sortenhierarchie bei ganzen Zahlen.

- Man kann partielle Funktionen total beschreiben, indem man den Definitionsbereich einer partiellen Funktion als Untersorte definiert.
- Man kann bei der Spezifikation von Funktionen die explizite Verwendung von Fallunterscheidungen oft vermeiden, indem man die Fallunterscheidungen in die Hierarchie der Sorten verlegt.

Hierbei müssen die Sorten der Prädikats– und Funktionsargumente sowie der Funktionsergebnisse festgelegt werden. Hierzu werden die Stelligkeitsfunktionen $\tau_{\mathcal{P}}$ und $\tau_{\mathcal{F}}$ einer Signatur derart erweitert, daß sie zu jedem Prädikats- bzw. Funktionssymbol die Folge der zulässigen Argumentsorten sowie ggf. die Ergebnissorte beschreiben (vgl. [HuOp80, Aven95]).

In Verbindung mit EITeL reicht diese Art der Definition jedoch nicht aus. So muß beispielsweise das Prädikat '=' für alle Terme t_1, t_2 mit der Eigenschaft $\mathcal{I}_{(\sigma,i)}(t_1), \mathcal{I}_{(\sigma,i)}(t_2) \in D_s$ für ein $s \in S$, die Terme also zu Werten der gleichen Sorte ausgewertet werden, definiert sein. Man benötigt damit ein Modell mit der Möglichkeit, Funktions- und Prädikatssymbole zu *überladen*. Dies bedeutet, daß Funktions– und Prädikatssymbole je nach Typ ihrer Argumente verschiedene Interpretationen haben können.

Ein weiterer Aspekt der Integration des Überladungskonzepts liegt in der Zielsetzung, ExTeLL und damit EITeL–Formeln in C^{++} zu integrieren, welches solche Funktionsdefinitionen unterstützt (vgl. [Stro92]). Dabei sind nicht nur die Argumenttypen zur Identifizierung einer Funktion, sondern auch die Anzahl der Argumente relevant. Deshalb sind die Funktionen $\tau_{\mathcal{P}}$ und $\tau_{\mathcal{F}}$ einer Signatur so zu definieren, daß sie zu einem Prädikats- bzw. Funktionssymbol eine Menge von Argumentbeschreibungen definieren.

Definition 3.4 Eine *Funktionsbeschreibung* fd zu einem Funktionssymbol $f \in \mathcal{F}$ ist eine Folge von Sorten $s_1, ..., s_n, s$. Dabei repräsentieren n die Stelligkeit, s_i für $i = 1...n$ die Argumentsorten und s die Ergebnissorte der Funktion f. Im folgenden wird hierfür auch die Schreibweise $f : s_1, ..., s_n \rightarrow s$ verwendet.

Definition 3.5 Eine *Prädikatsbeschreibung pd* zu einem Prädikatssymbol $p \in \mathcal{P}$ ist eine Folge von Sorten $s_1, ..., s_n$. Dabei repräsentieren n die Stelligkeit und s_i für $i = 1...n$ die Argumentsorten des Prädikats p. Im folgenden wird hierfür auch die Schreibweise $p : s_1, ..., s_n$ verwendet.

Die Einführung von Sortenhierarchien geschieht durch Aufnahme von Untersortendeklarationen in die Signatur.

Definition 3.6 Eine *Signatur* sig_{sh} zu einem Alphabet $\mathcal{A}$ ist bezüglich eines hierarchischen Sortensystems ist ein Sechs–Tupel $(\mathcal{F}, \mathcal{P}, \mathcal{S}, \tau_{\mathcal{F}}, \tau_{\mathcal{P}}, \blacktriangleleft)$ mit:

- $\mathcal{F}$ ist Menge der Funktions-, $\mathcal{P}$ die Menge der Prädikatssymbole und $\mathcal{S}$ die Menge der Sorten aus $\mathcal{A}$.

- $\blacktriangleleft$ ist eine abzählbare Menge von *Untersortendeklarationen* der Form $s < s'$ mit $s, s' \in \mathcal{S}$, s ist dabei Untersorte von s'. Ist $s < s' \in\blacktriangleleft$, so ist auch $list_s < list_{s'} \in\blacktriangleleft$.

- $\tau_{\mathcal{F}}$ ist eine Funktion $\mathcal{F} \longrightarrow Pot(\mathcal{S}^+)$, wobei $Pot(\mathcal{S}^+)$ die Potenzmenge über $\mathcal{S}^+$, und $\mathcal{S}^+$ die Menge der nicht leeren Folgen von Elementen aus $\mathcal{S}$ darstellt. Die Menge $\tau_{\mathcal{F}}(f) = \{fd_1, ...\} = \{(s_{1_1}, ..., s_{1_n}, s_1), ...\}$ ist die Menge der Funktionsbeschreibungen zu dem Funktionssymbol f.
  Weiterhin gelte $\tau_{\mathcal{F}}(f) \neq \emptyset$ für alle $f \in \mathcal{F}$.
  Dies bedeutet: Liegen $fd_i = (s_{i_1}, ..., s_{i_n}, s_i)$ und $fd_j = (s_{j_1}, ..., s_{j_m}, s_j)$ in $\tau_{\mathcal{F}}(f)$ für ein $f \in \mathcal{F}$, $i \neq j$, so ist $n \neq m$ oder es gibt ein $k \leq min(n, m)$ mit $s_{i_k} \neq s_{j_k}$, d.h. je zwei Funktionsbeschreibungen eines Funktionssymbols unterscheiden sich entweder in der Anzahl der Argumente oder in mindestens einem Argumenttyp.

- $\tau_{\mathcal{P}}$ ist eine Funktion $\mathcal{P} \longrightarrow Pot(\mathcal{S}^*)$, wobei $\mathcal{S}^*$ die Menge der Folgen aus $\mathcal{S}$ darstellt. Die Menge $\tau_{\mathcal{P}}(p) = \{pd_1, ...\} = \{(s_{1_1}, ..., s_{1_n}), ...\}$ ist die Menge der Prädikatsbeschreibungen zu dem Prädikatssymbol p. Weiterhin gelte $\tau_{\mathcal{P}}(p) \neq \emptyset$ für alle $p \in \mathcal{P}$.
  Dies bedeutet: Liegen $pd_i = (s_{i_1}, ..., s_{i_n})$ und $pd_j = (s_{j_1}, ..., s_{j_m})$ in $\tau_{\mathcal{P}}(p)$ für ein $p \in \mathcal{P}$, $i \neq j$, so ist $n \neq m$ oder es gibt ein $k \leq min(n, m)$ mit $s_{i_k} \neq s_{j_k}$, d.h. je zwei Prädikatsbeschreibungen eines Prädikatssymbols unterscheiden sich entweder in der Anzahl der Argumente oder in mindestens einem Argumenttyp.
  Des weiteren sei $\tau_{\mathcal{P}}(true) = \tau_{\mathcal{P}}(start) = \tau_{\mathcal{P}}(more) = \{\varepsilon\}$ und $\tau_{\mathcal{P}}(=) = \bigcup_{s \in \mathcal{S}} \{(s, s)\}$, wobei ε die leere Folge repräsentiert.

Für den Umgang mit Untersortendeklarationen ist es hilfreich, den transitiven Abschluß von $\blacktriangleleft$ zu betrachten:

Definition 3.7 Eine 2–stellige Relation $\lesssim$ auf einer Menge M ist eine *Quasiordnung*, falls $\lesssim$ reflexiv und transitiv ist. Der strikte Anteil $<$ und die Äquivalenzrelation $\approx$ zu $\lesssim$ sind dann definiert durch:

- $u \approx v$ gdw. $u \lesssim v$ und $v \lesssim u$

- $u < v$ gdw. $u \lesssim v$ und nicht $u \approx v$

Definition 3.8 Sei $sig_{sh} = (\mathcal{F}, \mathcal{P}, \mathcal{S}, \tau_{\mathcal{F}}, \tau_{\mathcal{P}}, \blacktriangleleft)$ eine Signatur. Dann sei $\lesssim_{sig_{sh}}$ die kleinste Quasiordnung mit der Eigenschaft $s < s' \in\blacktriangleleft$ impliziert $s \lesssim_{sig_{sh}} s'$. Man schreibt auch $s \lesssim s'$, wenn die Signatur bekannt ist.

Ein großes Problem im Zusammenhang mit Sortenhierarchien und überladenen Funktionen ist, daß manche Terme der Form $f(t_1, ..., t_n)$ nicht mehr eindeutig einer Funktionsbeschreibung aus $\tau_{\mathcal{F}}(f) = \{fd_1, ..., fd_j\}$ zugeordnet werden können, so daß die Interpretation nicht mehr eindeutig ist. Das analoge Problem gilt natürlich auch für atomare Formeln.

Beispiel: Sei sig_{sh} eine Signatur mit der Sortenhierarchie aus Abbildung 1. Weiterhin seien $-1, 1, 0, f$ Funktionssymbole aus $\mathcal{F}$ mit

$$\begin{aligned}
\tau_{\mathcal{F}}(-1) &= \{(\rightarrow NEGINT)\},\ \tau_{\mathcal{F}}(1) = \{(\rightarrow POSINT)\},\\
\tau_{\mathcal{F}}(0) &= \{(\rightarrow ZERO)\},\\
\tau_{\mathcal{F}}(f) &= \{(INAT \rightarrow INT), (NAT \rightarrow INT), (INT \rightarrow INT)\}.
\end{aligned}$$

Der Term $f(1)$ ist für jede der Funktionsbeschreibungen $(NAT \rightarrow INT)$ und $(INT \rightarrow INT)$ aus $\tau_{\mathcal{F}}(f)$ ein gültiger Term. In einem solchen Fall bietet es sich an, $(NAT \rightarrow INT)$ bei der Interpretation des Termes zu verwenden, da hier der '*Abstand*' der Argumentsorten zu den Sorten der Funktionsbeschreibungen kürzer ist.
Bei dem Term $f(0)$ kann die Funktionsbeschreibung $(INT \rightarrow INT)$ aufgrund des größeren Abstands ebenfalls vernachlässigt werden. Die Sortenabstände zu den beiden anderen Funktionsbeschreibungen sind jedoch gleich. Da dadurch bei der Interpretation solcher Terme keine eindeutige Zuordnung möglich ist, werden solche Terme verboten.

Definition 3.9 *Abstände bei Sortenhierarchien*

- Sei $s \lesssim s'$. Der *Abstand* A_{sh} *zwischen* s *und* s' ist wie folgt definiert:
  - $A_{sh}(s, s') = 0$, falls $s \approx s'$
  - $A_{sh}(s, s') = 1$, falls $s < s' \in\blacktriangleleft$
  - $A_{sh}(s, s') = n + 1$, falls $s \not\approx s'$, $s < s' \notin\blacktriangleleft$ und es existieren $s_1, ..., s_n, n > 0$, mit $s < s_1 \in\blacktriangleleft, s_i < s_{i+1} \in\blacktriangleleft, i = 1...(n-1)$ und $s_n < s' \in\blacktriangleleft$. Weiterhin sei $s_1, ..., s_n$ die bezüglich der Länge minimale Folge mit dieser Eigenschaft.
- Sei f ein Funktionssymbol, $(s_1, ..., s_n, s)$ eine Funktionsbeschreibung zu f und $t_1, ..., t_n$ Terme der Sorten $(s_1', ..., s_n')$ mit $s_i' \lesssim s_i$, $i = 1...n$.
  Der Abstand A_{sh} zwischen $f(t_1, ..., t_n)$ und der Funktionsbeschreibung $(s_1, ..., s_n, s)$ ist $min\{A_{sh}(s_i, s_i') \mid i = 1...n\}$, also der minimale Abstand zwischen den Sorten der Terme und denen der Funktionsbeschreibung.[3] Für Prädikatssymbole und deren Beschreibungen ist der Abstand analog definiert.

[3]Alternativ zu dieser Definition mit minimalem Abstand hätte man auch beispielsweise den durchschnittlichen Abstand verwenden können. Im Hinblick auf die Integration von ExTeLL in C^{++}, welches nach [Stro92] die minimalen Abstände bei der Zuordnung von Funktionen verwendet, wurde diese Variante gewählt.

3.2 Funktionale Sortenbeziehungen

Sortenhierarchien bieten in Verbindung mit überladenen Funktions- und Prädikatssymbolen flexible Möglichkeiten zur Spezifikation vieler Problemstellungen. Objektorientierte Programmiersprachen ermöglichen dabei durch Klassenableitungen diese Sortenhierarchie beliebig zu erweitern. Die Programmiersprache C^{++} geht, wie auch andere Programmiersprachen, jedoch einen Schritt weiter, indem sie die Beziehungen bei (benutzerdefinierten) Sorten frei definierbar macht. So ist es durch Definition von Konstruktoren, Zuweisungs- und Cast-Operatoren möglich, benutzerdefinierte Sorten implizit ineinander zu überführen. Hierbei können beliebige selbstdefinierte Konvertierungsfunktionen eingesetzt werden.

Der Vorteil dieses auf Konvertierungsfunktionen basierenden Ansatzes ist, daß auch artfremde Sorten in Beziehung gesetzt werden können. Der Domain einer untergeordneten Sorte muß also nicht mehr eine Teilmenge der Basissorte sein, die beiden Sorten können sogar völlig disjunkt sein. Um ExTeLL in C^{++} zu integrieren, müssen diese Möglichkeiten in die Logik integriert werden. Dies bedingt einen neuartigen Typisierungsansatz, der diese funktionalen Zusammenhänge innerhalb der Logik berücksichtigt.

Der funktionale Konvertierungsmechanismus ist im Gegensatz zu einem auf einer Sortenhierarchie basierenden Ansatz *nicht* transitiv. Eine Sorte kann dadurch nur als Untersorte einer direkt verbundenen Sorte angesehen werden. Hierdurch können Terme einer Sorte s nur dann als Argumente verwendet werden, wenn zur Argumentsorte eine direkte Verbindung besteht. Um diesen Mechanismus in EITeL zu integrieren, müssen zuerst Konvertierungsfunktionen in die Signatur aufgenommen werden.

Definition 3.10 Eine *Signatur* sig_{fs} zu einem Alphabet $\mathcal{A}$ ist bezüglich einer funktionalen Sortenbeziehung ist ein Sieben-Tupel $(\mathcal{F}, \mathcal{P}, \mathcal{S}, \tau_{\mathcal{F}}, \tau_{\mathcal{P}}, \blacktriangleleft, \mathcal{K})$, wobei

- $\mathcal{F}, \mathcal{P}, \mathcal{S}, \tau_{\mathcal{F}}, \tau_{\mathcal{P}}, \blacktriangleleft$ die selbe Bedeutung wie in Definition 3.6 haben,
- $\mathcal{K}$ eine endliche Menge von *Konvertierungsfunktionen* zwischen konvertierbaren Sorten der Form $k : D_s \to D_{s'}$ mit $s \neq s', s, s' \in \mathcal{S}$ und D_s ist der Domain der Sorte s ist.

Durch die Einführung dieser funktionalen Sortenzusammenhänge tritt das Problem der eindeutigen Zuordnung von Termen der Form $f(t_1, ..., t_n)$ zu einer Funktionsbeschreibung bzw. von Prädikatssymbolen zu ihrer Prädikatsbeschreibung verstärkt auf. Nach [Stro92] haben dabei Beziehungen, die auf einer hierarchischen Zuordnung beruhen, eine höhere Priorität. Beziehungen, die über Konvertierungsfunktionen bestehen, können die hierarchische Beziehung zur Konvertierung der Parametersorten und der Sorten der Ergebnisse der Konvertierungsfunktionen einbeziehen, so daß Kombinationen der beiden Beziehungsarten zu berücksichtigen sind. Dies muß nun in die Abstandsdefinition integriert werden.

Definition 3.11 *Abstände bei funktionalen Sortenbeziehungen*

- Der *Abstand* A_{fs} *zwischen zwei Sorten* s *und* s' ist ein Paar (a_1, a_2) von Werten gemäß

$$A_{fs}(s,s^{'}) := \begin{cases} (0, A_{sh}(s,s^{'})), & \text{falls } s \lesssim s^{'} \\ (A_{sh}(s,s_1)+1, A_{sh}(s_2,s^{'})), & \text{falls } k : D_{s_1} \rightarrow D_{s_2} \in \mathcal{K} \text{ mit} \\ & s \lesssim s_1 \text{ und } s_2 \lesssim s^{'} \\ \textit{undefiniert}, & \text{sonst} \end{cases}$$

Dabei sind die Paare lexikographisch unter Zugrundelegung von $<$ auf $\mathbb{N}_0$ sortiert, d.h. daß $(a_1, a_2) < (b_1, b_2)$ gdw.

- $a_1 < b_1$ oder
- $a_1 = b_1$ und $a_2 < b_2$.

- Seien f ein Funktionssymbol, $(s_1, ..., s_n, s)$ eine Funktionsbeschreibung zu f und $t_1, ..., t_n$ Terme der Sorten $(s_1^{'}, ..., s_n^{'})$ derart, daß $A_{fs}(s_i^{'}s_i)$ für $i = 1...n$ definiert ist.
  Der Abstand A_{fs} zwischen $f(t_1, ..., t_n)$ und der Funktionsbeschreibung $(s_1, ..., s_n, s)$ ist $min\{A_{fs}(s_i^{'}, s_i) \mid i = 1...n\}$, also der minimale Abstand zwischen den Sorten der Terme und denen der Funktionsbeschreibung. Analoges gilt für Abstände bzgl. Prädikatsbeschreibungen.

Definition 3.12 *Terme* werden bezüglich einer Signatur sig_{fs} und eines Alphabets $\mathcal{A}$ induktiv definiert durch:

- Jede Variable $x \in \mathcal{V}_s$, für ein $s \in \mathcal{S}$, ist ein Term der Sorte s.
- Ist $f \in \mathcal{F}$ ein Funktionssymbol, $\tau_{\mathcal{F}}(f) = \{fd_1, ..., fd_n\}$ und sind $t_1, ..., t_n$ Terme der Sorten $s_1, ..., s_n$, so ist $f(t_1, ..., t_n)$ ein Term der Sorte s, falls es ein i mit $1 \leq i \leq n$ gibt derart, daß
  - $fd_i = (s_{i_1}, ..., s_{i_n}, s_i)$ (passende Stelligkeit),
  - der Abstand $A_{fs}(s_j, s_{i_j})$ für jedes $j = 1...n$ definiert ist (passende Argumenttypen),
  - für alle $k \neq i$ mit $fd_k = (s_{k_1}, ..., s_{k_n}, s_k)$ und die Abstände $A_{fs}(s_j, s_{k_j})$ für $j = 1...n$ definiert sind, gilt: $A_{fs}(f(t_1, ..., t_n), fd_i) < A_{fs}(f(t_1, ..., t_n), fd_k)$ (minimaler Abstand bzgl. aller anderen Funktionsbeschreibungen mit passender Stelligkeit und Argumenttypen).
- Ist t ein Term der Sorte s, so sind $\circ t$ und $\circ_p t$ Terme der Sorte s.
- *Listenkonstruktion:* Seien $t_1, ..., t_n$ Terme der Sorten $s_1, ..., s_n$, n>0. Sei $s_i^{'} = s_i$ falls $s_i \in \mathcal{S}_B$ bzw. $s_i^{'} = s_l$ falls s_i Sorte der Form $list_{s_l}, i = 1...n$. (Nur die Basissorten von Listen sind für die Sorte der Ergebnisliste relevant.)
  Existiert $s \in \mathcal{S}$ derart, daß $A_{fs}(s_i^{'}, s)$ für $i = 1...n$ definiert ist und $A_{fs}(s_i^{'}, s^{'})$ für alle $s^{'} \neq s$ definiert ist gilt
  $min\{A_{fs}(s_i^{'}, s) \mid i = 1...n\} < min\{A_{fs}(s_i^{'}, s^{'}) \mid i = 1...n\}$,
  so ist $\langle t_1, ..., t_n \rangle$ ein Term der Sorte $list_s$.
- *Listenlänge*: Ist t ein Term der Sorte $list_s$, so ist $\mid t \mid$ ein Term der Sorte $\mathbb{N}_0$.

- *Indizierung*: Ist t ein Term der Sorte $list_s$ und sind $t_1, ..., t_n$ Terme der Sorten $(s_1, ..., s_n)$ für die $A_{fs}(s_i, \mathbb{N}_0), 1 \leq n \leq i$, definiert ist, so ist $t(t_1, ..., t_n)$ ein Term. Dieser hat die Sorte s, falls ein einfacher Wert indiziert wird bzw. $list_s$, falls eine Subliste indiziert wird.[4]

Wie bisher bezeichnet die Menge $TERM_{fs}(\mathcal{F}, \mathcal{V})$ die Menge aller nach obiger Definition bildbaren Terme.

Definition 3.13 Ist $p \in \mathcal{P}$ ein Prädikatssymbol, $\tau_{\mathcal{P}}(p) = \{pd_1, ..., pd_n\}$ und sind $t_1, ..., t_n$ Terme der Sorten $s_1, ..., s_n$, so ist $p(t_1, ..., t_n)$ eine *atomare Formel*, falls es ein i mit $1 \leq i \leq n$ gibt mit

- $pd_i = (s_{i_1}, ..., s_{i_n})$,
- $A_{fs}(s_j, s_{i_j})$ ist für $j = 1...n$ definiert,
- Für alle $k \neq i$ mit $pd_k = (s_{k_1}, ..., s_{k_n})$ und $A_{fs}(s_j, s_{k_j})$ ist für $j = 1...n$ definiert gilt $A_{fs}(p(t_1, ..., t_n), pd_i) < A_{fs}(p(t_1, ..., t_n), pd_k)$.

Durch die Möglichkeit, auch Terme einer untergeordneten Sorte als Argumente von Funktions- und Prädikatssymbolen zu verwenden, muß dies auch bei der Präinterpretation berücksichtigt werden.

Definition 3.14 Eine *Präinterpretation* P_{fs} bezüglich einer Signatur für ein Sortensystem mit funktionalen Sortenbeziehungen ist ein Trippel $P = (D, \mathcal{I}_{\mathcal{F}}, \mathcal{I}_{\mathcal{P}})$ mit:

- $D = \bigcup_{s \in \mathcal{S}} D_s$ ist das Universum der Interpretation mit $D_s \subseteq D_{s'}$ für $s \lesssim s'$.
- $\mathcal{I}_{\mathcal{F}}$ ist eine Abbildung, die jedem Funktionssymbol $f \in \mathcal{F}$ mit der Eigenschaft $\tau_{\mathcal{F}}(f) = \{fd_1, ..., fd_n\}$ und jeder Sortenfolge $s_1, ..., s_n$, für die ein Abstand zu einer Funktionsbeschreibung definiert ist und für die es eine Funktionsbeschreibung $fd_i = (s_{i_1}, ..., s_{i_n}, s_i)$ mit minimalem Abstand gibt, eine Abbildung der Eigenschaft
  $\mathcal{I}_{\mathcal{F}}(f, (s_1, ..., s_n)) : D_{s_{i_1}} \times ... \times D_{s_{i_n}} \rightarrow D_{s_i}$ zuordnet.
- $\mathcal{I}_{\mathcal{P}}$ ist eine Abbildung, die jedem Prädikatssymbol $p \in \mathcal{P}$ mit $\tau_{\mathcal{P}}(p) = \{pd_1, ..., pd_n\}$ und jeder Sortenfolge $s_1, ..., s_n$, für die ein Abstand zu einer Prädikatsbeschreibung definiert ist und für die es eine Prädikatsbeschreibung

[4]Man erkennt hier, daß beliebig geschachtelte Listen in Verbindung mit streng typisierten Ansätzen Probleme bereiten. Bei der Indizierung ist es nicht möglich, anhand des syntaktischen Aufbaus eines Terms dessen Sorte zu bestimmen. Wie man bei der Sprachintegration sieht, bereitet dies in Verbindung mit C++ Einbettungsprobleme. Ein möglicher Lösungsansatz wäre es, die Schachtelungstiefe in die Sortendefinition zu integrieren. Für die Integration in C++ könnte dann komplett auf die Listenintegration verzichtet werden, da C++ für solch konkret aufgebaute Listen die Verwaltungsmöglichkeiten ohnehin beinhaltet. Damit wird jedoch die Möglichkeit zum Aufbau von Listen abhängig von der Umgebungssituation erschwert. Da dies eine elementare Möglichkeit der deklarativen Sprache ExTeLL ist, wird hier für die Indizierung eine dynamische Typisierung verwendet, in der Integration muß jedoch eine zusätzliche Zugriffsschicht entwickelt werden, damit auch C++-Konstrukte Listeninhalte verarbeiten können.

$pd_i = (s_{i_1}, ..., s_{i_n})$ mit minimalem Abstand gibt, eine Abbildung
$\mathcal{I}_\mathcal{P}(p, (s_1, ..., s_n)) : D_{s_{i_1}} \times ... \times D_{s_{i_n}} \rightarrow \{true, false\}$ zuordnet.

Wie schon oben erwähnt, müssen anders als bei Sortenhierarchien bei funktionalen Beziehungen die Domains der Sorten nicht mehr Teilmengen einer übergeordneten Sorte sein. Die bisherige Interpretation des Prädikates '=' auf Intervallen vergleicht die Interpretation der einzelnen Terme. Durch die fehlende Domainbeziehungen können nun die Interpretationen der Terme unvergleichbar sein, so daß Definition 2.10 der Interpretation von '=' auf die Möglichkeiten der funktionalen Beziehungen angepaßt werden muß. Weiterhin müssen bei der Nutzung von Funktions- und Prädikatssymbolen eventuelle Konvertierungsfunktionen die Parameter in den passenden Domain überführen.

Definition 3.15 Sei $\mathcal{I}_{fs} = (P_{fs}, \Im)$ eine Interpretation einer Signatur $sig_{fs} = (\mathcal{F}, \mathcal{P}, \mathcal{S}, \tau_\mathcal{F}, \tau_\mathcal{P}, \blacktriangleleft, \mathcal{K})$ mit $P_{fs} = (D, \mathcal{I}_\mathcal{F}, \mathcal{I}_\mathcal{P})$ und $\Im = (\sigma, i) = (\sigma_0, ..., \sigma_{|\sigma|}) \in \Sigma^+ \times \mathbb{N}_0$.
Die zugehörige *Interpretationsabbildung* $\mathcal{I}_{(\sigma,i)}$ ist ist in Verbindung mit funktionalen Sortenbeziehungen induktiv mit den folgenden Eigenschaften definiert:

a) (Variableninterpretation) wie in Definition 2.10.

b) Seien $f \in \mathcal{F}$ ein Funktionssymbol und $t_1, ..., t_n$ Terme der Sorten $s_1, ..., s_n$ und $f(t_1, ..., t_n)$ ein in Verbindung mit $fd \in \tau_\mathcal{F}(f)$ korrekter Term, $fd = (s_1', ..., s_n', s')$, so ist $\mathcal{I}_{(\sigma,i)}(f(t_1, ..., t_n)) := \mathcal{I}_\mathcal{F}(f, (s_1, ..., s_n))(c_1, ..., c_n)$, wobei für $j = 1...n$
$c_j = \mathcal{I}_{(\sigma,i)}(t_j)$, falls $s_i \lesssim s_i'$ bzw.
$c_j = k(\mathcal{I}_{(\sigma,i)}(t_j))$, falls s_i, s_i' über die Konvertierungsfunktion $k \in \mathcal{K}$ in Beziehung stehen.

c) , d) (next, prev) wie in Definition 2.10.

e) Seien $t_0, ..., t_n$ Terme der Sorten $s_0, ..., s_n$ und $\langle t_0, ..., t_n \rangle$ ein in Verbindung mit der gemeinsamen Basissorte s korrekter Term, so ist
$\mathcal{I}_{(\sigma,i)}(\langle t_0, ..., t_n \rangle) := \langle c_0, ..., c_n \rangle$, wobei für $j = 0...n$
$c_j = \mathcal{I}_{(\sigma,i)}(t_j)$, falls $s_i \lesssim s$ bzw.
$c_j = k(\mathcal{I}_{(\sigma,i)}(t_j))$, falls s_i, s über die Konvertierungsfunktion $k \in \mathcal{K}$ in Beziehung stehen.

f) Seien $t, t_1, ..., t_n$ Terme der Sorten $list_s, s_1, ..., s_n$ und $t(t_1, ..., t_n)$ ein korrekter Term, so ist $\mathcal{I}_{(\sigma,i)}(t(t_1, ..., t_n)) := \mathcal{I}_{(\sigma,i)}(t)_{c_1,...,c_n}$, wobei für $j = 1...n$
$c_j = \mathcal{I}_{(\sigma,i)}(t_j)$, falls $s_i \lesssim \mathbb{N}_0$ bzw.
$c_j = k(\mathcal{I}_{(\sigma,i)}(t_j))$, falls $s_i, \mathbb{N}_0$ über die Konvertierungsfunktion $k \in \mathcal{K}$ in Beziehung stehen.

g) (Listenlänge) wie in Definition 2.10.

h) (*true, more, start*) wie in Definition 2.10.

i) Seien $p \in \mathcal{P}$ ein Prädikatssymbol und $t_1, ..., t_n$ Terme der Sorten $s_1, ..., s_n$ und $p(t_1, ..., t_n)$ eine in Verbindung mit $pd \in \tau_{\mathcal{P}}(p)$, $pd = (s_1', ..., s_n')$, korrekte Atomformel, so ist $\mathcal{I}_{(\sigma,i)}(p(t_1, ..., t_n)) := \mathcal{I}_{\mathcal{P}}(p, (s_1, ..., s_n))(c_1, ..., c_n)$, wobei für $j = 1...n$
$c_j = \mathcal{I}_{(\sigma,i)}(t_j)$, falls $s_i \lesssim s_i'$ bzw.
$c_j = k(\mathcal{I}_{(\sigma,i)}(t_j))$, falls s_i, s_i' über die Konvertierungsfunktion $k \in \mathcal{K}$ in Beziehung stehen.

j) Seien t_1 und t_2 Terme der Sorten s_1 und s_2, so ist

$$\mathcal{I}_{(\sigma,i)}(t_1 = t_2) := \begin{cases} \text{true}, & \text{falls } A_{fs}(s_1, s_2) = (a_1, a_2) \text{ oder } A_{fs}(s_2, s_1) = (a_1, a_2) \\ & \text{definiert ist und} \\ & \quad a_1 = 0 \text{ und } \mathcal{I}_{(\sigma,i)}(t_1) = \mathcal{I}_{(\sigma,i)}(t_2) \text{ oder} \\ & \quad A_{fs} \text{ ist mit } k \in \mathcal{K} \text{ definiert und} \\ & \quad\quad k(\mathcal{I}_{(\sigma,i)}(t_1)) = \mathcal{I}_{(\sigma,i)}(t_2) \text{ bzw.} \\ & \quad\quad \mathcal{I}_{(\sigma,i)}(t_1) = k(\mathcal{I}_{(\sigma,i)}(t_2)) \\ \text{false}, & \text{sonst} \end{cases}$$

k) – r) (temporale Operatoren) wie in Definition 2.10.

Die oben aufgeführten Möglichkeiten, Sorten zueinander in Beziehung zu setzen, stellen jeweils Erweiterungsschritte dar. Zulässige Formeln für Signaturen mit disjunkten Sorten sind auch zulässige Formeln bei Signaturen mit funktionalen Sortenbeziehungen. Im folgenden wird, obwohl für die reine ExTeLL^T-Umgebung hierarchische Sortenbeziehungen verwendet werden, immer von funktionalen Sortenbeziehungen ausgegangen, wobei die Komponeneten $\blacktriangleleft$ und $\mathcal{K}$ in der Signatur auch leer sein können.

4 Von EITeL^T zur Programmiersprache ExTeLL^T

Auf Basis der erweiterten Intervall–Temporal–Logik EITeL und abgeleiteten Konstrukten der Grundoperatoren entstand die Programmiersprache ExTeLL (vgl. [Kir94, Spi95]). Im folgenden wird die Spraache ExTeLL^T vorgestellt, die diesen Ansatz um die in Kapitel 3 eingeführte Typisierung mit Sortenhierarchien erweitert. Dabei werden die Grundtypen *string, bool, char, int* und *float* mit $float < int < char$ sowie als strukturierter Datentyp Listen integriert. Ebenfalls wird die Möglichkeit geschaffen, Funktionen und benutzerdefinierbare Prädikate zu überladen.

Da eine vollständige Beschreibung von ExTeLL^T zu umfangreich ist, werden nur die Grundelemente und deren Aufbau beschrieben, um damit einen Überblick über die Sprache zu ermöglichen. Nur die im Vergleich zur untypisierten Sprache ExTeLL hinzugekommenen Elemente werden ausführlicher vorgestellt. Eine vollständige Darstellung der Sprache ExTeLL findet sich in [Kir94, Spi95].

ExTeLL^T–Programme sind syntaktisch EITeL^T–Formeln, die Variablenbelegungen beschreiben. Ziel der Programmausführung ist es, diese Variablenbelegungen zu bestimmen. Die Syntax von ExTeLL^T ist nun so ausgelegt, daß sich aus diesen Formeln

mittels Rekurrenzgleichungen Variablenbelegungen konstruieren lassen. Dabei wird durch sukzessive Anwendung der entsprechenden Rekurrenzgleichungen eine Formel so lange transformiert, bis man entweder das minimale Modell anhand der entstandenen Normalform ermitteln kann oder bis die Unerfüllbarkeit der Formel festgestellt wird.

ExTeLL^T unterscheidet zwischen *Formeln* und *Ausdrücken*: ExTeLL^T-Formeln sind EITeL^T-Formeln, die zur Ermittlung der Variablenbelegung eines minimalen Modells genutzt werden. ExTeLL^T-Ausdrücke sind dagegen EITeL^T-Terme oder EITeL^T-Formeln, die als Boole'sche Funktionen aufgefaßt werden. Diese beschreiben eine Berechnungsvorschrift, die zur Ermittlung konkreter Werte dienen und dabei Variablenbelegungen zur Auswertung nutzen, jedoch nicht festlegen. Die Auswertung eines Ausdrucks ist also nur möglich, wenn die Variablen innerhalb des Ausdrucks schon durch Formeln belegt sind.

Man erkennt, daß die Begriffe Formeln und Ausdrücke in ExTeLL^T allgemeiner definiert sind als in EITeL^T. Dies erlaubt eine flexiblere Ausdrucksbehandlung. So sind als ExTeLL^T-Ausdrücke EITeL^T-Formeln wie $(x > 1)$ zulässig, da sich aus einer bereits vorhandenen Belegung der Variablen x feststellen läßt, ob die Formel erfüllt ist. Andererseits ist eine solche EITeL^T-Formel nicht zulässig als ExTeLL^T-Formel, da sich aus ihr keine Belegung der Variablen x ableiten läßt. Aus dem gleichen Grund ist in ExTeLL^T die Negation auf Formelebene nicht möglich.

4.1 Ausdrücke

Ausdrücke sind wie üblich mittels der zugelassenen Operanden und Operatoren rekursiv aufgebaut und definiert. Die Auswertung liefert Konstanten, wobei der jeweilige Ergebnistyp eines Ausdrucks (bis auf die Indizierung von Listen) durch den Aufbau des Ausdrucks eindeutig definiert ist (vgl. Kapitel 3).

Innerhalb von Ausdrücken können dabei Konstanten vom Typ *string*, *bool*, *char*, *int* und *float* verwendet werden. Weiterhin stehen die wichtigsten arithmetischen (+,-,*,/), Boole'schen (and, or, impl, equi) und String-Operatoren (Konkatenation) zur Verfügung. Eine Besonderheit in ExTeLL^T sind die Quantifizierungs-Operatoren *for-all, for-one*:

(for-all / for-one <Variable> from <Int1> to <Int2> holds <Bedingung>

Bei der Allquantifizierung *for-all* gilt der gesamte Ausdruck als erfüllt, wenn die Bedingung für jede Belegung gilt, die Existenzquantifizierung überprüft die Gültigkeit bei mindestens einer Substitution. Beide Konstrukte werden häufig im Zusammenhang mit Listen eingesetzt.

Des weiteren gibt es die Möglichkeit, temporale, also zeitübergreifende Berechnungsvorschriften innerhalb von Ausdrücken zu definieren. Da die Auswertung eines Ausdrucks nur möglich ist, wenn alle darin auftretenden belegt sind, können hier nur vergangenheitsorientierte Operatoren eingesetzt werden. Der Operator *pal* dient dabei zum Test, ob das Argument in allen bisherigen Zeitpunkten erfüllt war, *pst* testet,

ob mindestens eines Auswertung des Arguments in der Vergangenheit. ein positives Resultat liefert. Bei beiden Operatoren wird der aktuelle Zeitpunkt in den Zeitraum der Untersuchung mit eingeschlossen, das Ergebnis ist vom Typ *bool*. Direkte Zugriffe auf Variablenwerte in der Vergangenheit können mit den Operatoren *prev* und *weakprev* durchgeführt werden. Dabei ist der Wert einer Anwendung von *prev* nur definiert, wenn der aktuelle Zeitpunkt (Zustand) nicht der erste im zu ermittelnden Intervall ist, ein vorhergehender Zustand also tatsächlich existiert. Im Gegensatz hierzu kann *weakprev* auch im Anfangszustand 0 ausgewertet werden.

4.2 Formeln

Im Unterschied zu Ausdrücken erlauben Formeln eine Belegung der Variablen im Speicher sowie die Bestimmung der Länge des Ergebnisintervalls. Der Großteil der Sprachkonstrukte dient dabei zur Formulierung der zeitlichen Zusammenhänge zwischen Aussagen.

4.2.1 Typisierung von Variablen

Im Gegensatz zu ExTeLL müssen in ExTeLL^T alle Variablen typisiert werden, bevor ihnen ein Wert zugewiesen werden kann. Dabei gelten Typangaben für eine Variable immer für den gesamten Berechnungszeitraum. Es ist nicht möglich, den Datentyp einer Variablen in einem weiteren Zeitpunkt zu ändern. Der Zeitpunkt, an dem eine Variable typisiert wird, ist nicht von Bedeutung, Typangaben wirken auch rückwirkend in schon berechnete Intervalle. In ExTeLL^T werden die Grunddatentypen *string*, *bool*, *char*, *int* und *float* mit $float < int < char$, sowie Listen unterstützt.

Syntax

<typedef> = '(' ['**list**'] <typename> <simple var> ')'

<typename> = '**string**' | '**bool**' | '**char**' | '**int**' | '**float**'

Dabei werden diese Typprädikate als Typzuweisungen an die Variablen, die in EITeL^T implizit typisiert sind, angesehen. Die Interpretation der Typprädikate als Typzuweisung bedingt jedoch eine im Vergleich zu den sonstigen Prädikaten von ExTeLL^T völlig andere Handhabung. Da Zuweisungen erst nach Zuordnung einer Variablen mit einem Typ durchgeführt werden können, würde der Transformationsmechanismus bei der Behandlung der Prädikate zum Zeitpunkt der Auswertung die in [Kir94] dargestellte Vollständigkeit einschränken. Deshalb werden Typprädikate unmittelbar beim Parsen der Formel ausgewertet, so daß die Typen aller Variablen vor der eigentlichen Formelauswertung bekannt sind.[5] Dadurch können auch untypisierte Variablen schon

[5] Diese abweichende Behandlung der Typprädikate soll auch durch die abweichende Syntax verdeutlicht werden, die bei der Typisierung keine Klammerung des Arguments, aber eine Gesamtklammerung erfordert. Diese Form soll an die in imperativen Programmiersprachen verwendete Variablendeklaration erinnern. Weiterhin ist hierdurch das Schlüsselwort *list* zum Aufbau von Listen **und** für die Typisierung einsetzbar.

beim Parsen erkannt und mit einer entsprechenden Fehlermeldung abgewiesen werden. Diese Art der Behandlung steht mit der zugrundeliegenden Logik EITeL^T im Einklang, da auch dort vor der Interpretation die Typen aller Variablen definiert sind.

Weiterhin schließt diese Behandlung der Typprädikate Formeln der Form

if (Bedingung) then (Typ1 x) else (Typ2 x)

aus, wenn $Typ1$ und $Typ2$ verschieden sind. Die Zulassung solcher Formeln hätte neben dem Verlust der Vollständigkeit der Abarbeitung auch zur Folge, daß Fehler aufgrund von Typkonflikten (beispielsweise bei Zuweisungen und Vergleichen) nicht mehr zum Zeitpunkt des Parsens erkannt werden können. Auch die Zuordnung von Funktionen bei einem Funktionsaufruf könnte wegen der Möglichkeit des Überladens nicht mehr statisch erfolgen (vgl. Kapitel 3). Die Verbindung zur Logik EITeL^T, die eine Typisierung der Variablen vor der Interpretation voraussetzt, ginge ebenfalls verloren. Einer Variablen kann jedoch der gleiche Typ mehrfach zugewiesen werden.

4.2.2 Zuweisungen

Charakteristisch für Programmiersprachen mit imperativen Elementen ist die Möglichkeit der Zuweisungen von Werten an Variable. Rein imperative Sprachen sehen Variable als Platzhalter bzw. abstrakte Speicherplätze für Werte an, deren Inhalt beliebig oft geändert werden kann. Temporale Variable nehmen eine Folge von Werten auf. Der Wert einer Variablen zu einem Zeitpunkt kann dabei durch den Operator '=' belegt werden. Die logische Interpretation von '=' bedingt, daß der Variablenwert zum Zeitpunkt der Auswertung von '=' dem Ausdruck der rechten Seite entspricht. Zur Bestimmung eines Lösungsintervalls, das diese Bedingung erfüllt, ist diese Belegung erforderlich, sie kann nicht mehr verändert werden.

Der Operator '=' ist der einfachste Zuweisungsoperator. Zur Vereinfachung der Programmierung zeitübergreifender Variablenbelegungen existieren eine Reihe abgeleiteter Zuweisungsoperatoren. So dient der Operator '*gets*' zur Festlegung von Variablen für alle zukünftigen Zeitpunkte. Das vergangenheitsorientierte Gegenstück zu '*gets*' ist *pgets*, hier werden die Zuweisungen fortlaufend für alle Zustände in der Vergangenheit durchgeführt. Die Zuweisung ':=' belegt eine Variable im aktuellen Zeitpunkt und in allen folgenden Zeitpunkten mit der Auswertung des Ausdrucks auf der rechten Seite der Zuweisung, wobei der Ausdruck zum Zeitpunkt der Zuweisung ausgeführt wird. Weiterhin kann mittels des Zuweisungsoperators '<–' eine Zuweisung im letzten Zeitpunkt des Lösungsintervalls erreicht werden, wobei der zuzuweisende Ausdruck im aktuellen Zeitpunkt ausgewertet wird.

4.2.3 Formeln zur Modellängenbestimmung

Da ExTeLL^T auf endlichen Intervallen arbeitet, müssen Möglichkeiten zur Bestimmung bzw. Überprüfung von Intervallängen vorhanden sein. So testet das Prädikat '*empty*',

ob der aktuelle Zustand der letzte des Berechnungsintervalls ist, '*more*' fordert, daß noch mindestens ein und '*skip*', daß genau ein Folgezustand existiert. Mit dem Prädikat '*length*' kann über ein Integer–Argument die Länge des Restintervalls definiert bzw. getestet werden. Weiterhin existiert das Prädikat '*halt*', welches einen Boole'schen Ausdruck als Argument erwartet. Es zeigt an, ob das Intervallende erreicht und die Berechnung für das Intervall abgeschlossen ist.

4.2.4 Einstellige temporale Operatoren

Die Operatoren '*next*', '*prev*', '*always*' und '*pal*' (past–always) von EITeL stehen auch innerhalb von ExTeLL^T–Formeln zur Verfügung. Eine Formel der Gestalt '*next (form)*' ist genau dann erfüllt, wenn das Argument *form* im nächsten Zeitpunkt erfüllt ist. Auf diese Weise kann beispielsweise der Zeitpunkt, an dem eine Formel gültig sein soll, durch führende *next* festgelegt werden. Der Operator '*weaknext*'ist, wie aus dem Namen hervorgeht, ist ein abgeschwächtes '*next*'. Sein Argument muß im nächsten Zeitpunkt erfüllt sein, wenn ein solcher existiert, also durch die Abarbeitung einer anderen Teilformel gefordert wird. Der Vergangenheitsoperator '*prev*' ist das Gegenstück zu '*next*'. Er ist erfüllt, wenn sein Argument im vorherigen Zeitpunkt, der existieren muß, erfüllt ist. Der Vergangenheitsoperator zu '*weaknext*' ist '*weakprev*' und entspricht einem abgeschwächten '*prev*'. Sein Argument muß im vorherigen Zeitpunkt erfüllt sein, wenn ein solcher existiert. Im ersten Zustand eines Intervalls wird '*weakprev*' immer zu *true* ausgewertet.

Die Auswertung des Operators '*always*' ist erfüllt, wenn sein Argument im aktuellen und in allen folgenden Zeitpunkten eines Intervalls erfüllt ist. Der entsprechende Vergangenheitsoperator '*pal*' liefert den Wert *true*, wenn sein Argument im aktuellen und in allen vergangenen Zeitpunkten erfüllt ist.

Neben diesen Basisoperatoren existieren in ExTeLL^T auch abgeleitete Operatoren. So können mittels '*sometimes*' Anweisungen formuliert werden, deren Ausführungszeitpunkt in der Zukunft variabel, jedoch abhängig von der Restformel ist. Für entsprechende vergangenheitsorientierte Aussagen dient der Operator '*pst*' (past–sometimes). Formeln der Gestalt '*final (form)*' sind erfüllt, wenn das Argument im letzten Zeitpunkt des Restintervalls gilt. Damit läßt sich die Ausführung bestimmter Aktionen am Intervallende erreichen, wie z.B. die Ausgabe von Werten. Entsprechend dient der Operator '*initial*' zur Formulierung von Anweisungen, die im ersten Zeitpunkt eines Intervalls ausgeführt werden sollen.

4.2.5 Verknüpfung von Formeln

Für die Verknüpfung von Teilformeln stehen die Boole'schen Operatoren '*and*' und '*or*' sowie die Intervallverknüpfung '*chop*' zur Verfügung. Der Einsatz von '*chop*' dient dabei zur sequentiellen Komposition von Formeln, wobei sich die Teilintervalle in einem Zustand überlappen. Mit '*chop*' lassen sich also zeitlich aufeinanderfolgende Vorgänge erfassen. Der gemeinsame Zeitpunkt ermöglicht dabei die Übergabe von

Variablenbelegungen. Für die Abarbeitung eines '*chop*' ist es notwendig, daß in der linken Teilformel die Länge des Teilintervalls spezifiziert ist, um die Eindeutigkeit der Intervallteilung zu gewährleisten.

4.2.6 Schleifen

ExTeLL^T verwendet '*while*'–, '*until*'– und '*for*'–Schleifen. Damit werden die Möglichkeiten von Schleifen imperativer Programmiersprachen auf das Intervallkonzept von ExTeLL^T übertragen. Bei der Abarbeitung wird bei jedem Schleifendurchlauf ein Teilintervall ermittelt, auf dem der Rumpf der Anweisung erfüllt sein muß. Beim nächsten Schleifendurchlauf wird dann das ermittelte Intervall um ein weiteres Teilintervall erweitert, auf dem wiederum der Rumpf erfüllt sein muß. Dabei überlappen sich die jeweiligen Teilintervalle in genau einem Zeitpunkt, so daß Variablenwerte an den nächsten Schleifendurchlauf übergeben werden können. Um die Länge des durch den Rumpf zu erfüllenden Intervalls exakt bestimmen zu können, muß diese im Rumpf festgelegt werden. Alle ExTeLL^T–Schleifen lassen sich dadurch auf wiederholte Anwendung von '*chop*' zurückführen.[6]

4.2.7 Bedingte Anweisungen, Ein– und Ausgabe

Bedingte Ausdrücke, Formeln und Anweisungen formuliert man wie üblich mittels '*if-then-else*'. Für Bedingungen ohne *else*–Fall steht die Verknüpfung *impl* zur Verfügung.

Zur Erstellung interaktiver Programme benötigt man Funktionen für die Ein– und Ausgabe von Werten. Eingaben werden mittels des Operators '*request*' angefordert. Dieser erfragt interaktiv die Belegung seiner Argumente (beliebige einfache Variablen). Das Prädikat '*print*' ist das einfachste Ausgabeprädikat, es gibt *im Moment seines Aufrufes durch den Interpreter* alle Argumente (Ausdrücke) aus. Ausdrücke, die aufgrund fehlender Variablenbelegungen noch nicht auswertbar sind, werden, auch wenn die Werte der Variablen im späteren Programmablauf belegt werden, als 'undefiniert' betrachtet. Der Operator '*display*' gibt anders als '*print*' seine Argumente erst dann aus, wenn alle auswertbar sind. Intervallorientierte Ausgaben können mittels *display–seq* durchgeführt werden; Dabei wird jeweils die Folge der Werte aller Argumente vom Zeitpunkt 0 bis einschließlich des Aktuellen ausgegeben. Dabei müssen die Ausdrücke der Argumente in jedem dieser angesprochenen Zeitpunkte auswertbar sein.

4.3 Funktionen und Prozeduren

Zur Erstellung übersichtlicher und strukturierter Programme wird eine Möglichkeit benötigt, Formeln bzw. Ausdrücke zu benennen und in andere Programmstücke einzusetzen. Weiterhin ist für den universellen Einsatz solcher Teilprogramme an unterschiedli-

[6] Dieses Schleifenkonzept unterscheidet sich von dem in [Krö86] vorgestellten. Dort wird die Schleifeneintrittsbedingung in jedem folgenden Zeitpunkt, also unabhängig von der Länge des im Rumpf beschriebenen Intervalls, getestet.

chen Programmstellen eine Parameterübergabe notwendig. ExTeLL^T verwendet hierzu eine Unterprogrammtechnik die die Möglichkeit bietet, Funktionen und Prozeduren zu überladen (vgl. Kapitel 3).

5 Integrationsarten

Wie eine Sprache, die für sehr spezielle Zwecke entwickelt wurde, in prozedurale Programmiersprachen eingebettet werden kann, ist in Verbindung mit Datenbankanfragesprachen schon vielfältig untersucht ([Här87]) worden. Obwohl die Ergebnisse dieser Arbeiten aufgrund der zugleich temporalen und logischen Eigenschaften von ExTeLL nur zu einem geringen Teil verwendet werden können, bieten sie einen guten Ausgangspunkt für die Wahl der Integrationsart. Die grundlegenden Probleme bereiten dabei die Einbettung der Datenstrukturen und Sprachkonstrukte sowie die Art und Weise, wie auf die Ergebnisse von der Wirtssprache aus zugegriffen werden kann. Für die Bereitstellung der Sprachkonstrukte innerhalb der Wirtssprache bieten sich alternativ eine vollständige Integration der Sprache oder die Nutzung der einzubettenden Sprache über spezielle Funktionsaufrufe (Call-Technik). Bezüglich der Datenstrukturen ergeben sich die Möglichkeiten eines einheitlichen Typkonzepts mit sprachübergreifenden Deklarationen bzw. einer getrennten Handhabung mit expliziter Wertübergabe und Typkonvertierung bei der Kommunikation der beiden Sprachkomponenten (vgl. Abbildung 5 und [Här87]).

		Operatoren in Wirtssprache integriert	
		Nein	Ja
Datenstrukturen und Variablen in Wirtssprache integriert	Nein	Call-Technik Übergabe an Programm-variable / Pufferbereich	Direkte Spracheinbettung Übergabe an Programm-variable / Pufferbereich
	Ja	Call-Technik Übergabe in statisch zugordneten Datenstrukturen	Direkte Spracheinbettung Übergabe in statisch zugordneten Datenstrukturen

Abbildung 2: Formen der Spracheinbettung

5.1 Trennung der Operatoren und Datenstrukturen

Eine Trennung der Operatoren und Datenstrukturen entspricht der einfachsten Form der Sprachintegration, die zu integrierende und die Wirtssprache sind vollkommen getrennt. ExTeLL–Programme würden hierbei in Form einer Zeichenkette als Parameter einer Interpreter–Funktion übergeben. Über den Rückgabewert einer solchen Funktion kann dann festgestellt werden, ob die Auswertung des Programms erfolgreich war oder nicht. Der Zugriff auf durch ein ExTeLL–Programm ermittelte Variablenbelegungen geschieht in dieser Einbindungsform über weitere Funktionen, wobei aufgrund der Typisierungseigenschaften von C^{++} für jeden Typ eine eigene Zugriffsfunktion erforderlich ist. Diese Funktionen könnten in einer Bibliothek implementiert werden, zur Nutzung von ExTeLL in von C^{++} müßte folglich nur die Bibliothek eingebunden werden.

Bibliotheksimplementierungen besitzen den großen Vorteil, daß sie recht universell in verschiedenen Umgebungen genutzt werden können; aufgrund der Trennung der beiden Sprachen ist man von der Wirtssprache unabhängig. Diese Sprachintegration ist folglich innerhalb jeder Programmiersprache, die Bibliotheken einbinden kann, wie *Java*, *Basic* und *Pascal*, nutzbar, ohne daß Anpassungen am Interpreter durchgeführt werden müssen. Dies hat weiterhin zur Folge, daß kein erweiterter Compiler für die Sprachintegration erforderlich ist.

Da ExTeLL–Programme hier in Zeichenketten abgelegt sind, die erst bei der Übergabe interpretiert werden, können diese Programme vollkommen dynamisch zusammengesetzt werden. So ist es möglich, daß das zu interpretierende ExTeLL–Programm jeweils abhängig von Umgebungsbedingungen innerhalb der Wirtssprache aufgebaut werden kann. Dieser Vorteil des dynamischen Programmaufbaus zieht jedoch den größten Nachteil dieser Integrationsform nach sich. Syntaxfehler können erst zum Zeitpunkt der Übergabe an den Interpreter festgestellt werden, wodurch das Testen stark erschwert wird. Das Parsen während der Programmausführung sowie die explizite Wertübergabe der Ergebnisse nach der Interpretation wirkt sich auch auf die Performance nachteilig aus. Weiterhin verursacht das Herauskopieren der Variablenwerte für eine Auswertung einen erhöhten Programmieraufwand. Diese Integrationsform wird daher als *unnatürliche* Sprachintegration (vgl. [Här87]) angesehen.

5.2 Integration der Datenstrukturen

Voraussetzung für eine natürlichere Integration von ExTeLL^T in C^{++} ist die Bereitstellung der Datentypen, also der temporalen Variablen, in der Wirtssprache. Die auf diese Weise deklarierten Variablen müssen natürlich temporaler Natur sein und eine Folge von Werten repräsentieren. Diese Variablen müssen mit dem ExTeLL–Programm, welches hier immer noch in Stringform vorliegt, in Bezug gesetzt werden, so daß spezielle Verknüpfungsfunktionen zur Verfügung gestellt werden müssen. Hier gibt es mehrere Möglichkeiten:

(1) Das ExTeLL^T–Programm wird in Teile zerlegt, wobei es bei jedem Auftreten einer Variablen zu einer Aufspaltung kommt. Die Variablen werden dabei aus dem

Programm entfernt, so daß alle Programmteile variablenfrei sind. Im Anschluß daran werden die Teile an den ExTeLL^T–Interpreter übergeben, wobei an den ursprünglichen Variablenpositionen Referenzen auf die zu verwendenden Variablen übergeben werden. Die ExTeLL^T–Umgebung kann aus diesen Informationen das komplette Programm aufbauen. Bei der Auswertung werden die ermittelten Variablenbelegungen direkt in die übergebenen Verweise eingetragen. Diese Form der Verbindung ist jedoch mit einem erheblichen Programmieraufwand verbunden.

(2) Das ExTeLL^T–Programm wird analog zur vollständigen Trennung der Sprachen als Zeichenfolge gefolgt von Verweisen auf die in der Wirtssprache deklarierten Variablen übergeben. Dabei muß die Reihenfolge der Variablen genau der Reihenfolge des Auftretens der Variablen im übergebenen Programm entsprechen. Diese Form ist jedoch sehr fehleranfällig.

(3) Das vollständige Programm wird als Zeichenkette übergeben. Es wird jedoch eine explizite Verknüpfung zwischen einem Variablennamen innerhalb eines ExTeLL^T–Programmes und einer in der Wirtssprache deklarierten Variablen hergestellt. Hierzu kann eine zweiparametrige Funktion dienen, die als ersten Parameter den Variablennamen und als zweiten einen Verweis auf die eigentliche Variable erwartet. Bei Parsen des ExTeLL^T–Programmes werden diese Informationen zur Zuordnung von Variablen genutzt.

Die Integration der Datentypen erspart das explizite Herauskopieren der Ergebniswerte, der Programmieraufwand ist jedoch bedeutend größer als bei einer vollständigen Trennung der Sprachen. Auch hier ist keine Erweiterung des Wirtssprachen–Compilers notwendig, die Bindung an die Wirtssprache ist intensiver, da der Aufbau und damit die Implementierung der temporalen Variablen in der Wirtssprache unterschiedlich sein können. Zudem gibt es keine Möglichkeit, Typen der Wirtssprache in ExTeLL^T zu verwenden, es stehen nur die in ExTeLL^T integrierten Typen für temporale Variablen zur Verfügung. Dabei findet die Belegung der Variablen durch das ExTeLL^T–Programm als *Seiteneffekt* der Interpretation statt, wodurch eine Fehlersuche stark erschwert wird. Aus diesen Gründen wird diese Form der Sprachintegration selten gewählt.

5.3 Integration der Sprache

Eine vollständige Integration der Sprache erfordert einen modifizierten Compiler der Wirtssprache bzw. einen Präprozessor, der aus dem integrierten Programmcode Wirtssprachenquelltext erzeugt, der auch Bibliotheksfunktionen nutzt. Dies erlaubt eine Syntaxprüfung zur Übersetzungszeit, so daß für den Programmtext der integrierten Sprache ein effizienter Code erzeugt werden kann. Mit einer Integration der Datentypen ist dann die Möglichkeit einer *natürlichen* Einbettung gegeben, die eine einheitliche, syntaktisch homogene Gesamtsprache darstellt. Hierbei sind die Eigenschaften der eingebetteten Sprache in der Wirtssprache nutzbar, aber auch umgekehrt sind die

Möglichkeiten der Wirtssprache in der eingebetteten Sprache einsetzbar. Hiermit wird eine *Sprachverschmelzung* erreicht.

Eine Verschmelzung zweier syntaktisch und semantisch stark unterschiedlicher Sprachen bedingt Anpassungen, wobei diese in der zu integrierenden Sprache durchzuführen sind, da weiterhin reine Wirtssprachenprogramme mit dem Compiler übersetzt werden sollen. So hat beispielsweise das Schlüsselwort *while* in C^{++} und $ExTeLL^T$ nicht ganz dieselbe Bedeutung. Weiterhin soll jeder zur Verfügung stehende Datentyp in beiden Sprachkomponenten verwendbar sein. Eine solch homogene Integration erlaubt es, daß Entwickler, die mit der Wirtssprache vertraut sind, recht leicht den Umgang mit der Integration erlernen können. Diese nahezu optimale Form der Integration war unser Ziel bei der Integration von ExTeLL in C^{++}, wobei die Verschmelzung der beiden Sprachen mit $ExTeLL^{++}$ bezeichnet wird.

6 Aufbau der syntaktischen Sprachintegration

6.1 Temporale Variablen

C^{++} verwendet neben seinen Grundtypen auch Aufzählungstypen sowie Strukturen bzw. Klassen als benutzerdefinierte Datentypen. Für eine möglichst homogene Integration von $ExTeLL^T$ sind alle in C^{++} verwendbaren Datentypen auch im integrierten $ExTeLL^T$ einsetzbar. Dabei muß man zwischen zwei Arten von Variablen eines Datentyps unterscheiden. Zum einen bleiben weiterhin die klassischen imperativen Variablen erhalten, die in Form von Platzhaltern zur Speicherung von Werten eingesetzt werden. Zum anderen werden Variablen logischer und temporaler Natur benötigt, die eine Folge von nicht veränderbaren Belegungen über ein Zeitintervall und damit ein Modell einer Formel beschreiben.

6.1.1 Deklaration temporaler Variablen

In C^{++} werden die Eigenschaften einer Variablen in der Deklaration festgelegt. Mittels sogenannter *Storage-Class-Specifier* kann dabei angegeben werden, *wo* bzw. *wie* die Werte der Variablen im Speicher bzw. Programmodul abgelegt sind. Zusätzlich können über *Type-Specifier* die Eigenschaften der Variablen bestimmt werden. Für die Integration von $ExTeLL^T$ wurden drei neue *Type-Specifier* zur Beschreibung des Variablenverhaltens eingeführt. So kann mittels *'temporal'* eine temporale Variable deklariert werden, wobei **jeder** Datentyp zur Spezifikation dieser Variablen verwendet werden kann. Damit ist es möglich, auch temporale Variablen von Klassen und Strukturen zu deklarieren, wobei das Schlüsselwort *'temporal'* auch mit anderen Specifiern gemischt angewendet werden kann. Weiterhin deklariert der Specifier *'list'* Listen und *'templist'* temporale Listen eines Datentyps.

Die Einbindung temporaler Variablen über Type-Specifier erlaubt auch, temporale Variablen von temporalen Variablen zu bilden, wobei dies aus pragmatischen Ge-

sichtspunkten keinen Gewinn darstellt. Man erkennt jedoch die Universalität dieser Deklarationsart. Des weiteren können temporale Variablen innerhalb nicht–temporaler Strukturen und als Rückgabewert von Funktionen verwendet werden. Syntaktisch ist die Deklaration analog zu den C^{++} eigenen Konzepten aufgebaut, so daß die Einführung der temporalen Variablen für C^{++}–Entwickler keine Umstellung erfordert.

Alle temporale Variablen, die in $ExTeLL^T$–Formeln eingesetzt werden, müssen wie in C^{++} vor ihrer Verwendung deklariert werden. Damit sind wie in $EITeL^T$ alle Variablen typisiert. Dadurch entfallen bei dieser Integrationsform natürlich die Typisierungsprädikate von $ExTeLL^T$ innerhalb von Formeln.

6.1.2 Verwendung temporaler Variablen in C^{++}

Temporale Variablen repräsentieren immer eine Folge von Variablenwerten. Damit diese für Auswertungen genutzt werden können, muß von der Wirtssprache C^{++} eine Zugriffsmöglichkeit auf die Werte bestehen. Dies wird mittels des Arrayszugriffoperators '[]' realisiert, wobei dieser eine *Integer*–Zahl größer oder gleich 0 als Argument erwartet. Dieses Argument repräsentiert den *Zeitpunkt* des gewünschten Wertes. Diese Indizierung läßt sich beliebig schachteln, beispielsweise um temporale Matrizen darzustellen. Wird auf eine temporale Variable ohne Zeitindex zugegriffen, so wird für die Wertbestimmung der *aktuelle* Zeitpunkt verwendet, wobei dieser außerhalb einer $ExTeLL^T$–Programmbearbeitung immer der Anfangszustand 0 ist. Ist das Ergebnis eines solchen Zugriffs eine Struktur oder Klasse, so kann auf die Elemente der Struktur und Methoden der Klasse in gewohnter Weise zugegriffen werden.

Nun hat eine temporale Variable nicht zu jedem Zeitpunkt einen Wert. In Verbindung mit den Standardvariablen in C^{++} hat ein Zugriff auf eine unbelegte Variable meist einen undefinierten, *zufälligen*[7] Wert. Ein analoges Verhalten ließe sich auch für temporale Variablen einrichten, dies wäre jedoch nicht mit der Logik $EITeL^T$ vereinbar, denn auch ein *zufälliger* Wert ist ein Wert. Deshalb wird bei einem Zugriff auf eine zu einem Zeitpunkt unbelegten Variable das Programm in einen Ausnahmezustand versetzt, es wird eine *Exception*[8] mit der Bezeichnung *ExNoValue* ausgelöst. Das zugrundeliegende C^{++}–Basisprogramm hat nun die Möglichkeit, auf diese Exception mittels entsprechender C^{++}–Konstrukte zu reagieren. Wird dies jedoch nicht implementiert, so terminiert das komplette Programm aufgrund des illegalen Zugriffs.[9]

Zur Vermeidung des Aufwands der Exceptionbehandlung besitzen alle temporalen

[7] Der Wert ist in diesen Fällen meist abhängig von der Stackbelegung, die jedoch nicht vorhersehbar ist, so daß man von einem zufälligen Wert spricht. In Verbindung mit Klassen ist es jedoch möglich, über Konstruktoren eine Defaultbelegung für eine Instanz zu erzeugen, dies sind dann jedoch spezielle Konstrukte, die sich nicht auf den allgemeinen Fall übertragen lassen.

[8] Dieser Bestandteil der Sprache C^{++} beruht auf der Idee, daß Programmteile, die aufgrund einer *unnormalen* Situation keine sinnvolle Ergebnisse liefern können, eine *Ausnahme*–Situation erzeugen, in der Hoffnung, daß der (direkte oder indirekte) aufrufende Kontext das Problem behandeln kann. Programmteile, die diese Ausnahmen behandeln wollen, müssen diese *fangen* (vgl. [Stro92]).

[9] Dieses Verhalten ist vergleichbar mit der Dereferenzierung von Pointern in C^{++}, die auf keinen gültigen Speicherbereich verweisen. Auch dies führt meist zur Programmterminierung, wobei dies in diesem Fall durch das Betriebssystem ausgelöst wird.

Variablen eine Methode '*bool HasValue (unsigned int ZP)*', die es dem Programmierer erlaubt, vor einem temporalen Zugriff die Zulässsigkeit des Zugriffs zu erfragen. Diese erwartet als Parameter den Zeitpunkt als *Integer*–Wert und ergibt einen *Boole'schen* Wert, der bei einer belegten *true* und bei unbelegten Variablen *false* ist.

Temporale Variablen können nicht beliebig belegt werden, da sie zu einem Zeitpunkt immer nur einen Wert annehmen können. Aus der Sicht von ElTeL^T und ExTeLL^T ist dabei die Belegung einer Variablen mittels des Zuweisungsoperators '=' eine Formel, die bei einer schon existierenden Belegung mit anderem Wert *false* als Ergebnis hat. Eine solche Zuweisung ist folglich eine erlaubte, jedoch nicht erfüllbare Anweisung und darf deshalb nicht mit einer Exception behandelt werden. Die Belegung temporaler Variablen wird deshalb auf die ExTeLL^T–Seite der Sprachintegration fixiert.

6.2 Integration der ExTeLL^T–Sprachelemente

Die zu integrierenden Komponenten von ExTeLL^T sind Formeln, Ausdrücke, Prozeduren und Funktionen. Da die Ausführung eines ExTeLL^T–Programms oder einer –Prozedur neben der berechneten Variablenbelegung auch ein Ergebnis (i.a. *true* oder *false*) liefert, handelt es sich bei ExTeLL^T–Formeln in Verbindung mit C^{++} auch um Ausdrücke. Zur Unterscheidung von Standardausdrücken, also Ausdrücken, wie sie in reinem C^{++} oder ExTeLL^T verwendet werden, und ExTeLL^T–Formeln und –Prozeduren werden letztere als *temporale Berechnungsausdrücke* bezeichnet.

6.2.1 Standardausdrücke und Funktionen

Die Möglichkeiten zur Durchführung von Berechnungen sind in ExTeLL^T aufgrund der geringen Anzahl von Standardfunktionen stark beschränkt. Weiterhin gibt es keine Möglichkeit, auf Struktur–, Klassen– und Arrayelemente zuzugreifen. Da C^{++} diesbezüglich eine Vielfalt von Möglichkeiten zur Verfügung stellt, wurde die Syntax von ExTeLL^T–Formeln dahingehend erweitert, daß beliebige C^{++}–Ausdrücke als Operator- und Prädikatsargumente zugelassen sind. Dies erlaubt zum einen den Zugriff auf zusammengesetzte Datentypen und ermöglicht zum anderen einen beliebig erweiterbaren Funktionsumfang. So sind auch externe Bibliotheken mit komplexen Berechnungsalgorithmen innerhalb von ExTeLL^T–Formeln einsetzbar. Weiterhin können innerhalb von Funktionen Zwischenergebnisse in nicht temporalen Variablen gespeichert werden.

Nun existieren in ExTeLL^T auch zusätzliche temporale Operatoren innerhalb von Ausdrücken. Diese werden natürlich weiterhin unterstützt, wobei das Problem auftritt, daß diese Operatoren wie *pst* auf Formeln, die ja innerhalb C^{++} ebenfalls Ausdrücke sind, und auf nicht temporalen Berechnungsausdrücken eine andere Semantik besitzen.

Deutlicher wird dieses Problem bei parameterlosen Prädikaten wie '*more*' und '*empty*'. So kann der Ausdruck '*more*' als Boole'scher Ausdruck interpretiert werden, der die Existenz eines Folgezustandes testet, oder als temporaler Berechnungsausdruck, der einen Folgezustand bedingt. Nun könnte man die Interpretation abhängig vom gewünschten Ergebnistyp definieren. Dies steht jedoch im Widerspruch zu der Wirtssprache C^{++}, die

Operatoren mit gleichen Parametern, aber unterschiedlichem Ergebnistyp nicht zuläßt. Weiterhin ist es in C^{++} möglich, Ausdrücke zu verwenden, ohne den Rückgabewert zu betrachten, wodurch die Interpretation in diesen Fällen undefiniert ist.

Aus diesen Gründen erhalten die temporalen Operatoren '*empty*', '*more*', '*start*', '*pal*' und '*pst*', die zum Testen von Intervall- und Belgungsbedingungen eingesetzt werden, auf Ausdrucksebene außerhalb von $ExTeLL^T$–Formeln eigene Bezeichner, so daß die Semantik eindeutig zugeordnet werden kann. Dabei wird dem jeweiligen Schlüsselwort ein '*check*' vorangestellt. Der Ausdruck '*checkempty*' überprüft folglich, ob eine aktuell ausgewertete Formel sich im letzten Auswertungszeitpunkt befindet, während der Ausdruck '*empty*' eine Formel, also einen temporalen Berechnungsausdruck repräsentiert, die ein Intervall der Länge 1 als Ergebnis des Ausdrucks berechnet.

6.2.2 Formeln

Wie schon zuvor beschrieben, werden Formeln als temporale Berechnungsausdrücke in C^{++} aufgenommen, die wie Standardausdrücke zu Berechnungen herangezogen werden können. Der Aufbau einer Formel entspricht bis auf wenige Ausnahmen der $ExTeLL^T$–Syntax, wobei für Ausdrücke innerhalb einer Formel nun beliebige C^{++}–Ausdrücke entsprechenden Typs verwendet werden können. Führt die Programmausführung zu einer Auswertung eines temporalen Berechnungsausdrucks, wird dieser mittels der Abarbeitungsmethoden von $ExTeLL^T$ berechnet. Für das Ergebnis dieser Ausdrücke wurde der neue Datentyp *tempenv* (*Temporal Environment*) in die Wirtssprache aufgenommen, der bis auf die Variablenbelegungen sämtliche Informationen der Berechnung enthält, beispielsweise die berechnete Intervallänge und Berechnungsinformationen für Alternativen. Da eine $ExTeLL^T$–Formel aufgrund nicht auswertbarer Ausdrücke nicht immer vollständig abgearbeitet werden kann, wird auch eine mögliche Restformel in dieser Struktur abgelegt. Variablen diesen Typs kann nun eine Belegung in Form eines $ExTeLL^T$–Programms zugewiesen werden:

temporal double pi;
*tempenv first_evaluation = always (pi = 2 * acos(0)) and length (3);*

6.2.2.1 Zuweisungen an temporale Variablen

Zuweisungen an temporale Variablen sind immer Formeln und liefern als Rückgabewert damit immer eine Instanz vom Typ *tempenv*. Wie oben beschrieben wird bei einem Zugriff auf eine temporale Variable ohne Angabe eines Zugriffszeitpunkts der aktuelle Zeitpunkt herangezogen, so daß die in $ExTeLL^T$ verwendeten Zuweisungsoperatoren in gewohnter Weise eingesetzt werden können. Hat nun eine temporale Variable ein zusammengesetzter Datentyp, so können dabei die Strukturelemente nicht einzeln belegt werden. Andernfalls tritt das Problem auf, feststellen zu müssen, wann eine Variable zu einem Zeitpunkt vollständig belegt und damit auswertbar ist. Aus diesem Grund können Variablen von zusammengesetzten Datentypen nur als ganzes belegt werden. Dies erfordert jedoch bei Strukturen, die intern Pointer auf Strukturelemente verwenden, daß das Objekt als ganzes zuweisbar ist. Dies bedingt in der Regel eine

Überladung des Zuweisungsoperators. Ebenso ist für einen Vergleich auf gleiche Belegung der Vergleichsoperator '==' anzupassen. Dies ist jedoch in Verbindung mit solchen Strukturen in C^{++} ohnehin anzuraten.

6.2.2.2 Operatoren

Formeln können innerhalb eines C^{++}-Programms an allen Positionen verwendet werden, an denen Ausdrücke zugelassen sind. Anhand der Operatoren, also der Schlüsselwörter von $ExTeLL^T$, kann der $ExTeLL^{++}$-Compiler erkennen, ob es sich hierbei um eine Formel handelt und damit ein anderer Verarbeitungsmechanismus durchzuführen ist. Diese anhand des syntaktischen Aufbaus eines Konstrukts verzweigende Quelltextbearbeitung bedingt jedoch, daß alle $ExTeLL^T$-Operatoren genauso wie C^{++}-Schlüsselwörtern nicht als Variablen- und Funktionsnamen verwendet werden können (vgl. [Stro92]).

Des weiteren existieren in C^{++} und $ExTeLL^T$ Schlüsselwörter, die in beiden Sprachen mit unterschiedlicher Semantik verwendet werden. Für den Zuweisungsoperator '=' stehen dabei in C^{++} Mechanismen zur Verfügung, diesen für eigene Bedürfnisse zu überladen, so daß sich die besondere Behandlung der Zuweisung an temporale Variablen homogen in C^{++} integrieren läßt. Andere Operatoren müssen jedoch bezüglich der Schlüsselwörter bzw. ihrer Syntax an die Wirtssprache angepaßt werden. So werden bedingte Formeln in $ExTeLL^T$ mittels des *if*-Operators repräsentiert, der in der Integration als Ergebnis eine Instanz von *tempenv* ergeben muß. In C^{++} leitet *if* jedoch ein Statement ein und hat folglich kein Ergebnis. Bedingte Ausdrücke werden in C^{++} mittels des '*? :* '-Operators[10] dargestellt. Daher bietet es sich an, diese Form der Darstellung bedingter Formelauswertungen für $ExTeLL^{++}$ zu übernehmen. Aber auch dann ist keine eindeutige Zuordnung bzgl. C^{++} bzw. $ExTeLL^T$ möglich wie das folgende Beispiel zeigt:

```
int i;
temporal int tempint;
...
(i >= 0) ? (tempint = 5) : (tempint = 6);
```

Dieses Beispiel zeigt, daß es nicht von Bedeutung ist, ob der bedingte Ausdruck als C^{++}-Ausdruck oder als $ExTeLL^T$-Formel anzusehen ist. Beide Auswertungsarten haben die gleiche Interpretation und liefern bei positivem *i* eine Instanz von *tempenv*, die durch die Zuweisung von 5 an *tempint* entsteht, bzw. bei negativem *i* das entsprechende Ergebnis mit der Zuweisung von 6. Dies zeigt auch, daß die '*? :*'-Schreibweise für bedingte Formeln eine homogene und natürliche Integration darstellt.

Auch die Schleifenkonstrukte '*while*' und '*for*' sind in C^{++} Statements, also Anweisungen ohne Ergebniswert. Im Gegensatz zu '*if*' gibt es jedoch dazu kein analoges Ausdrucks-Konstrukt in C^{++}. Des weiteren unterscheiden sich die Semantiken der

[10] Der '*? :* '-Operator hat in C^{++} den Aufbau '<expression> ? <expression> : <expression>'. Ist der erste Ausdruck *true* (*nicht* 0) so wird das Ergebnis des zweiten ansonsten das Ergebnis des dritten Ausdrucks als Gesamtausdrucksergebnis ermittelt.

Schleifen der beiden Sprachen zu stark, so daß diese Schlüsselwörter zur Erkennung von ExTeLL^T-Formeln durch ein vorangestelltes *t* (temporal) modifiziert wurden. Zusätzlich wurde die Syntax der Schleifenkonstrukte von ExTeLL^T an die Wirtssprache angepaßt, um eine homogenere Sprache zu erreichen. Dadurch wurden die beiden (sehr ähnlichen) '*for*'-Schleifen von ExTeLL^T zu einem allgemeineren Konstrukt, wie es die Syntax von C^{++} erlaubt, zusammengefaßt.

6.2.3 Verknüpfung von Formeln

Teilformeln können in der Integration weiterhin mittels '*and*', '*or*' und '*chop*' verknüpft werden,[11] die Einführung des Datentyps *tempenv* erlaubt weitere, flexiblere Verknüpfungsmöglichkeiten. So ist es in der Integration möglich, das Ergebnis einer Formelauswertung mit zusätzlichen Formelelementen zu ergänzen, wodurch sich Formeln je nach Umgebung oder abhängig von Eingaben aufbauen lassen. Man betrachte hierzu folgendes Beispiel:

```
bool add;
temporal int tempint;

tempenv env = (tempint = 0) and ...;
if (add == true)
        env = env and (tempint gets tempint + 1);
else
        env = env and (tempint gets tempint - 1);

tempenv env2 = env chop (length(3) and ...);
tempenv env3 = env chop (halt (tempint < 0) and ...);
```

Bei diesem Beispiel wird zuerst die Formel mit der Initialisierung der Variablen *tempint* ausgewertet. Die Variable *env* enthält hiernach die Auswertungsinformationen, die u.a. die Restformel beinhaltet, die bei vollständiger Abarbeitung *true* oder *false* repräsentiert. Diese Restformel wird dann je nach Belegung der Variablen *add* mit der Formel zur Berechnung der Zukunftswerte für *tempint* ergänzt, so daß insgesamt die Formel ausgewertet wird. Das Ergebnis wird in diesem Beispiel wieder in der ursprünglichen Variablen abgespeichert. Dies kann jedoch auch einer anderen Variablen zugewiesen werden, so daß eine Grundformel duch mehreren Formeln ergänzt werden kann und dadurch Mehrfachdefinitionen vermieden werden können. Dies bietet insbesondere mit *chop* die Möglichkeit, mehrere Intervallfortsetzungen zu implementieren.

Bei diesen Verknüpfungen ist jedoch zu beachten, daß ohnehin alle Formeln bezüglich der Variablenbelegungen *und*-verknüpft sind. Der Vorteil einer Formelerweiterung mit *and* liegt folglich primär in den Intervallängenprädikaten, die sich auf die resultierende Gesamtformel beziehen.

[11] Auf eine syntaktische Anpassung an C^{++}, in der für Boole'sche Verknüpfungen die Operatoren '&&' (and) und '||' (or) zur Verfügung stehen, wurde verzichtet, da bei diesen Operatoren im Gegensatz zu der Verknüpfung *and* innerhalb von Formeln die genaue Auswertungsreihenfolge festgelegt ist (vgl. [Stro92]).

Bei Formelergänzungen mittels *or* entsteht das Problem der Zuordnung der hinzukommenden Formel. Diese könnte sich zum einen auf die Restformel und zum anderen auf die ursprüngliche Formel beziehen. Dabei ist es bei einer Zuordnung zur Restformel nur bei sehr guter Kenntnis der Reduktionsmechanismen der Formelbearbeitung abschätzbar, in wie weit ein evtl. notwendiges Backtracking Variablenbelegungen zurücksetzten kann. Ein Bezug auf die ursprüngliche Formel bedingt, daß für jede Formelauswertung präventiv für eine evtl. folgende Formelergänzung via *or* Backtrackinginformationen gesammelt werden müssen, was Performancerückgänge bedingt. Zur Vermeidung dieser Probleme wird eine solche Art der Verknüpfung für den Operator *or* nicht zugelassen.

6.2.4 Prozeduren

Die Einführung des Datentyps *tempenv* und die oben beschriebenen Verknüpfungsmöglichkeiten erlauben es, Formeln in eigene Funktionen mit dem Ergebnistyp *tempenv* auszulagern. In der Integration wurde diese Möglichkeit derart verallgemeinert, daß solche Funktionen an beliebiger Stelle innerhalb einer Formel (z.B. als Argument von *always*) verwendet werden können. Dadurch wird das Prozedurkonzept von ExTeLL^T, das intern Prozeduren als Makros behandelt, die beim Aufruf in die aufrufende Formel eingesetzt werden, überflüssig. So ergibt ein Prozedurkonzept auf der Basis von C^{++}–Funktionen zum einen eine homogenere Integration. Zum anderen wird die Möglichkeit geboten, Prädikate, die in Formeln verwendet werden können, in C^{++} zu implementieren.

Diese Art der Integration gestattet folglich die Nutzung von ExTeLL^T innerhalb eines C^{++}–Programmes **und** zugleich die Nutzung von C^{++} innerhalb einer ExTeLL^T–Formel. Dies erlaubt Programme zu entwickeln, deren zentrale Verarbeitungsfunktion eine ExTeLL^T–Formel ist und C^{++} für Hilfsfunktionen nutzt, sowie Programme mit vertauschter Konstellation. Darüber hinaus können aufgrund der Struktur von C^{++} Prozeduren überladen und eine objektorientierte Programmkonzeption für die Entwicklung eingesetzt werden. Man kann hier von einer echten **Sprachverschmelzung** sprechen.

7 Aufbau des ExTeLL^{++}–Compilers

Der ExTeLL^{++}–Compiler wird in Form eines Präprozessors realisiert. Hierbei wird aus einem ExTeLL^{++}–Programm ein C^{++}–Quelltext erzeugt, der in Verbindung mit einer einzubindenden Ausführungskontrolle die Durchführung der Formelauswertung ermöglicht. Das System besteht aus folgenden Komponenten:

Der Parser

Hauptaufgabe des Parsers ist die Erkennung der ExTeLL–Komponenten innerhalb des ExTeLL^{++}–Programms und deren anschließende Überführung in eine interne Struktur. Da eine Bearbeitung des vom Code–Generator

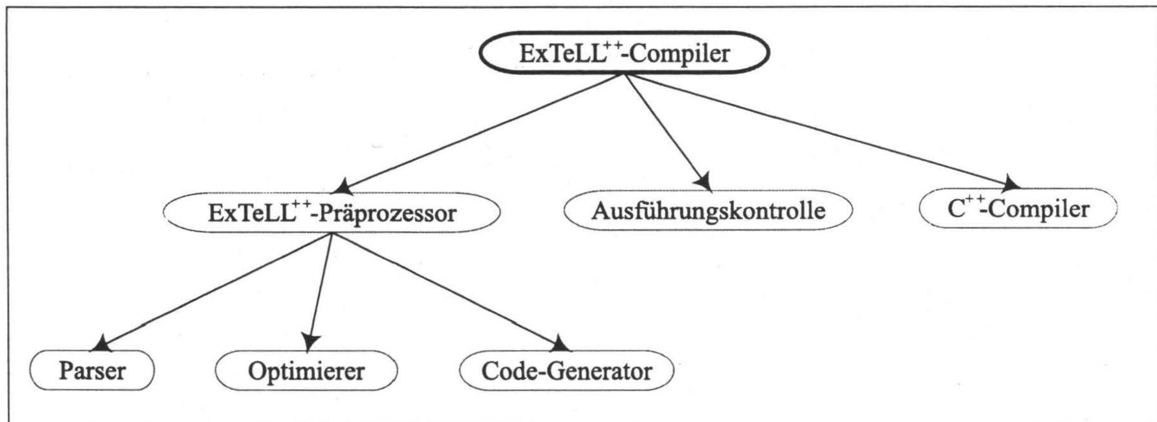

Abbildung 3: Komponenten des ExTeLL^{++}–Compilers

erzeugten C^{++}–Quelltextes nur mit sehr guten Kenntnissen des Präprozessors möglich ist, müssen dabei syntaktische Fehler auch im C^{++}–Quelltext erkannt werden.

Der Optimierer

Um die Formelauswertung zu beschleunigen, werden innerhalb des Optimierers die nach dem Parsen in einer internen Struktur vorliegenden Formeln vor der Code–Erzeugung so weit wie möglich ausgewertet. Weitergehende Optimierungsmöglichkeiten sind noch zu untersuchen.

Der Code–Generator

Der Code–Generator erzeugt anhand der internen Formelrepräsentation einen C^{++}–Quelltext, der eine Formelauswertung ohne erneutes Parsen der Formel erlaubt. Dabei werden interne Strukturen, Verweise und Ausführungsanweisungen aufgebaut, die direkt von der Ausführungskontrolle verarbeitet werden können. Dadurch wird die sequentielle Programmausführung der Wirtssprache beibehalten, ExTeLL–Formeln werden immer zwischen den einschließenden C^{++}–Konstrukten ausgeführt.

Die Ausführungskontrolle

Die Ausführungskontrolle enthält die Mechanismen zur Verarbeitung der Formelrepräsentation in Form einer Funktionsbibliothek, die zu dem ausführbaren Programm hinzugebunden werden muß.

Der C^{++}–Compiler

Der C^{++}–Compiler übernimmt weiterhin die eigentliche Erzeugung eines ausführbaren Programms, wobei bis auf ANSI–Konformität keine Anforderungen an diesen bestehen. Dies erlaubt auch eine plattformunabhängige Programmentwicklung mit ExTeLL^{++}.

8 Ausblick

Die Ausführung einer ExTeLL^{++}–Formel wird durch fortlaufende Reduktion realisiert. Dabei endet die Reduktion, wenn als Restformel *true* oder *false* verbleibt. Im positiven Fall beschreiben dann die vorliegenden Variablenbelegungen ein Standard–Modell der Ausgangsformel. Damit die Korrektheit der Transformation und die Modelleigenschaft sichergestellt werden können, müssen die Zusammenhänge zwischen ExTeLL^{++} und EITeL formal definiert werden. Dabei sind Insbesondere die Möglichkeiten mehrere Formeln innerhalb eines Programms auszuwerten, C^{++}–Funktionen innerhalb von Formeln zu nutzen und Variablen vorzubelegen, sowie die funktionalen Typzusammenhänge zu berücksichtigen.

Insbesondere werden Untersuchungen zum Einsatz der vorgestellten Integration an konkreten Aufgabenstellungen durchgeführt. Dabei ist u.a. die Verwendung in Steuerungsprogrammen geplant. Weiterhin sind Analysen zum Einsatz von ExTeLL^{++} in Verbindung mit temporalen Datenbanken vorgesehen. Wir erwarten dabei im Vergleich zu imperativen Lösungsansätzen eine Vereinfachung des Programmaufbaus und damit eine effizientere Programmentwicklung sowie besser verifizierbare Programme.

Literatur

[Aven95] Avenhaus, Jürgen: *Reduktionssysteme.* Springer Verlag, 1995.

[GoMe89] Goguen, Joseph A., Meseguer, José,: *Order–Sorted Algebra I: Equational Deduction for Multiple Inheritance, Overloading, Exceptions and Partial Operations.* Technical Monograph PRG–80, Oxford University Computing Laboratory, 1989.

[Här87] Härder, T.: *Realisierung von operationalen Schnittstellen.* Datenbank–Handbuch, pp. 163–335, Springer Verlag, 1987

[HuOp80] Huet, G.P., Oppen, D.C.: *Equations and rewrite rules.* In Formal Languages, pp. 349–393, Academic Press, New York, 1980.

[Kan92] Kandzia, P.-T.: *Ein Interpreter für ExTELL – eine erweiterte temporallogische Programmiersprache.* Diplomarbeit im Fachbereich Informatik, Universität Kaiserslautern, 1992.

[Krö86] Kröger, F.: *Temporal Logic of Programs* Springer Verlag, 1986.

[MaMa94] Malik, Robi; Mayer, Otto: *Eine Fixpunkt–Semantik für temporal stratifizierte Programme.* Interner Bericht am Fachbereich Informatik, Universität Kaiserslautern, 1994.

[Kir94] Kirchberg, Paul: *ExTELL – Eine erweiterte temporallogische Programmiersprache.* Diplomarbeit im Fachbereich Informatik, Universität Kaiserslautern, 1994.

[Schwi93] Schwind, Stefan: *Pragmatik von ExTELL.* Projektarbeit im Fachbereich Informatik, Universität Kaiserslautern, 1993.

[Spi95] Spies, K.G.: *Transformationsbasierte temporallogische Programmierung.* Dissertation, Universität Kaiserslautern, 1995.

[SpMa92] Spies, K.G., Mayer, O.: *EITeL – ein intervallbasierter temporallogischer Ansatz.* Interner Bericht, FB Informatik, Universität Kaiserslautern, 1992.

[Stro92] Stroustrup, Bjarne: *Die C++ – Programmiersprache.* Addison–Wesley, 2. Auflage, 1992.

[Watt96] Watt, David A.: *Programmiersprachen: Konzepte und Paradigmen.* Carl Hanser Verlag, 1996.