Gefördert vom
bmb+f
Bundesministerium für
Bildung, Wissenschaft,
Forschung und Technologie

Verbmobil
Verbundvorhaben

Getypte Merkmalstrukturen und Unifikation zur Verarbeitung temporaler Ausdrücke in Verbmobil

Stefan Haas

Technische Universität Berlin

Vm
Report 234
Februar 1999

Februar 1999

Stefan Haas

Kontaktadresse:
Technische Universität Berlin
Fachbereich Informatik (13)
Projektgruppe KIT
Sekr. FR 6-10
Franklinstraße 28/29
D-10587 Berlin

Tel.: (030) 314 - 24944
Fax: (030) 314 - 24929
e-mail: `kitvm11@cs.tu-berlin.de`

Gehört zum Antragsabschnitt: AP4.2 Domänenmodellierung
AP4.5 Desambiguierung u. Klärungsdialoge

Die vorliegende Arbeit wurde im Rahmen des Verbundvorhabens Verbmobil vom Bundesministerium für Bildung, Wissenschaft, Forschung und Technologie (BMBF) unter dem Förderkennzeichen 01 IV 701 P 9 gefördert. Die Verantwortung für den Inhalt dieser Arbeit liegt bei dem Autor.

ISSN 0949-6084

Kurzfassung

Zur Modellierung temporaler Äußerungen in Terminabsprachedialogen werden getypte Merkmalstrukturen verwendet, welche in einem objektorientierten Rahmenwerk implementiert sind. Ein Typsystem spezifiziert Objekte, die zeitliche Informationen uniform als Intervallbeschreibungen (TIDs[1]) kodieren. Zur Verarbeitung der TIDs wird der Unifikator des Rahmenwerks als Theorieunifikator an die Anforderungen der Domäne angepaßt. Die erweiterte Unifikationsoperation berechnet Teilunifikatoren und interpretiert Inklusion als Spezialisierung von TIDs. Eine in Java realisierte Komponente integriert das Domänenmodell als Inferenzserver in das System und implementiert Operationen zur Konstruktion von TIDs aus Ausdrücken kontextfreier Zeitbeschreibungssprachen.

Inhaltsverzeichnis

[1] *Time Interval Descriptions*

1 Verarbeitung temporaler Ausdrücke

In den Domänen "Terminverhandlung" sowie "Reiseplanung" und "Hotelreservierung" des VERBMOBIL-Projektes[2] spielt die Verarbeitung zeitlicher Ausdrücke eine wesentliche Rolle. Um zeitliche Ausdrücke repräsentieren zu können, wurde in dem im Teilprojekt TP11 angesiedelten Projekt KIT-VM11 an der Technischen Universität Berlin die kontextfreie Grammatik ZeitGram [Küssner, Stede, 1995] (ZG) entwickelt. Die Sprache der ZG ($\mathcal{L}(ZG)$) dient als Zwischenrepräsentation von temporalen Äußerungen für die Übersetzung. Auf der in einem Ausdruck der $\mathcal{L}(ZG)$ repräsentierten temporalen Information basieren verschiedene kontextbasierte Inferenzoperationen.

Um zum Beispiel den Dialogakt einer Äußerung wie:

> "Freitag könnte mir passen"

eindeutig bestimmen zu können, muß das Übersetzungssystem erkennen, ob es sich bei dem durch "Freitag" ausgedrückten propositionalen Gehalt um einen neuen Termin handelt, d.h. ein Vorschlag gemacht wird, oder ob ein bereits verhandelter Termin akzeptiert wird.

Die Analyse spontan gesprochener Sprache führt häufig zu syntaktisch und semantisch deformierten Äußerungen. Um dennoch korrekte Übersetzungen zu leisten, muß das System in der Lage sein, Kontextinformation zur Evaluation und eventuellen Reparatur der Eingabe nutzen zu können. Gerade bei Terminabsprachen treten Fälle auf, die eine tiefergehende Interpretation wünschenswert machen. So sollte ein intelligentes Übersetzungssystem einen inkonsistenten Termin wie z.B: "Freitag der 30. Februar" oder "heute früh um 17:00 Uhr" erkennen und einen Klärungsdialog mit einem Korrekturvorschlag initiieren.

Daraus ergeben sich für eine temporale Inferenzkomponente im Rahmen der maschinellen Übersetzung folgende Anforderungen:

1. Unterschiedliche kontextuelle Wissensquellen müssen zur Auswertung temporaler Ausdrücke herangezogen werden.

2. Verschiedene zeitliche Relationen, wie z.B. "vorher" oder "gleich" zwischen den durch zeitliche Ausdrücke referierten Objekten müssen berechenbar sein.

3. Die Repräsentation temporaler Information muß die Darstellung partieller Information ermöglichen und einen Mechanismus bereitstellen, diese sukzessiv und konsistent mit neuem Wissen anzureichern.

[2] http://www.dfki.uni-sb.de/verbmobil

Die wesentliche Informationsquelle für die Konstruktion einer Repräsentation des temporalen Gehaltes eines Dialoges sind die bei der Analyse erzeugten durch die ZG beschriebenen Zeitausdrücke. Die ZG soll im folgenden kurz vorgestellt werden.

1.1 Kurzübersicht *ZeitGram*

Als Ausgangspunkt der inhaltlichen Verarbeitung zeitlicher Information dient die im Rahmen des VERBMOBIL-Teilprojektes KIT-VM11 entworfene ZG. Eine ausführliche Beschreibung der ZG findet sich im VERBMOBIL-Memo von Uwe Küssner und Manfred Stede ([Küssner, Stede, 1995])[3]. An dieser Stelle soll nur auf die für eine inhaltliche Auswertung relevanten Aspekte eingegangen werden.

Die ZG beschreibt eine umfangreiche Menge zeitlicher Ausdrücke, die für ontologisch sehr unterschiedliche Objekte stehen. Unter der Hauptkategorie `DATE_EXPR` werden quantifizierte und fuzzifizierte Ausdrücke sowie Auswahllisten und `TEMPOBJ`s zusammengefaßt. Quantifizierte Ausdrücke, wie z.B. "jeder Montag im Mai", analysiert als `all([dow:mon] , [moy:mai])` oder "jeder Montag" (`all([dow:mon],[])`) läßt sich intuitiv als eine Struktur über einem Kalender auffassen, wohingegen Ausdrücke wie `all([],[moy:may])` ("Immer im Mai") oder `any([dow:mon],[moy:may])` (" irgendein Montag im Mai") auf diskrete Objekte referieren, die sich als Zeitspannen konzeptualisieren lassen. Hier soll, ausgehend von den Vergleichsoperationen *vorher*, *gleich*, *enthält* usw., die Repräsentation von Terminen bzw. Datumsobjekten geleistet werden.

Damit beschränkt sich der hier vorgestellte Ansatz im wesentlichen auf Audrücke der ZG-Kategorie `TEMPOBJ`, wobei jedoch auch der `FUZZYFIER about`, Auswahllisten[4] und bedingt quantifizierte Ausdrücke mitberücksichtigt werden. Innerhalb der grammatischen Kategorie der `TEMPOBJ`s stellen Zeitdauern (`DURATION`) eine semantische bzw. ontologische Ausnahme dar. So wird z.B. "für drei Stunden" als `dur(3,hour)` dargestellt. Der Typ des so beschriebenen Objektes "Zeitdauer von drei Stunden" läßt eine andere Klasse von Prädikaten als die über den `TEMPOBJ`s definierbaren zu; so z.B. *gleich(-lang)*, *länger* etc. Prinzipiell kommen also nur solche Audrücke für diesen Vorschlag in Betracht, die sich in irgendeiner Form als kalendarisches Datum - wenn auch unterspezifiziert - darstellen lassen. Insbesondere betrifft das alle `POINT`, `POINTLIKES` sowie Intervalle, Ausdrücke mit Referenzzeitpunkt[5] und Ausdrücke mit Abzählung[6]. Die so informell definierte Teilmenge von $\mathcal{L}(ZG)$ soll kurz L genannt werden.

---

[3]Ein Verzeichnis der Abkürzungen und Kategorien der ZG findet sich im Abkürzungsverzeichnis auf Seite 53.

[4]`ao(LIST_OF_DATE_EXPRS+)`

[5]Z.B. "die letzte Woche vor Ostern" (`before(last, week, holiday:easter)`)

[6]"im nächsten Januar" (`next(moy:jan)`); s.a. Kap 2 [Küssner, Stede, 1995]

Die ZG, als eine sprachnahe Repräsentation für zeitliche Ausdrücke, hat den Vorteil, die Übersetzung VIT[7] $\Longrightarrow \mathcal{L}(ZG)$ durch ihre Ausdrucksstärke zu erleichtern. Dieser Übersetzungsschritt beinhaltet, daß von einzelsprachspezifischen Ausdrucksformen abstrahiert wird, d.h. die durch ZG beschriebene Sprache hat das Abstraktionsniveau einer Interlingua. So werden z.B. die linguistischen Varianten "viertel vor elf", "dreiviertel elf", "zehn Uhr fünfundvierzig", ... in dem kanonischen Format `hh:mm` zu einer Äquivalenzklasse zusammengefaßt. Es bleibt jedoch die Art und Weise, in der auf ein temporales Objekt – *eine Zeit* im weitesten Sinne – referiert wird, erhalten und in dem entstandenen Ausdruck der ZG kodiert.

So bietet die ZG mindestens folgende Formulierungen[8] an, um sich auf die mittels der kalendarischen Angabe "24.12.97 ca. von 18:00 bis 22:00" bezeichnete Zeit zu beziehen:

1. `[christmaseve, year:1997]`
2. `[moy:dec, dom:24, pod:evening, year:1997]`
3. `[woy:52, dow:wed, pod:evening, year:1997]`
4. `[woy:52, dow:wed, boundaries(tod:6:00, tod:10:00, from_to), pod:pm, year:1997]`

Damit ist diese Zwischenrepräsentation im Rahmen der maschinellen Übersetzung gleichermaßen für die Generierung wie für die Analyse geeignet.

Die komplexe Syntax erschwert es jedoch sehr, den durch ZG generierten Ausdrücken eine klare Semantik zuzuordnen. So scheint eine modellbasierte Semantik direkt auf der Basis von ZG in Form einer induktiven Definition einer Interpretation für ZG-Ausdrücke kompliziert und unübersichtlich. Im Rahmen der Erweiterung der ZG zu einer ausdrucksstärkeren Grammatik TEL[9] wurde eine denotationelle Semantik auf der kontextfreien Sprache $\mathcal{L}$(TEL) definiert, die jedoch nicht zu einer brauchbaren Implementierung führte. Problematisch für diesen Ansatz sind unterspezifizierte Ausdrücke, deren Semantik als Mengen von Zeitpunkten repräsentiert wird. Die Kardinalität der Menge, die einem Zeitausdruck als semantische Repräsentation zugeordnet ist, wächst umgekehrt proportional zu dessen Informativität und kann deshalb nicht effizient verarbeitet werden.

Gerade die in der ZG angelegte Möglichkeit, auf verschiedene Art und Weise auf ein temporales Objekt zu referieren, bedeutet, daß dabei überflüssige Arbeit geleistet werden muß. $\mathcal{L}(ZG)$ enthält Teilmengen syntaktisch unterschiedlicher

[7] *VERBMOBIL Interface Term*

[8] Mit etwas Mühe lassen sich weitaus mehr Ausdrücke mit gleichem Gehalt konstruieren, vor allem, wenn man alle trivialen Kombinationen miteinbezieht.

[9] Die Grammatik der *Temporal Expression Language* (TEL, s. [Endriss, 1998]) ist ausdrucksstärker im linguistischen Sinne; denn TEL ist wie ZG kontextfrei.

Ausdrücke, die das gleiche zeitliche Objekt beschreiben und für die alle die gleiche Semantik definiert werden muß.

Es empfielt sich also für eine semantische Interpretation der ZG diese auf eine uniforme Darstellung abzubilden und auf dieser Basis die nötigen Inferenzoperationen durchzuführen.

1.2 Interpretation temporaler Ausdrücke

Der folgende Abschnitt zeigt, daß ein auf Intervallen beruhendes Modell als Basis geeignet ist, die im Rahmen der relevanten Verhandlungssituationen auftretenden Zeitausdrücke darzustellen. Besonders die Verarbeitung partieller Information wird anhand von zwei Dialogbeispielen verdeutlicht und die Modellierung als unterspezifizierte Intervallbeschreibungen motiviert.

1.2.1 Intervalle als Referenzobjekte von Terminabsprachen

Der Verlauf des Beispieldialogs (Abb.1) zeigt, wie ein zunächst weiter terminlicher Rahmen sukzessive eingeengt wird. Wie auch sonst oft bei Terminabsprachen zu beobachten, wird hier zwar ein Zeitpunkt intendiert, aber innerhalb eines Intervalls ausgehandelt. Dieser Vorgang läßt sich natürlicher auf der Basis von Intervallen mit Zeitpunkten als Sonderfällen modellieren als umgekehrt.

A01:	hi how you doing I would like to schedule a meeting some time in June how does the eleventh sound
B01:	okay . der elfte irgendwann nach zwölf Uhr paßt mir .
A02:	okay then I will take a one o'clock appointment thank you
B02:	gut . halten wir ein Uhr fest .

Abbildung 1: Beispieldialog

Die logischen Probleme, die sich einem Repräsentationssystem zeitlicher Information stellen, das ausgehend von der Darstellung von Zeitpunkten als zugrundeliegendes Modell die reellen Zahlen wählt, werden in (Allen [Allen, 1983], [Allen, 1984]) ausführlich diskutiert. Allens Ansatz der Darstellung temporaler Information ist vor allem auf mit Zeiten assoziierte Ereignisse gerichtet. Die Möglichkeit der Dekomposition eines Ereignisses wirft die Frage nach einer prinzipiell unendlichen Teilung der zeitlichen Dauer auf. Der hier angestrebten

Modellierung stellt sich dieses Problem des Zeitkontinuums nicht, da von einem implizit in der *ZG* enthaltenen Kalendersystem ausgegangen wird. Dessen Granularität läßt sich zwar prinzipiell beliebig verfeinern, setzt jedoch in jedem Fall diskrete kleinste Intervalle wie Minuten oder Sekunden als Grundelemente voraus. In diesem Fall ist die kleinste Einheit durch `minute` gegeben und Elemente diesen Typs werden im weiteren auch als Zeitpunkte bezeichnet.

A11: alright, how is your schedule on the sixth, tenth,
or thirteenth.
B11: vom zweiten bis fünfzehnten bin ich im Urlaub
wir müssen es also danach machen .
A12: okay, how about the seventeenth, twentieth, or twenty fourth.
I'm free all day on those days
B12: ¡äh¿ der vierundzwanzigste würde gehen.
um vier Uhr habe ich einen Arzttermin
wir müssen
uns also vorher treffen .
A13: how about an early start. nine to eleven on the morning of the
twenty fourth

Abbildung 2: Beispieldialog

Sowohl voll spezifizierte als auch unterspezifizierte Ausdrücke, die sich auf Zeitpunkte[10] beziehen, lassen sich leicht als Intervall-Beschreibungen mit gleichem Anfangs- und Endpunkt darstellen [Allen, 1984]. Deshalb scheint ein expliziter Unterschied zwischen Zeitpunkten und Intervallen in einem Repräsentationssystem nicht notwendig zu sein. Die in der ZG bestehende Unterscheidung zwischen `POINT` und `POINTLIKE` ist also für die Repräsentation der Bedeutung zeitlicher Ausdrücke nicht relevant.

1.2.2 Die Darstellung partieller Information

Besonders die im VERBMOBIL-Projekt gestellte Aufgabe der Übersetzung spontan gesprochener Sprache beinhaltet, daß große Anteile der zum vollständigen Verstehen einer Äußerung notwendigen Information nur implizit im Kontext gegeben ist. Die beiden Dialogbeispiele in Abb.1 und Abb.2 machen dies deutlich. So fehlt z.B. bei der Äußerung A01 die Jahresangabe, bei B02 ist allein die Uhrzeit gegeben. Dennoch kommt auf der Basis von allgemeiner Kontext-

[10] In diesem Fall Minuten.

information[11] und dem Diskurskontext[12] eine eindeutige Referenz zustande.

Im zweiten Beispiel bleiben sowohl Jahr als auch Monat unausgesprochen. Auch ohne diese Information sind die für das Verständnis notwendigen Relationen interpretierbar. So ist der Schritt von B12 nach A13 nachvollziehbar, da "zwischen neun und elf am Morgen" **vor** "vier"[13] liegt - vorausgesetzt der Referenzrahmen ist gleich.

2 Domänenmodellierung

Um trotz unvollständiger Information Relationen berechnen und Inferenzen ziehen zu können sowie den monotonen Zugewinn an Information in uniformer Weise darstellbar zu machen, werden getypte Merkmalstrukturen als Repräsentationsform gewählt[14]. Die Modellierung basiert auf JTFS[15], einem Rahmenwerk[16] zur objektorientierten Repräsentation getypter Merkmalstrukturen (TFS) in Java.

Ein Typsystem formalisiert die zeitliche Information in Terminverhandlungsdialogen, indem Zeitausdrücke uniform als Intervallbeschreibungen (TIDs[17]) kodiert werden, die JFS[18] erweitern. Zur Spezifikation wird die AVM Darstellung von FS verwendet und UML-Diagramme[19] illustrieren die OOM als JFS.

Der Unifikator von JTFS wird zur Verarbeitung der TIDs zu einem Theorieunifikator TIDU[20] erweitert, der partiell korrekte Teilunifikatoren berechnet und Inklusion als Spezialisierung von Intervallbeschreibungen interpretiert. Das Rahmenwerk JTFS zur Spezifikation von TFS in Java wird um eine Operation zur Modellierung der kalendarischen Nachfolgerelation und um eingeschränkte Disjunktion erweitert.

2.1 JTFS

Die Abb.3 zeigt das Klassenstrukturdiagramm des Rahmenwerks JTFS. JFS werden als objektorientierter Typ realisiert, dessen Design auf dem *Composite* Entwurfsmuster[21] basiert. Teilobjekte der durch das *Composite* Entwurfs-

[11] Jahreszahl
[12] Tag im Monat
[13] pm
[14] S.[Stede, Haas, Küssner, 1998]
[15] *Java Typed Feature Structures*
[16] Engl.: *framework*
[17] *Time Interval Descriptions*
[18] *Java Feature Structures*
[19] *Unified Modeling Language*, [Rational, 1998]
[20] TID *Unificator*
[21] S.[Gamma *et al.*, 1995], S.163ff.

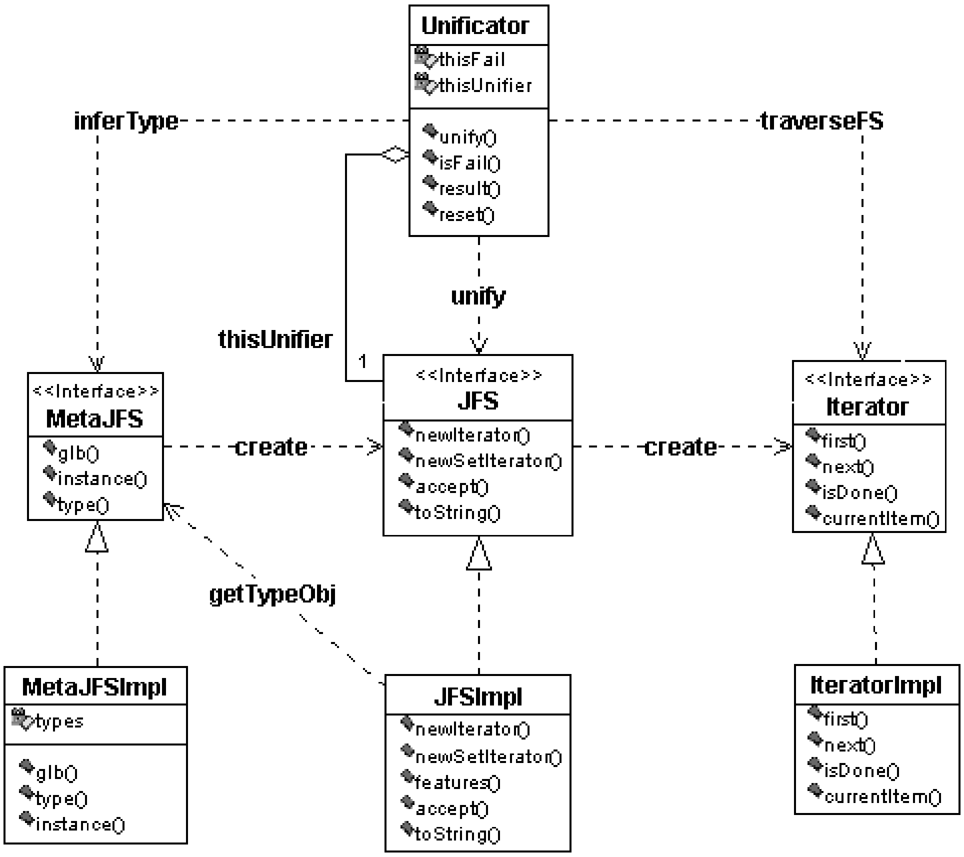

Abbildung 3: Klassenstrukturmodell der JTFS

muster spezifizierten Objekthierarchie erzeugen Instanzen des Typs Iterator[22]. Iteratoren stellen einem Client ein uniformes Interface für den sequentiellen Zugriff auf Elemente eines Aggregats zur Verfügung und werden von Unificator zum Traversieren und Kopieren der *Composite* Objekthierarchien genutzt. Das *Iterator* Entwurfsmuster abstrahiert von konkreten Attributausprägungen, so daß die Methode der Unifikationsoperation unabhängig von spezifischen Objekthierarchien definiert werden kann. Der Algorithmus basiert so auf der rekursiven Definition der Objektstruktur, wie sie durch das *Composite* Entwurfsmuster gegeben ist, und nicht auf der individuellen Definition spezifischer Attribute der einzelnen Objekte der Hierarchie.

[22]S.[Gamma *et al.*, 1995], S.257ff.

Attribute werden in JTFS als ein Feld und Zugriffsmethoden modelliert. Ausgehend dieser Modellierung der Attribute als Zugriffsoperationen auf einzelne Teile einer JFS, ist eine Methode features() als Basis für den generischen Zugriff auf Teile einer JFS in der Klasse JFSImpl definiert. Der Aufruf von features() bei einer Instanz von JFS gibt die Menge der Zugriffsmethoden des Objekts in einem Vektor zurück. Da Java 1.1 durch das Reflection API über ein MOP[23] verfügt, ist die Methode features() sehr leicht zu implementieren.

Iteratoren werden durch die *Factory* Methoden newIterator() und newSetIterator() erzeugt und mit den Methodenobjekten der Rückgabe von features() initialisiert. Die *Iteratoren* unterscheiden sich in der Art von Methodenobjekten, über die sie iterieren und sind durch die Attribute mit dem erzeugenden Objekt assoziiert. Der setIterator enthält die Methoden zum Setzen von Werten einer JFS und dient der Modifikation der Attribute, der einfache *Iterator* enthält die Zugriffsmethoden und wird zum Lesen der Werte der Attribute verwendet. Iterator verwendet Methoden des MOP von Java um den Aufruf der Methodenobjekte bei Teilinstanzen der Objekthierarchie zu realisieren.

Methoden, die Erweiterungen des MOP von Java realisieren, sind in einer eigenen Klasse MetaJFSImpl mit dem Interface MetaJFS zusammengefaßt. MetaJFS stellt eine Erweiterung der Reflection API des JDK 1.1 dar. Das durch `java.lang.Class` realisierte MOP von Java erlaubt Klassen und Interfaces zur Laufzeit als Objekte anzusprechen. MetaJFS verwenden diese Methoden, um die Operationen glb(), instance() und type() zu implementieren.

glb() operiert auf der Hierarchie von Interfaces, die Instanzen von JFS spezifizieren. Die Implementierung von glb() erfordert, daß zur Laufzeit dynamisch von einem Interfaceobjekt auf ein Subinterfaceobjekt zugegriffen werden kann. Javas MOP bietet keine Möglichkeit, den Zugriff direkt zu realisieren, so daß als Erweiterung des MOP von Java die Assoziation von einem Interface zu seinen Subinterfaces realisiert werden muß. Da es sich bei JTFS um einen Prototyp handelt, wird eine implementationstechnisch einfache aber auch ineffiziente Variante gewählt. Interfaces werden mit den geordneten Mengen ihrer Subinterfaces assoziiert, um durch Schnittmengenbildung die Typinferenz zu berechnen. Diese Assoziation wird durch eine Operation berechnet, die die Hierarchie der Interfaces bottom up durchläuft und ein assoziatives Hashtable types anlegt, das jedem Interfaceobjekt eine geordnete Menge von Interfaceobjekten zuordnet. Die Ordnung der Menge repräsentiert die Distanz innerhalb der Hierarchie der Interfaces. Das Hashtable realisiert so eine Abbildung von Interfaceobjekten auf eine nach Nähe sortierte geordnete Menge von Subinterfaceobjekten. In der Methode glb() wird die Tabelle verwendet, um ausgehend von zwei Interfaces das in der Hierarchie am weitesten oben stehende gemeinsame Subinterface zu berechnen, welches das erste gemeinsame Element der geordneten Mengen von Subinterfaces ist.

[23] *Meta Object Protocoll*

Um die Methode glb() aufzurufen, müßen die Typen der zu unifizierenden Objekte bestimmt werden. Die Bestimmung des Typs eines Objektes wird durch die Operation type() in der Klasse MetaJFS realisiert, die den durch das MOP von Java bereitgestellten Link von einer Instanz zum Klassenobjekt der instanziierenden Klasse und von diesem zu den Interfaceobjekten gebraucht. Die Operation type() basiert direkt auf Methoden von `java.lang.Class` und berechnet für ein Objekt vom Typ JFS dessen Interfaceobjekt.

instance() basiert auf dem MOP `java.lang.Class`, um das durch den Namen identifizierte Klassenobjekt zu laden und davon eine Instanz zu bilden.

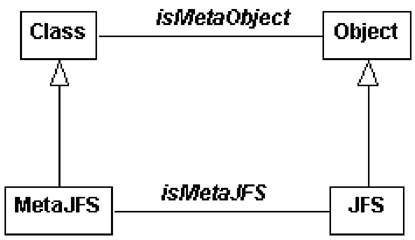

Abbildung 4: MetaJFS ist eine Spezialisierung des MOP von Java

So wie der Typ JFS eine Spezialisierung des Objektkonzeptes von Java darstellt, kann MetaJFS als eine Spezialisierung der Metaklasse `java.lang.Class` aufgefaßt werden. `java.lang.Class` ist zwar als `final` deklariert, was besagt, daß von `java.lang.Class` nicht geerbt werden kann, Spezialisierung muß jedoch nicht notwendig durch Vererbung realisiert werden, sondern kann z.B. auch durch Einbettung und Delegation implementiert werden[24]. Abb.4 zeigt den Zusammenhang der Typen, wie er sich aus dieser Beobachtung ergibt.

2.1.1 API Dokumentation der Komponenten von JTFS

Class jtfs.Unificator

```
java.lang.Object
   |
   +----jtfs.Unificator
```

---

[24] S. hierzu [Odell, 1988]: "*delegation to a hidden class*" und [Martin, Odell, 1998].

public class Unificator extends Object

Die Klasse Unificator implementiert eine Unifikationsmethode für FS. Die Unifikationsoperation berechnet den Unifikator oder setzt eine Variable, wenn die Eingabe inkonsistent ist und die Operation fehlschlägt. Bei erfolgreicher Unifikation wird der Unifikator in einer Variablen gespeichert und das Ergebnis kann durch die Zugriffsmethoden result() und isFail() abgefragt werden. Eine einstellige Unifikationsmethode ist definiert und ermöglicht es, sukzessive die intern gespeicherte Ergebnisstruktur durch Unifikation zu erweitern, bis das Ergebnis durch einen Aufruf von reset() gelöscht wird.

Variable

protected boolean thisFail
: ist ein Flag. Der Wert ist `true`, wenn die Unifikationsoperation erfolgreich war und `false` sonst.

protected JFS thisUnifier
: Variable speichert Ergebnisstruktur der Unifikation. Der Wert ist undefiniert, falls die Unifikation fehlschlägt.

Methoden

public boolean isFail()
: gibt aktuellen Wert des Flags **thisFail** zurück.

public JFS result()
: gibt aktuellen Wert der Unifikationsoperation zurück.

public JFS reset()
: setzt die Variablen auf ihre Defaultwerte zurück.

public void unify(JFS fs1, JFS fs2)
: unifiziert die FS **fs1** mit **fs2**. Die resultierende Struktur wird in einer Variablen gespeichert und bei einem Fehlschlagen ein Flag gesetzt.

public void unify(JFS fs)
: unifiziert die FS fs mit dem Resultat der vorhergehenden Unifikation, das in einer Variablen gespeichert ist.

Interface jtfs.MetaJFS

Class jtfs.MetaJFSImpl

```
java.lang.Object
   |
   +----jtfs.MetaJFSImpl
```

---

public interface MetaJFS

public class MetaJFSImpl extends Object implements MetaJFS

MetaJFSImpl implementiert Basisoperationen für Typen und erweitert konzeptionell die Klasse `java.lang.Class`. `java.lang.Class` ist zwar als `final` deklariert, so daß nicht von ihr geerbt werden kann, eine Erweiterung ist aber dennoch durch Delegation möglich. MetaJFSImpl implementiert die Operation glb() zur Berechnung des Infimum für zwei gegebene Typen im Typverband. Eine Tabelle repräsentiert die Assoziation eines Typs mit seinen Subtypen und erweitert dadurch das MOP von Java. Die Methode initTable() geht davon aus, daß die Verbandstruktur sowohl ein bottom als auch ein top Element enthält.

Variable

static protected Class thisBot
enthält das $\perp$ Element der Hierarchie der Typen, das bei der Initialisierung mit einer Typhierarchie von der Methode initTable() gesetzt wird.

static protected Class thisTop
$\top$ Element der Hierarchie der Typen (...s.o.).

static protected Hashtable types
Tabelle realisiert Abbildung Type $\longrightarrow$ Subtypes. Die Tabelle wird durch die Methode **initTable()** initialisiert. Interfaceobjekte werden als Schlüssel in die Tabelle eingetragen und verweisen auf Vektoren (`java.util.Vector`), die die Interfaceobjekte der Subinterfaces des Schlüssels enthalten.

Methoden

static public Class Bot(), static public Class Top()
geben die jeweiligen Interfaceobjekte zurück, die $\top$ und $\perp$ der Hierarchie von Typen repräsentieren.

static public Class glb(Class i1, Class i2)
Berechnet ein Interfaceobjekt **i** ausgehend von Interfaceobjekten **i1** und **i2** unter Verwendung der Tabelle types. **i** repräsentiert das Infimum der durch **i1** und **i2** dargestellten Typen in der Hierarchie.

static public void initTable(Class bot, Class top)
initialisiert die Tabelle **types** und setzt die Variablen **thisBot** und **thisTop**. Die Methode verwendet `getInterfaces()` aus der Klasse `java.lang.Class` um beim Traversieren der Interfacehierarchie auf die Superinterfaces zuzugreifen.

static public Class instance(Class i)
instanziiert das Interfaceobjekt **i**. **instance()** verwendet die Methode `forName(String)`, um ein Klassenobjekt zu erzeugen, dessen Name durch die Konkatenation mit dem Suffix "`Impl`" aus dem Namen des Interfaceobjektes **i** abgeleitet wird, und die Methode `newInstance()`, um eine Instanz von der Klasse zu bilden. Beide Methoden stammen aus `java.lang.Class`.

Interface jtfs.JFS

Class jtfs.JFSImpl

```
java.lang.Object
   |
   +----jtfs.JFSImpl
```

---

public interface JFS

public class JFSImpl extends Object implements JFSImpl

JFS und JFSImpl realisieren FS als *Composite* Entwurfsmuster (S. [Gamma *et al.*, 1995]). Die Klasse verwendet Methoden von MetaJFS sowie von `java.lang.Class` und `java.lang.reflect.Method`, um durch Introspektion auf die Zugriffsmethoden der eigenen Attribute zugreifen zu können.

Methoden

public String toString()
überschreibt die Methode `toString` in der Klasse `Object`.

public void accept(FSVisitor visitor)
implementiert die *Accept* Operation für *Visitor* (S. *Visitor* Entwurfsmuster in [Gamma *et al.*, 1995]).

public Iterator newIterator()
erzeugt einen Iterator für das Objekt.

public SetIterator newSetIterator()
erzeugt einen Iterator zum Setzen von Attributen für das Objekt.

protected Vector features()
gibt einen Vektor der Methoden des Objektes zurück. Die Methode verwendet type() aus MetaJFS, um auf den Typ des Objektes zuzugreifen. Durch `getMethods()` aus `java.lang.Class` wird ein Vektor der eigenen Methodenobjekte erzeugt und durch Methoden der Klasse gefiltert. Hierdurch lassen sich z.B. *set*- von *get*-Methoden unterscheiden, um die jeweiligen Iteratoren zu initialisieren.

Interface jtfs.Iterator

---

public interface Iterator

Iterator realisiert das *Iterator* Entwurfsmuster (S. [Gamma *et al.*, 1995]) für FS. *current* ist ein Zeiger auf einen Vector von Methodenobjekten, die den Zugriff auf die Struktur FS realisieren. Der Zugriff wird durch die Methode `invoke( Object, Object[])` der Klasse `java.lang.reflect.Method` realisiert.

Methoden

public abstract void first()
setzt den Zeiger *current* zurück auf das erste Element der Struktur.

public abstract void next()
erhöht den Zeiger *current* um eins.

public abstract boolean isDone()
überprüft, ob die Struktur durchlaufen wurde.

public abstract JFS currentItem()
gibt die Struktur zurück, auf die der Zeiger *current* zeigt.

public abstract FS currentItem(JFS fs)
gibt die Struktur innerhalb der Struktur fs zurück, indem die Zugriffsmethode bei fs aufgerufen wird, auf die der Zeiger *current* zeigt.

2.1.2 Spezifikation für die Anwendung

Durch Spezialisierung des Typs FS wird das Paket JTFS zur Modellierung verwendet. Das OOD der JTFS zeigt, daß bei der Anwendung verschiedene Konventionen zu beachten sind, die die beschriebene Implementierung in Java ermöglichen. Auf diese Konventionen wurde im Rahmen der Beschreibung des OOD an verschiedenen Stellen hingewiesen. Als Spezifikation für eine Anwendung werden sie im Folgenden kurz zusammengefasst:

- Die modellierte Typhierarchie muß ein $\perp$- und ein $\top$-Element enthalten. Die Methode initTable(Class bot, Class top) aus MetaJFS wird mit dem $\perp$- und $\top$-Element aufgerufen, um eine Repräsentation der Hierarchie zu konstruieren, die Typinferenz ermöglicht.
- Typen einer Modellierung werden als Interfaces und implementierende Klassen in Java definiert. Der Name einer implementierenden Klasse ist der Name des Interfaces, der um das Suffix "`Impl`" erweitert ist.
- Typen enthalten Attribute als Variable und Methoden zum Zugriff sowie zum Setzen der Variablen. Die Signatur der Methoden hat folgende Form:
  - `X x()` für die das Attribut `x` vom Typ `X` lesende Methode und
  - `void x(X value)` für die das Attribut `x` vom Typ `X` setzende bzw. modifizierende Methode.
- Mehrfachvererbung von Interfaces wird als Delegation implementiert.

2.1.3 Der Typ TID

Die Beschreibung eines Intervalls ist als folgende JFS vom Typ TID[25], bestehend aus zwei Datumsangaben vom Typ Date, gegeben:

$$\begin{bmatrix} \mathsf{TID} & \\ \text{BEG:} & \begin{bmatrix} \mathsf{Date} \end{bmatrix} \\ \text{END:} & \begin{bmatrix} \mathsf{Date} \end{bmatrix} \end{bmatrix}$$

Abbildung 5: Der Typ TID

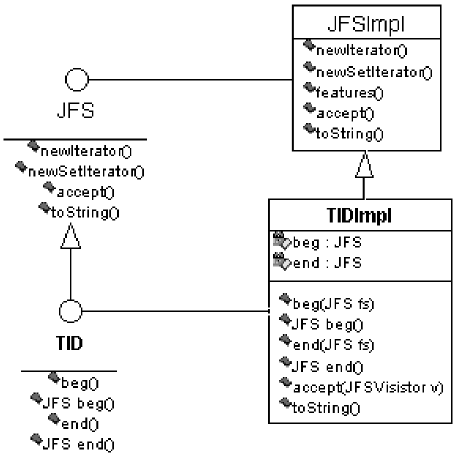

Abbildung 6: Modellierung von TID als JFS

Eine Intervallbeschreibung besteht also aus der Angabe der Eckpunkte, die selbst strukturierte JFS vom Typ Date sind. Die vollständige Beschreibung des Typs Date erfolgt weiter unten. Er ist aus den im nächsten Abschnitt behandelten atomaren Typen zusammengesetzt, die im wesentlichen auf Elementen der ZG beruhen.

Abb.6 zeigt die Modellierung des Typs TID als JFS. Diese Darstellung ist kanonisch auf die im weiteren spezifizierten Typen übertragbar. Mittels des Modellierungswerkzeugs *Rational Rose*[26], das zur Erstellung der Diagramme verwendet wird, läßt sich direkt aus der graphischen Repräsentation der Klassen und Interfaces Javacode erzeugen. Die übersetzten Klassen implementieren das Domänenmodell und werden von dem Unifikator TIDU und weiteren Komponenten zur Konstruktion und Verarbeitung verwendet.

[25] Time Interval Description

[26] http://www.rational.com/rose

2.1.4 Atomare Typen

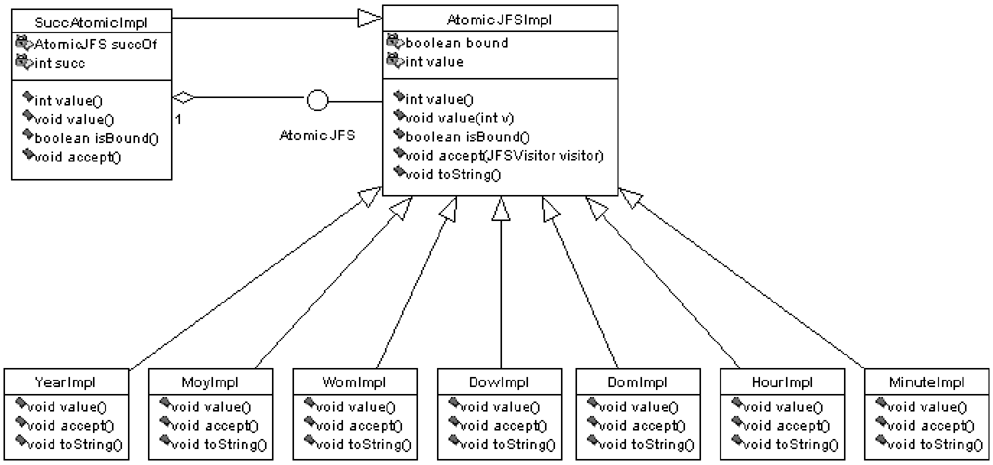

Abbildung 7: Modellierung der atomaren Typen

Atomare Typen sind dadurch ausgezeichnet, daß für sie keine Attribute definiert sind. Um die Basiskategorien der ZG zu modellieren reichen ganze Zahlen als atomare Typen aus. Diese sind in folgende Klassen zusammengefasst:

- $\boldsymbol{x}$:Year $\Rightarrow \boldsymbol{x} \in [-\infty, +\infty]$[27]
- $\boldsymbol{x}$:Moy $\Rightarrow \boldsymbol{x} \in [1, 12]$
- $\boldsymbol{x}$:Wom $\Rightarrow \boldsymbol{x} \in [1, 5]$
- $\boldsymbol{x}$:Dom $\Rightarrow \boldsymbol{x} \in [1, 31]$
- $\boldsymbol{x}$:Dow $\Rightarrow \boldsymbol{x} \in [0, 6]$
- $\boldsymbol{x}$:Hour $\Rightarrow \boldsymbol{x} \in [0, 23]$
- $\boldsymbol{x}$:Minute $\Rightarrow \boldsymbol{x} \in [0, 59]$

Hierbei werden die Wochentage {mon, tue, wed, thu, fri, sat, sun} der ZG als ganze Zahlen repräsentiert. Die atomaren Typen des temporalen Modells sind Spezialisierungen des Typs AtomicJFS der JTFS. AtomicJFS haben wie in Abb.7[28] dargestellt ein Feld value vom Typ int, in dem der Wert gespeichert wird, und Zugriffsoperationen sowie ein Flag, das anzeigt, ob das Feld definiert

[27] Dieser Bereich wird praktisch durch eine Kalenderimplementierung auf ein definiertes Maximum und Minimum eingeschränkt.

[28] Die Funktion der Klasse SuccAtomicImpl wird auf S.24 erklärt.

ist. Wenn kein Wert gesetzt ist, ist die AtomicJFS eine Variable und darf bei der Unifikation mit einem Wert überschrieben werden. Es ist nur die Darstellung der implementierenden Klassen im Diagramm der Abb.7 wiedergegeben. Dabei werden die Zugriffsmethoden von AtomicJFSImpl auf den Wert value so überschrieben, daß die mit den Typen spezifizierten Wertebereiche überprüft werden.

Die atomaren Typen lassen sich gemäß ihrer Spezifität in folgende Ordnung Spec bringen:

$$\mathsf{Year} >_{\mathsf{Spec}} \mathsf{Moy} >_{\mathsf{Spec}} \mathsf{Wom} >_{\mathsf{Spec}} \mathsf{Dow} >_{\mathsf{Spec}} \mathsf{Dom} >_{\mathsf{Spec}} \mathsf{Hour} >_{\mathsf{Spec}} \mathsf{Minute}$$

Dabei sei $>_{\mathsf{Spec}}$ als transitive Hülle definiert. Diese totale Ordnungsrelation entspricht der Spezifität insofern, als daß sie die Relation des *Enthaltenseins* der kalendarischen Einheiten wiederspiegelt. Etwas konstruiert ist der Vergleich von Dow und Dom, läßt sich aber dadurch motivieren, daß innerhalb eines Monatsrahmens ein Dow wie z.B. mon $(= 0)$ mindestens viermal vorkommt im Gegensatz zu einmal von einem Dom. Die Ordnung Spec steuert den weiter unten angegebenen Unifikationsalgorithmus und ist wesentlich bei dem Vergleich des Informationsgehaltes zweier Strukturen. Spec wird im nächsten Abschnitt auf komplexe Typen erweitert.

2.1.5 Komplexe Typen

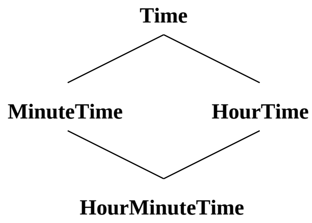

Abbildung 8: Typhierarchie für Time

Komplexe Typen unterscheiden sich von atomaren Typen dadurch, daß sie Attribute definieren. So hat z.B. eine Struktur vom Typ MinuteTime ein Objekt vom (atomaren) Typ Minute als Wert des Attributs MINUTE. Ausgehend von der Definition der atomaren Typen, können nun Typen für komplexe Strukturen, die die atomaren Typen enthalten, beschrieben werden.

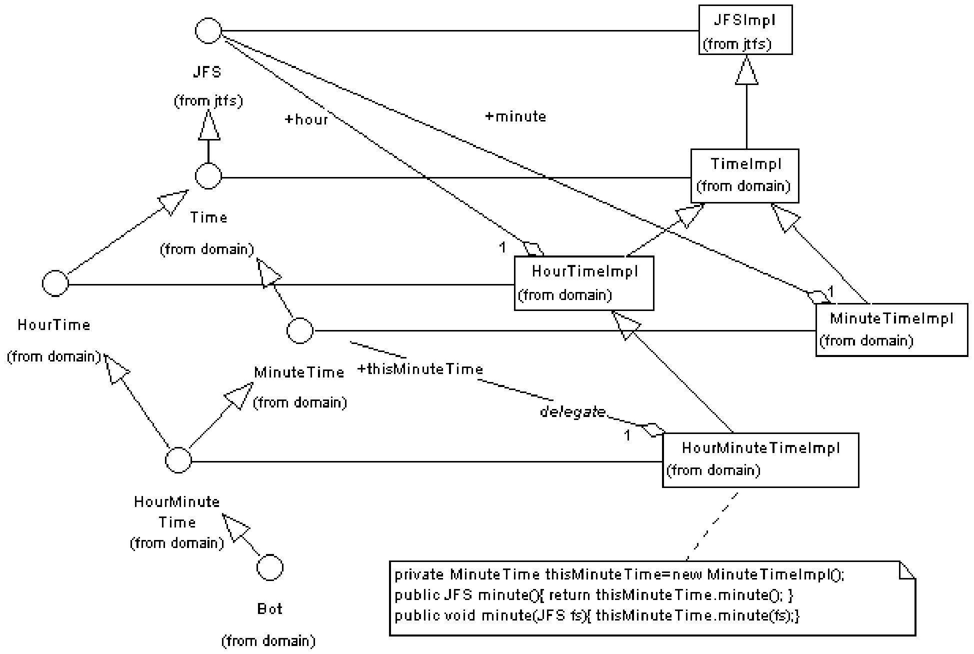

Abbildung 9: Modellierung von Time als JFS

$$\left[\,\mathsf{Time}\,\right] \quad \begin{bmatrix}\mathsf{MinuteTime}\\ \text{MINUTE: } \left[\,\mathsf{Minute}\,\right]\end{bmatrix} \quad \begin{bmatrix}\mathsf{HourTime}\\ \text{HOUR: } \left[\,\mathsf{Hour}\,\right]\end{bmatrix} \quad \begin{bmatrix}\mathsf{HourMinuteTime}\\ \text{HOUR: } \left[\,\mathsf{Hour}\,\right]\\ \text{MINUTE: } \left[\,\mathsf{Minute}\,\right]\end{bmatrix}$$

Abbildung 10: Typdefinitionen für Time

Das in der Abb.9 abgebildete UML-Diagramm zeigt die Implementierung der Typhierarchie von Time aus Abb.8. Das Diagramm stellt dar, wie der Vererbungsmechanismus dazu dient, Attributdefinitionen weiterzureichen und so das Verhältnis unterschiedlich informativer Strukturen bestimmt. Die multiplen Supertypen von HourMinuteTime werden sowohl durch Klassenvererbung als auch durch Delegation implementiert. Durch mehrfache Aggregationsbeziehungen wird so durch die Komposition von Objekten multiple Klassifikation implementiert, wie es in [Laffra, 1997] auf S.20f unter dem Titel "*Implementing Multiple Inheritance*" beschrieben wird Um das Diagramm übersichtlich zu halten, wurden die Signaturen der Zugriffsmethoden der Attribute und Felder nicht dargestellt. Diese sind direkt aus den Typspezifikationen in Abb.10 basierend auf den Vorgaben für die Modellierung mittels JTFS des vorhergehenden Kapitels abzuleiten.

Die Definitionen der Typen WomMoyMonth und DomDowDay aus Abb.11 sind ausgehend von den Typen Month und Day in der gleichen Weise gebildet wie der Typ HourMinuteTime in Abb.10.

$$\begin{bmatrix} \textsf{WomMoyMonth} \\ \text{MOY: } [\,\textsf{Moy}\,] \\ \text{WOM: } [\,\textsf{Wom}\,] \end{bmatrix}, \quad \begin{bmatrix} \textsf{DomDowDay} \\ \text{DOM: } [\,\textsf{Dom}\,] \\ \text{DOW: } [\,\textsf{Dow}\,] \end{bmatrix}$$

Abbildung 11: Subtypen von Month und Day für vollständige Monats- und Tagesangaben

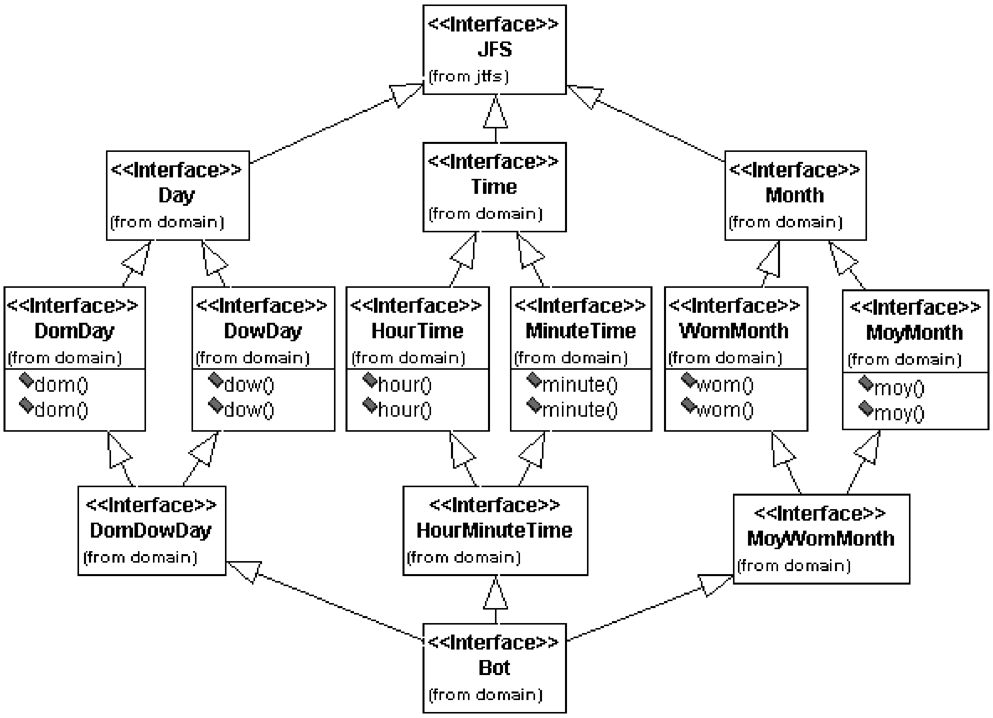

Abbildung 12: Modellierung der Typhierarchien von Day, Time und Month

Den Gesamtüberblick der Typhierarchie zeigt Abb.12 als UML-Diagramm der Subsumptionshierarchie der Interfaces der Typen. Die Klassendefinitionen sind nicht angegeben, da die Implementierung der Interfaces kanonisch dem Schema folgt, das im zweiten Abschnitt des vorhergehenden Kapitels beschrieben wurde und in Abb.9 als Beispiel angegeben wird.

$$[\,\mathsf{Date}\,]\ ,\quad \begin{bmatrix}\mathsf{TimeYearDate}\\ \text{YEAR: }[\,\mathsf{Year}\,]\\ \text{TIME: }[\,\mathsf{Time}\,]\end{bmatrix}\ ,\quad \begin{bmatrix}\mathsf{TimeDayMonthYearDate}\\ \text{YEAR: }[\,\mathsf{Year}\,]\\ \text{MONTH: }[\,\mathsf{Month}\,]\\ \text{DAY: }[\,\mathsf{Day}\,]\\ \text{TIME: }[\,\mathsf{Time}\,]\end{bmatrix},\ldots$$

Abbildung 13: Typdefinitionen für Date

Die für die Modellierung von Kalenderintervallen wesentlichen Eckpunkte in Form von Datumsangaben sind vom Typ Date, der aus den Typen Time, Month usw. besteht und wie in Abb.13 strukturiert ist. Wie Abb.14 zeigt ist hier jeder Kombination der Merkmale YEAR, MONTH, DAY und TIME ein Typ zugeordnet, da dies zur Bestimmung der Informativität einer JFS von Typ Date notwendig ist. Dem Typ Date entspricht die leere Struktur.

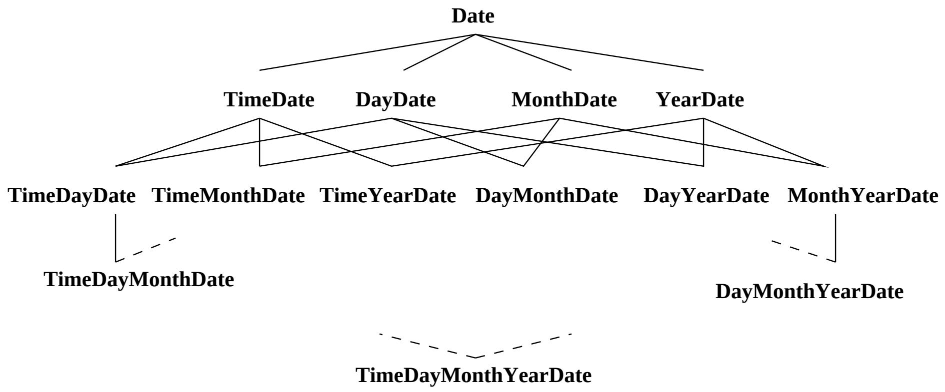

Abbildung 14: Vererbungshierarchie für Date

Erweiterung von Spec auf Komplexe Typen

Ein Typ $\boldsymbol{\tau}$ enthält einen Typ γ, wenn $\boldsymbol{\tau}$ für ein Attribut A definiert ist, dessen Wertebereich entweder vom Typ γ ist oder der Wertebereich ist vom Typ $\boldsymbol{\sigma}$ und $\boldsymbol{\sigma}$ enthält γ. Durch die Transitivität der Attribut-Wert-Assoziation läßt sich Spec leicht von atomaren auf komplexe Typen generalisieren. Es gilt für komplexe Typen $\boldsymbol{\sigma} <_{\mathsf{Spec}} \boldsymbol{\tau}$, gdw. es einen atomaren Typ $\boldsymbol{\alpha}$ in $\boldsymbol{\sigma}$ gibt, so daß für alle atomaren Typen $\boldsymbol{\delta}$ in $\boldsymbol{\tau}$ gilt: $\boldsymbol{\alpha} <_{\mathsf{Spec}} \boldsymbol{\delta}$.

Für jede JFS vom Typ Date ist also das in Hinsicht auf die Granularität des Bezugssystem maximale und damit unspezifischste atomare Element eindeutig durch den Typ bestimmt. Im weiteren bezeichne $max_{\mathsf{Spec}}(fs)$ das in der FS fs enthaltene Atom, dessen Typ maximal bezüglich Spec innerhalb von fs ist.

2.1.6 Die Nachfolgefunktion *succ*

Um relationale Einschränkungen wie z.B. Zeitdauern repräsentieren zu können, ist für Typen $\boldsymbol{t}$ = Year, Moy, ... die partielle Abbildung $succ(\boldsymbol{t})$:$\boldsymbol{t}$ bzw. $succ(\boldsymbol{t}, n$:int$)$:$\boldsymbol{t}$ mit $n \in N$ für den n-ten Nachfolger definiert. Hierbei ist darauf zu achten, daß ein bei einer Berechnung auftretender Werteüberlauf korrekt innerhalb der JFS zur nächst-weniger spezifischen Teilstruktur propagiert wird. D.h. die Berechnung von *succ*(12:Moy) beinhaltet, daß *succ*(X:Year) innerhalb der betreffenden JFS berechnet wird[29]. Dabei muß evtl. ein neuer Typ inferiert werden, wenn wie z.B. *succ*(12:Moy) berechnet wird und YEAR in der Struktur nicht gegeben ist.

Allgemein kann *succ* nur berechnet und durch den Wert ersetzt werden, wenn genügend Information zur Bestimmung der Werte in der Struktur gegeben ist. Da Strukturen durch Unifikation an Information zugewinnen, kann der Wert evtl. auch erst später innerhalb einer Berechnung bestimmt werden.

Ein Problem stellen dabei die Objekte vom Typ Dom dar, da die zulässigen Werte nicht lokal bestimmt werden können, sondern vom betreffenden Monat abhängen. Hierfür sollte *succ* wie folgt parametrisiert sein:

succ(Dom, Moy):Dom, bzw. *succ*(Dom, int, Moy):Dom.

Die in Abb.7 dargestellte Modellierung der atomaren Strukturen zeigt, wie durch die Klasse SuccAtomicImpl die Abbildung *succ* als AtomicJFS + succ repräsentiert wird. SuccAtomicImpl enthält eine AtomicJFS und ein Feld succ vom ganzzahligen Typ int. Die Zugriffsmethoden value der Klasse AtomicJFSImpl werden so überschrieben, daß sie den Wert des eingebetteten Objektes um den Wert des Feldes succ erhöhen. Da die Klasse SuccAtomicImpl das gleiche Interface wie AtomicJFSImpl implementiert, kann eine Instanz nach der Berechnung des Nachfolgers der eingebetteten AtomicJFSImpl durch eine AtomicJFSImpl mit dem inkrementierten Wert in einer JFS ersetzt werden. Die Propagierung von einem Werteüberlauf wird durch einen Visitor realisiert, der die gesamte Objekthierarchie rekursiv durchläuft und die entsprechenden Operationen auf den Teilobjekten durchführt. Hierbei werden Werte entweder direkt berechnet oder in Instanzen von SuccAtomicImpl eingebettet, wobei der zu inkrementierende Wert im Feld succ gespeichert ist.

2.1.7 Disjunktion

Disjunktionen als Werte von Attributen werden als Mengen von JFS eines Typs modelliert und in der AVM Notation durch geschweifte Klammern ({}) dargestellt. Eine Disjunktion ist vom gleichen Typ wie die Elemente der Menge, die

[29] Dies kann durch die implizite Parametrisierung mit einer Referenz auf die Gesamtstruktur realisiert werden.

sie repräsentiert. Sie kann also insbesondere auch an eingebetteter Stelle innerhalb einer Struktur auftreten, an der ein Element der Disjunktion zugelassen ist. Solche verteilten Disjunktionen erlauben es, platzökonomisch Ambiguitäten darzustellen, erhöhen jedoch den Aufwand, der bei der Unifikation getrieben werden muß.

Die Möglichkeit, Disjunktion innerhalb der als Konjunktion aufzufassenden JFS auszudrücken, ist in Hinsicht auf verschiedene Repräsentationsprobleme nützlich. So läßt sich z.B. eine ao-Liste oder der ZG-Ausdruck [`dow:weekday`][30] als Menge von Objekten des entsprechenden Typs darstellen.

Disjunktion werden auf atomare Typen und den Typ TID für die Darstellung von Auswahllisten eingeschränkt, um den Unifikationsalgorithmus nicht unnötig kompliziert zu gestalten. Insbesondere wird durch die Einschränkung der Disjunktion auf atomare Typen vermieden, daß durch Pfadgleichungen bzw. Koreferenzen aus der Disjunktion in entfernte Teile der Struktur die disjunktive Information *exportiert* wird[31]. Indem die Distributivität der Disjunktion bei der Unfikation ausgenutzt wird, läßt sich der Unifikator durch einfaches Ausmultiplizieren berechnen.

Für TID und jeden der atomaren Typen werden Listentypen eingeführt, die einen Vektor von JFS des entsprechenden Typs enthalten. Die Listentypen implementieren die Interfaces der Elemente der Liste und können so stattdessen in einer Objekthierarchie verwendet werden. Der Unifikator überprüft, ob ein zu unifizierendes Objekt einen Listentyp hat. Hierzu werden die Iteratoren um eine Testmethode und ein Flag erweitert. Eine Instanz des Listentyps erzeugt einen Listeniterator, der die Elemente des Vektors aufzählt und das Flag setzt, das von dem Testprädikat überprüft wird.

2.2 Kalendermodell für TID

Zur Interpretation der temporalen Ausdrücke soll hier ein Zeitmodell auf der Basis eines Kalenders konstruiert werden. Die durch das Typsystem gegebenen Beschreibungen werden durch das Modell interpretiert und erhalten so eine klare Semantik.

Sei $\mathcal{U}$ die Menge aller Graphstrukturen mit Knoten N und Kanten $\mathsf{K} \subseteq \mathsf{N} \times \mathsf{N}$, die sich als legale Kombinationen von Attributen und Werten durch den Kalender erzeugen lassen. Die Kanten K sind mit Attributen des Kalenders wie Jahr, Tag und Monat assoziiert, so daß sie sich mit Elementen aus der Menge der Attribute $\mathsf{Feat} = \{\text{YEAR, MONTH, DAY, HOUR, MINUTE}, \ldots\}$ der Typen der Modellierung beschriften lassen. Somit entsprechen die Graphstrukturen strukturell den im vorangegangenen Kapitel definierten maximalen Typen. Es sind dann JFS vom Typ Date partielle Beschreibungen der Elemente in $\mathcal{U}$ und damit ist $\mathcal{U}$ eine Interpretation für JFS vom Typ Date.

[30] {mon,tue,wed,thu,fri}

[31] Siehe hierzu [Eisele, Dörre, 1990].

Aus der Konstruktion von $\mathcal{U}$ ergibt sich eine injektive Zuordnung der Typen von JFS zu den jeweiligen Knoten der Graphen in $\mathcal{U}$. Damit kann die Relation before[32] $\subset \mathcal{U} \times \mathcal{U}$ folgendermaßen definiert werden:
$(u, v) \in$ before gdw. $a < b$ mit

- $a \in u$, $b \in v$ sind atomare Knoten gleichen Typs $\boldsymbol{\tau}$
- für jeden atomaren Knoten $a_i \in u$ vom Typ $\boldsymbol{\sigma}$ mit $\boldsymbol{\sigma} >_{\mathsf{Spec}} \boldsymbol{\tau}$ gibt es Knoten $b_i \in v$ vom Typ $\boldsymbol{\sigma}$, so daß $a_i = b_i$

Dann ist eine Interpretation für JFS vom Typ TID gegeben als die Menge $\mathcal{I} \subset \mathcal{U} \times \mathcal{U}$ mit $(u, v) \in \mathcal{I}$ gdw. $(u, v) \in$ before oder u = v.

2.2.1 Interpretationsmöglichkeiten für partielle Beschreibungen

Intervallbeschreibungen, die unvollständige Datumsangaben enthalten, die aber gemäß Spec absteigend bei Year beginnend definiert sind, wie z.B. YearDate:f, YearMonthDate:f, ..., lassen sich zu vollständig definierten Angaben ergänzen. Hierzu werden die fehlenden Werte mittels des Kalenders, der bestehenden Struktur f und des Typsystems inferiert. Es handelt sich dabei um maximal-, bzw. minimal-Werte – je nach dem, ob es sich bei der unterspezifizierten Teilstruktur von f um ein BEG- oder END-Datum handelt – um die f dann durch Unifikation ergänzt wird.

Ausdrücke, die Datumsangaben enthalten, in denen unspezifische Felder undefiniert, spezifische jedoch definiert sind, wie z.B. bei[33]:

$$\begin{bmatrix} \mathsf{TID} & \\ \text{BEG:} & \begin{bmatrix} \mathsf{TimeMonthDate} \end{bmatrix} \\ \text{END:} & \begin{bmatrix} \mathsf{TimeMonthDate} \end{bmatrix} \end{bmatrix}$$

können durch Unifikation mit unspezifizierten Strukturen der komplementären Datumstypen auf ähnliche Weise ergänzt werden. Der komplementäre Typ $\overline{\boldsymbol{\tau}}$ von $\boldsymbol{\tau}$ ist der Typ, der für genau die Felder definiert ist für die $\boldsymbol{\tau}$ undefiniert ist. So ist z.B.:

$$\overline{\mathsf{TimeMonthDate}} = \mathsf{DayYearDate}$$

Auf diese Weise erhält man also Strukturen, die anstatt der Leerstellen Variablen enthalten, was durchaus sinnvoll ist, da Variablen sich als existenzquantifiziert – und damit als Teilmengen von $\mathcal{U}$ – interpretieren lassen.

Die uniforme Darstellung des semantischen Gehaltes der Ausdrücke von ZG als Intervall-Beschreibungen ermöglicht eine Abbildung auf Zeitintervalle und liefert damit eine präzise Semantik für die durch ZG beschriebene Sprache $L \subset \mathcal{L}(ZG)$. Das der Interpretation zugrundeliegende Kalendermodell gewährleistet, daß syntaktisch zwar korrekte Strukturen, wie sie z.B. für "31. Februar" generiert werden, als nicht erfüllbar erkannt werden können.

[32] Zu dem Vergleich von Wochentagen s. S. 33.
[33] $\begin{bmatrix} \mathsf{TimeMonthDate} \end{bmatrix}$ ist für Attribute TIME und MONTH definiert.

2.3 Unifikation für TID

Die für JFS definierte Unifikationsoperation muß für die hier intendierte Darstellung von Intervallen erweitert werden; denn es soll eine Beschreibung eines in einem Intervall enthaltenen Intervalls als Spezialisierung des äußeren interpretiert werden und von daher mit dessen Repräsentation unifizieren. Um dies zu gewährleisten, wird ein spezieller Unifikationsalgorithmus für JFS vom Typ Date benötigt, der im folgenden spezifiziert wird.

2.3.1 Definitionen

Es seien Type[34] und Feat die gegebenen Mengen von Typen und Attributen sowie *sub*: $\mathsf{Type} \longrightarrow \mathrm{P}(\mathsf{Type})$ die Abbildung, die jedem Typ die Menge seiner Untertypen vereinigt mit der Menge der in ihm enthaltenen Typen zuordnet. Die Menge der in einem Typ $\boldsymbol{\tau}$ enthaltenen Typen ist gegeben durch die Menge all der Typen, die über die Attribute erreichbar sind, für die $\boldsymbol{\tau}$ definiert ist. Die Menge der Untertypen ist die Menge aller Typen, die in der transitiven Subtyp-Relation zu gegebenem Typ $\boldsymbol{\tau}$ stehen und *sub* berechnet die Vereinigung von beiden Mengen.

Ferner seien die Mengen[35]:

$$\begin{aligned}
\mathsf{IdFS} &= \{f \mid f : \mathsf{TID}\} \\
\mathsf{DateFS} &= \{f \mid f : \boldsymbol{t},\ \boldsymbol{t} \in sub(\mathsf{Date})\} \\
\mathsf{Rest} &= \{(r_1 : \boldsymbol{t},\ r_2 : \boldsymbol{t}) \mid\ r_{1,2} \in \mathsf{DateFS},\ \text{und} \\
&\qquad max_{\mathsf{Spec}}(r_1) \neq max_{\mathsf{Spec}}(r_2)\} \cup \{()\}
\end{aligned}$$

und die Relationen:

- before $\subset$ Rest, mit
  $(d_1, d_2) \in \mathsf{before}$, gdw. $max_{\mathsf{Spec}}(d_1) < max_{\mathsf{Spec}}(d_2)$
- dur[36] $\subset \mathsf{Rest} \times \mathsf{Rest}$, mit
  $((c_1, c_2), (d_1, d_2)) \in \mathsf{dur}$, gdw.
  - $c_1 = c_2$ oder $(c_1, c_2) \in \mathsf{before}$ und
  - $d_1 = d_2$ oder $(d_1, d_2) \in \mathsf{before}$ und
  - $c_1 = d_1$ oder $(d_1, c_1) \in \mathsf{before}$ und
  - $c_2 = d_2$ oder $(c_2, d_2) \in \mathsf{before}$

[34]Mit $\perp$ für den inkonsistenten und $\top$ für den uninformativen Typ.
[35]max_{Spec} s. S. 23.
[36]during

gegeben.

Die Mengen IdFS und DateFS sind Mengen getypter Merkmalstrukturen und werden definiert, um im weiteren einen Unifikationsalgorithmus spezifisch für diese Mengen beschreiben zu können. Rest ist eine Menge von Paaren von JFS, wobei für $(f_1,f_2) \in$ Rest gilt, daß das maximal unspezifischste Merkmal der beiden Strukturen nicht unifiziert. In Rest sind also Paare nicht unifizierbarer JFS enthalten, wodurch sich Reste für Teilunifikation angeben lassen, wobei () für einen leeren Rest steht.

Die Relation before unterscheidet sich von der Definition auf Seite 26 für Modelle, da $<$ auf Funktionen, Variablen und Disjunktionen von Atomen erweitert ist. So ist z.B. $(\boxed{x}, succ(\boxed{x})) \in$ before aber $(\boxed{y}, succ(\boxed{x})) \notin$ before, wenn $\boxed{y}$ nicht gebunden ist. Für Disjunktionen m, n gilt $(m,n) \in$ before gdw. $m \cap n = \emptyset$ und für alle $m_i \in m$ gilt $(m_i, n_{\mathrm{min}}) \in$ before, es muß also für alle Disjunkte m_i aus m gelten, daß kein n_j aus n davor liegt.

Die Relation dur ist die Inklusionsrelation für Intervalle, also die Menge aller Paare von Intervallen (i_1, i_2), für die gilt, daß i_1 innerhalb von i_2 liegt. Das Inklusionsverhältnis dur wird durch die before-Relation bezogen auf die Eckpunkte bestimmt.

Folgende Funktionen seien zur Angabe des Unifikationsalgorithmus definiert:

$$\begin{aligned}
&selectA\text{: DateFS} \times \text{DateFS} \longrightarrow \text{Feat}\\
&subType\text{: DateFS} \times \text{DateFS} \longrightarrow \text{Type}\\
&unifyFun\text{: DateFS} \times \text{DateFS} \longrightarrow \text{(DateFS, Rest)}\\
&unifyMod\text{: DateFS} \times \text{DateFS} \longrightarrow \text{(DateFS, Rest)}
\end{aligned}$$

Das durch *selectA* ausgewählte Attribut $A \in$ Feat ist maximal bzgl. Spec. D.h. $A = selectA(f_1, f_2)$, gdw. für $f = \delta(A, f_i)$ gilt, daß $f = max_{\mathsf{Spec}}(f_i)$ und $max_{\mathsf{Spec}}(f_i) >_{\mathsf{Spec}} max_{\mathsf{Spec}}(f_j)$, mit i,j $\in \{1, 2\}$.

subType ist die Funktion, die als Wert für $f_{1,2} \in$ DateFS und Typen $\boldsymbol{\tau}$, $\boldsymbol{\sigma}$ mit $f_1{:}\boldsymbol{\tau}$ und $f_2{:}\boldsymbol{\sigma}$ den größten gemeinsamen Subtyp gemäß der Typhierarchie oder den inkonsistenten Typ $\perp$ liefert. *subType* realisiert insofern die bei Unifikation von JFS notwendige Typinferenz.

Ferner sei ein Kontext $\mathcal{K}$, eine Variablenbindung Θ und eine Operation *push* gegeben. $\mathcal{K}$ ist eine Struktur vom Datentyp Stapel zur Speicherung von Verarbeitungszuständen. Ein Verarbeitungszustand ist definiert als ein Paar bestehend aus dem Tripel von JFS der aktuell konstruierten bzw. durchwanderten beiden Eingabestrukturen und der Ausgabestruktur sowie der dazugehörenden Variablenbindung Θ.

push ist eine Operation auf Kontexten $\mathcal{K}$, die dem aktuellen Stack einen neuen Verarbeitungsszustand zusammen mit einer neu berechneten Bindung Θ hinzufügt. Die Argumente für *push* bestehen aus einem Tripel (d, f_1, f_2) von JFS, wobei d der Teilunifikator von f_1 und f_2 ist sowie einer Variablenbindung Θ und einem Kontext $\mathcal{K}$. *push* wird nur angewendet, wenn wenigstens ein f_i

eine Variable ist. Der Rückgabewert ist eine neue Variable, falls die (beiden) Eingabevariable(n) in Θ noch nicht gebunden ist (sind). *push* bindet die Eingabevariable(n) in Θ an die neue Variable oder liefert diese als Wert, falls die Bindung bereits durch einen früheren Aufruf von *push* mit gegebenem Θ und Variable(n) bereits erzeugt worden ist.

Θ ist eine Funktion, die Variablen v der Typen aus Type Werte zuordnet und deren Anwendung als $\Theta(v)$ geschrieben wird. Das Ergebnis ist der Wert der Bindung von v in Θ. Bei ungebundenen Variablen ist der Wert $[\top]$ und somit nie selbst eine Variable. Variablenbindungen untereinander werden nach den durch *push* neu erzeugten aufgelöst. Wird Θ auf eine JFS fs aus DateFS angewandt, dann ist der Wert fs die Identitätsabbildung.

Die Funktion *unifyFun* ist eine Hilfsfunktion, die Details der Unifikation von Funktionen realisiert, die als Terme innerhalb der JFS dargestellt werden. *unifyFun* berechnet den Wert der Funktion, falls dies möglich ist, und behandelt die evtl. auftretenden Fehler. Es muß dabei sichergestellt sein, daß als Terme gegebene Funktionen nur unifizieren, d.h. gleichgesetzt werden, wenn die Typisierung übereinstimmt.

unifyFun(f_1, f_2), geg: Kontext $\mathcal{K}$ mit Variablenbindung Θ

1. Wende Funktion(en) f_i an; bei Fehlschlag: return($[\top]$,(f_1, f_2))
2. Wenn f_i Variable und f_i kommt in Funktion f_j vor:
   return($[\top]$, (f_1, f_2))
   sonst: $\boxed{x} := push((f_j, f_1, f_2), \Theta, \mathcal{K})$;
   return($[\boxed{x} f_j]$, ())
3. Wenn f_i Funktionen oder ein Paar von Funktion und Atom, berechne Unifikator, erweitere Bindung falls erforderlich und gib Werte zurück
4. sonst: return($[\top]$, (f_1, f_2))

Abbildung 15: Algorithmus *unifyFun*

Für JFS f_1, f_2 ist

$$unifyMod(f_1, f_2) = \begin{cases} ((f_1 - r_1) \sqcup (f_2 - r_2), (r_1, r_2)) & \text{falls } r_1 \not\trianglelefteq r_2 \\ (f_1 \sqcup f_2, ()) & \text{sonst} \end{cases}$$

$f - r$ bezeichne hier die JFS, die man erhält, wenn man die durch r beschriebene Teiltruktur in f "subtrahiert", wie z.B.:

$$\begin{bmatrix} \mathsf{t} & \\ \mathrm{A}: & [\,\mathsf{v1}\,] \\ \mathrm{B}: & [\,\mathsf{v2}\,] \end{bmatrix} - \begin{bmatrix} \mathsf{t} & \\ \mathrm{B}: & [\,\mathsf{v2}\,] \end{bmatrix} = \begin{bmatrix} \mathsf{t} & \\ \mathrm{A}: & [\,\mathsf{v1}\,] \end{bmatrix}$$

2.3.2 Unifikation für JFS vom Typ Date

In Abb.16 wird der Algorithmus skizziert, womit die nicht-unifizierenden Reste $r_{1,2}$ und der Teilunifikator d von $f_{1,2}$ berechnet werden. Die zugrundeliegende Idee besteht darin, der Ordnung Spec folgend die zu unifizierenden Strukturen *top-down* zu traversieren und dabei sukkzessive eine neue Struktur bestehend aus den miteinander identifizierten Knoten der Ausgangsstrukturen zu konstruieren.

Unter 1. in Abb.16 werden die Abbruchbedingungen der Rekursion in einer Fallunterscheidung geprüft.

Funktionen müssen angewendet werden, bevor sie unifiziert werden können. Dies geschieht innerhalb der Hilfsfunktion *unifyFun*, dargestellt in Abb.15, wobei Variablen besonders behandelt werden, da in *unifyFun*2. ein *occur-check* durchgeführt wird.

Der Algorithmus ist möglichst allgemein formuliert, da die Behandlung von Funktionen davon abhängt, wie deren Verwendung eingeschränkt wird. Für die Repräsentation von Zeitintervallen reicht die oben beschriebene Nachfolgerfunktion *succ* aus. Die Nachfolgerfunktion wird als ein Term $succ(fs)$[37] spezifiziert, wobei fs ein atomarer Typ ist. Es wird also unter Punkt 1. durch die Anwendung der (n-te) Nachfolger bestimmt und der die Funktion repräsentierende Term durch diesen ersetzt, falls kein Fehler auftritt. Die Anwendung beinhaltet, daß Überläufe der Werte in anderen Teilen der Struktur auftreten können. Dies betrifft Teile der Struktur, die bereits unifiziert wurden, d.h. der an dieser Stelle der Verarbeitung berechnete Teilunifikator ist von der Auswertung der Funktion betroffen.

[37] bzw. $succ(fs, n)$ mit $n \in \mathcal{N}$

unifyMod(f_1, f_2), geg: Kontext $\mathcal{K}$ mit Variablenbindung Θ

1. Fallunterscheidung:
   (a) falls f_1 oder f_2 Funktion:
   return(*unifyFun*(f_1, f_2))
   (b) falls f_1 und f_2 Atom:
   return($f_1 \sqcup f_1$, ()) falls $f_1 \sqcup f_1$ definiert ist,
   sonst: return($[\top]$, (f_1, f_2))
   (c) falls f_1 oder f_2 Variable:
   i. (d, R) := *unifyMod*($\Theta(f_1)$, $\Theta(f_2)$)
   ii. $\boxed{\text{x}}$:= *push*((d, f_1, f_2), Θ, $\mathcal{K}$);
   iii. return($[\boxed{\text{x}}d]$, R)

2. falls ***sub*** := *subType*(f_1, f_2) definiert ist:
   initialisiere d:***sub***,
   sonst: return($[\top]$, (f_1, f_2))

3. wähle Subknoten q_1, q_2 :

$$\begin{aligned} A &:= selectA(f_1, f_2) \\ q_i &:= \begin{cases} \delta(A, f_i) : \boldsymbol{t} & \text{falls } \delta(A, f_i) : \boldsymbol{t} \text{ definiert ist} \\ [\top] & \text{sonst} \end{cases} \end{aligned}$$

   berechne (d_q, R) := *unifyMod*(q_1, q_2) und setze $\delta(A, d) := d_q$

4. falls R = ()

$$\begin{aligned} f_1 &:= f_1 - [A\colon [\delta(A, f_1)]] \\ f_2 &:= f_2 - [A\colon [\delta(A, f_2)]] \end{aligned}$$

   sonst: return(d, R)

5. falls $f1 \neq [\top]$ oder $f2 \neq [\top]$ weiter bei 3
   sonst: return(d, ())

Abbildung 16: Algorithmus *unifyMod*

Ist der Wert an der Stelle des Teilunifikators, an der ein Überlauf auftritt, bereits definiert, schlägt die Unifikation fehl, da die Auswertung der Funktion durch die während der Unifikation hinzugekommene Information zu einer inkonsistenten Struktur führt.

Betrifft der Überlauf eine ungebundene Variable v vom Typ t, dann wird diese an einen Term $\mathsf{t}{:}succ(\mathsf{t}{:}v')$ gebunden, wobei v' eine neue, noch ungebundene Variable ist. Diese "Nichtlokalität" der Funktionsauswertung macht eine globale Zugriffsmöglichkeit auf die Struktur des Teilunifikators notwendig, was technisch leicht zu realisieren ist, indem in der Klasse der Unfikationsoperation ein Zeiger auf den Wurzelknoten des Teilunifikators gespeichert wird.

Der Unifikator von Atomen und der Disjunktion von Atomen wird durch Gleichheit bzw. Schnittbildung berechnet, d.h. die Unifikation von Disjunktionen wird unter Ausnutzung der Distributivität ausmultipliziert. Da nur Atome und TID innerhalb von Disjunktionen zugelassen werden, ist dieses Vorgehen unproblematisch.

Bindungen von Variablen werden in Θ gespeichert und in der Zeile 1.(c)i. unifiziert. Durch die Funktion *push* (Abbildung 16 1.(c)ii.) werden neu erzeugte Variablenbindungen zusammen mit dem aktuellen Verarbeitungszustand im Kontext gespeichert, so daß späteres *backtracking* möglich ist. Der evtl. unverbrauchte *tag*, den *push* als Wert zurückgibt, wird in die neue Struktur eingetragen.

Unter Punkt 2. in Abb.16 wird mittels der Funktion *subType* der jeweilige gemeinsame Subtyp der zu identifizierenden Knoten inferiert und mit dem neuen Knoten assoziiert. Falls kein solcher Typ existiert, unifizieren die Knoten nicht, so daß der unifizierte Teil sowie die verbliebenen Reste als Werte zurückgegeben werden. Konnte ein Subknoten erfolgreich instanziiert werden, wird in einer Schleife über die Punkte 3.-5. die Struktur rekursiv aufgebaut. Erfolgreich durchwanderte Teilstrukturen der Eingabe werden dabei gelöscht. Wenn $([\top], (f_1, f_2))$ der Wert der Funktion und $f_{1,2} \neq [\top]$ ist, dann ist dies äquivalent zu *fail*, d.h. die Unifikation ist fehlgeschlagen.

2.3.3 Unifikation für JFS vom Typ TID

Zunächst werden die JFS unter den Attributen BEG und END unifiziert und damit ein Kontext $\mathcal{K}$ und Teilunifikatoren und Reste berechnet. Da die Unifikation der Teilstrukturen entlang der durch Spec gegebenen Reihenfolge verläuft, genügt die Relation $\mathsf{before} \subset \mathsf{Rest}$[38], um zu entscheiden, ob für Reste $R_{BEG,END}$ gilt; $(R_{BEG}, R_{END}) \in \mathsf{dur}$[39].

Gilt für alle $f \in \mathsf{IdFS}$ folgende Bedingung:

[38] Mit $\{()\} \in \mathsf{before}$.

[39] bzw. dur^{-1}; $(f_1, f_2) \in \mathsf{dur}$ gdw. $(f_2, f_1) \in \mathsf{dur}^{-1}$.

(1) wenn $(d, (r_1, r_2)) = unifyMod(\delta(\text{BEG}, f), \delta(\text{END}, f))$,
dann $(r_1, r_2) \in \textsf{before}$

d.h. kein "Endpunkt" darf vor einem "Anfangspunkt" liegen[40], dann folgt aus der Transitivität von before ($\textsf{before}^{-1}$):

(2) für alle Paare von Resten B, $E \in \textsf{Rest}$ und $f_{1,2}$ in IdFs mit:

$$(d_B, B) = unifyMod(\delta(\text{BEG}, f_1), \delta(\text{BEG}, f_2))$$
$$(d_E, E) = unifyMod(\delta(\text{END}, f_1), \delta(\text{END}, f_2))$$

ist $(B, E) \in \textsf{dur}$ ($\textsf{dur}^{-1}$)
gdw. $B \in \textsf{before}$ ($\textsf{before}^1$) und $E^{-1} \in \textsf{before}$ ($\textsf{before}^1$) ist.

Besteht das Resultat aus (einem) Teilunifikator(en) und Resten, bleibt zu überprüfen, ob die nicht unifizierten Reste in dem Inklusionsverhältnis dur stehen. Ist dies der Fall, dann ist die Unifikation der inkludierten Teilstruktur mit dem Teilunifikator Ergebnis der gesamten Unifikation. Für einen direkten Vergleich der Wochentage Dow muß der Bezugsrahmen in Form der Sequenz der in Frage kommenden Tage angegeben sein oder man macht die heuristische Annahme, daß die Woche mit Montag beginnt und mit Sonntag endet.

Die Reihenfolge, in der die Teilstrukturen unifiziert werden, erzeugt u.U. unterschiedliche Variablenbindungen. Da anders als bei der Standardunifikation auch konsistente Teilunifikatoren relevant sind und damit verschiedene *fails* in Betracht gezogen werden, erfordert dieser Umstand *backtracking*. *backtracking* wird im wesentlichen durch die Funktion *push* und den Kontext $\mathcal{K}$ realisiert. Die Notwendigkeit von *backtracking* wird im folgenden Beispiel (Abb.17) deutlich, wenn man das Intervall (C, D) einfach in den nächsten Monat verschiebt.

Ausgehend von der im Kontext $\mathcal{K}$ des Algorithmus (Abbildung 16) enthaltenen Information, kann eine Lösungsmenge statt einer einzelnen Lösung durch *backtracking* berechnet werden. Hierzu muß allein der Stapel der gespeicherten Variablenbindungen und Verarbeitungszustände in Form von Teilgraphen abgearbeitet werden, bis ein neuer Teilunifikator und Reste berechnet sind.

Ein nicht leerer Kontext mit den enthaltenen Variablenbindungen Θ kann zusammen mit dem Typ des Teilunifikators als hypothetische Voraussetzung für eine Beurteilung des Verhältnisses zweier Strukturen betrachtet werden.

Unifikation für JFS vom Typ TID **terminiert**, gdw. *unifyMod* für die Teilstrukturen unter den Attributen BEG und END terminiert[41]. Die **Korrektheit** folgt direkt aus der durch *unifyMod* geleisteten Berechnung der Teilunifikatoren und Reste und den Bedingungen (1) und (2)[42], die für den Typ TID gelten. Der Algorithmus ist für JFS vom Typ TID **vollständig**, da *unifyMod* vollständig ist und durch *backtracking* alle zuläßigen Variablenbindungen berechnet werden können.

[40]Wie ja durch das Modell vorgegeben.
[41]Beweis siehe Anhang.
[42]S.S.32.

2.3.4 Erweiterung des Unifikators von JTFS

Die Spezifikationen von *unifyMod* werden von dem Standardunifikator von JTFS nicht erfüllt, da die Unifikationsoperation von JTFS nur vollständige Unifikatoren berechnet oder fehlschlägt, wenn die Eingabe inkonsistent ist. Um konsistente Teilunifikatoren und inkonsistente Restmengen berechnen zu können, muß Unificator erweitert werden. Die Klasse TIDU stellt eine Spezialisierung des Unifikators für JFS der Typen der Modellierung der temporalen Ausdrücke dar und implementiert die Funktionen *unifyMod* und *unifyFun* als Methoden.

Für die erste prototypische Version wurde auf eine Modellierung der Variablenbindung durch einen Kontext verzichtet. Stattdessen wird auf kopierten Strukturen operiert und die durch Java bereitgestellte Bindung von Variablen verwendet. Hierdurch ist kein Backtracking möglich, so daß die implementierte Methode nicht vollständig ist, da durch die feste Vorgabe der Reihenfolge der Unifikation der Substrukturen der Teilunifikator bestimmt ist. Da bei der Unifikation von TID *de facto* nur die Attribute BEG und END variiert werden können, läßt sich die Methode von TIDU leicht erweitern, so daß *beide* Teilunifikatoren berechnet werden können.

TIDU wird im wesentlichen durch Iteratoren gesteuert. Der Zugriff auf Teilstrukturen wird durch die Iteratoren realisiert, die von den Objekten der Objekthierarchie erzeugt werden. Die Iteratoren dienen auch dazu, in Fallunterscheidungen Informationen über die Objekte bereitzustellen, die sie erzeugt haben. Damit der Zugriff der Granularität der Attribute folgt, müssen Iteratoren die Relation Spec implementieren und die Attribute sortieren. Das Interface Iterator und die implementierende Klasse IteratorImpl werden um entsprechende Methoden erweitert. JFS erzeugen im einfachen Fall die Standarditeratoren der JTFS und für spezielle Objekte wie zum Beispiel Listenobjekte bei Disjunktionen oder Instanzen der Klasse SuccAtomicImpl spezialisierte Iteratoren.

Die Funktion *subType* wird durch Methode glb() der MetaJFS für die Standardtypen realisiert. Für die korrekte Behandlung von Listen- und Nachfolgertypen wird die Methode innerhalb einer Spezialisierung von MetaJFS überladen. Die Initialisierung von Subtypen, die durch glb() inferiert werden, ist durch die Methode instance() in MetaJFS implementiert. Um die Funktion *unifyFun* durch die Methode unifyFun() modellieren zu können, wurde eine Subklasse von AtomicJFSImpl eingeführt, die die notwendigen Informationen enthält. Werte, die lokal berechnet werden können, werden von der Methode direkt in den Unifikator eingetragen. Sonst wird ein Objekt der Klasse SuccAtomicImpl von dem atomaren Objekt erzeugt und die Korrektheit der Struktur nach Ablauf der Unifikation durch einen Visitor überprüft.

TIDU enthält zwei Felder, in denen die berechneten Reste einer Unifikation gespeichert werden, und ein Feld für den Unifikator der Eingabe. Reste werden berechnet, indem die Teile der Eingabestrukturen, die unifiziert sind, gelöscht werden. Die Reste einer Unifikation sind die Eingabestrukturen ohne die unifizierten Teilobjekte. Das Löschen von Teilstrukturen wird durch das Überschreiben mit einer leeren Struktur vom Typ JFS realisiert. Die Unifikationsmethode liefert keinen Wert, sondern speichert das Resultat in den Feldern für die Reste sowie für den Unifikator und setzt ein Flag, das den Erfolg der Unifikation anzeigt. TIDU enthält Zugriffsmethoden, mit denen der Status und die Werte einer Unifikationsoperation abgefragt werden können.

2.3.5 Beispiele

Anhand des ersten Beispiels soll der Zusammenhang zwischen Bedingung (1)[43] und der Unifikation der jeweiligen Referenzrahmen als Voraussetzung für einen korrekten Vergleich der Reststrukturen illustriert werden. Das zweite Beispiel demonstriert die Repräsentation eines Dialogs.

Beispiel (1): Unifikation mit Resten

Sei die Situation wie in Abb.17 gegeben, dann sind folgende (teilweise) unkorrekte Intervallbeschreibungen aus Abbildungen 18 und 19 denkbar.

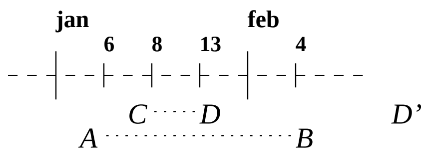

Abbildung 17: Situation: $((C, D), (A, B)) \in$ dur

Der Versuch die Struktur id_{A-B} in Abb.18 mit id_{C-D}, bzw. id'_{C-D} in Abb.19 zu unifizieren, schlägt fehl, da jeweils für die aus der Unifikation der END-Werte verbliebenen Reste r_D, r_B:

$$\left(\begin{bmatrix} \text{DayDate} & \\ \text{DAY:} & \begin{bmatrix} \text{DomDay} & \\ \text{DOM:} & [\,13\,] \end{bmatrix} \end{bmatrix}_D , \begin{bmatrix} \text{DayDate} & \\ \text{DAY:} & \begin{bmatrix} \text{DomDay} & \\ \text{DOM:} & [\,4\,] \end{bmatrix} \end{bmatrix}_B \right) \notin \textsf{before}$$

gilt. Es sind $(r_D, r_B) \notin$ before, da die unifizierten Struturen unterspezifizeirt sind und das Attribut MOY nicht enthalten.

Das gilt auch für die Unifikation der korrigierten Struktur id'_{A-B} in Abb.18 mit id_{C-D}, da zunächst die MONTH-Werte unifiziert werden und sich daraus wieder der Vergleich 13 before 4 ergibt; allerdings mit einer etwas anderen Bedeutung, da 13 wegen der vorangegangenen Unifikation der Monatswerte korrekterweise im nächsten Monat interpretiert wird $(= D')$.

[43] S.S.32.

Zu dem gewünschten Ergebnis, nämlich der Struktur id'_{C-D} führt erst die Unifikation von id'_{A-B} mit id'_{C-D}, da gilt:

$$\begin{bmatrix} \mathsf{DayMonthDate} & \\ \text{M.:} & \begin{bmatrix} \mathsf{MoyMonth} & \\ \text{MOY:} & \begin{bmatrix} \boxed{1'}\mathsf{moy} \end{bmatrix} \end{bmatrix} \end{bmatrix}_D , \begin{bmatrix} \mathsf{DayMonthDate} & \\ \text{M.:} & \begin{bmatrix} \mathsf{MoyMonth} & \\ \text{MOY:} & \begin{bmatrix} \mathsf{Moy} \\ succ(\boxed{1'}) \end{bmatrix} \end{bmatrix} \end{bmatrix}_B \in \mathsf{\mathbf{before}}$$

Hierbei wird deutlich, daß zuerst die aus der Unifikation der BEG-Werte berechnete Bindung Θ angewendet werden muß. Nach Anwendung der Bindung $\Theta = \boxed{1}_{A-B}/\boxed{1}_{C-D}, \ldots$ scheitert die Unifikation von $\boxed{1}$ mit $succ(\boxed{1})$ durch den in der Funktion *unifyFun* durchgeführten *occur-check* aus Abb.15 Punkt 2.

$$id_{A-B} = \begin{bmatrix} \mathsf{TID} & \\ \text{BEG:} & \begin{bmatrix} \mathsf{DayDate} & \\ \text{DAY:} & \begin{bmatrix} \mathsf{DomDay} & \\ \text{DOM:} & \begin{bmatrix} 6 \end{bmatrix} \end{bmatrix} \end{bmatrix} \\ \text{END:} & \begin{bmatrix} \mathsf{DayDate} & \\ \text{DAY:} & \begin{bmatrix} \mathsf{DomDay} & \\ \text{DOM:} & \begin{bmatrix} 4 \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{bmatrix} \quad \text{(unkorrekt, da (1)(S. 32) verletzt)}$$

$$id'_{A-B} = \begin{bmatrix} \mathsf{TID} & \\ \text{BEG:} & \begin{bmatrix} \mathsf{DayMonthDate} & \\ \text{MONTH:} & \begin{bmatrix} \mathsf{MoyMonth} & \\ \text{MOY:} & \begin{bmatrix} \boxed{1}\mathsf{Moy} \end{bmatrix} \end{bmatrix} \\ \text{DAY:} & \begin{bmatrix} \mathsf{DomDay} & \\ \text{DOM:} & \begin{bmatrix} 6 \end{bmatrix} \end{bmatrix} \end{bmatrix} \\ \text{END:} & \begin{bmatrix} \mathsf{DayMonthDate} & \\ \text{MONTH:} & \begin{bmatrix} \mathsf{MoyMonth} & \\ \text{MOY:} & \begin{bmatrix} \mathsf{Moy} \\ succ(\boxed{1}) \end{bmatrix} \end{bmatrix} \\ \text{DAY:} & \begin{bmatrix} \mathsf{DomDay} & \\ \text{DOM:} & \begin{bmatrix} 4 \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{bmatrix} \quad \text{(korrekt)}$$

Abbildung 18: Beispielstrukturen für das Intervall $A - B$

$$id_{C-D} = \begin{bmatrix} \mathsf{TID} & \\ \text{BEG:} & \begin{bmatrix} \mathsf{DayDate} & \\ \text{DAY:} & \begin{bmatrix} \mathsf{DomDay} & \\ \text{DOM:} & [\,8\,] \end{bmatrix} \end{bmatrix} \\ \text{END:} & \begin{bmatrix} \mathsf{DayDate} & \\ \text{DAY:} & \begin{bmatrix} \mathsf{DomDay} & \\ \text{DOM:} & [\,13\,] \end{bmatrix} \end{bmatrix} \end{bmatrix} \quad \text{((zu) unterspezifiziert)}$$

$$id'_{C-D} = \begin{bmatrix} \mathsf{TID} & \\ \text{BEG:} & \begin{bmatrix} \mathsf{DayMonthDate} & \\ \text{MONTH:} & \begin{bmatrix} \mathsf{MoyMonth} & \\ \text{MOY:} & [\,\boxed{1}\mathsf{Moy}\,] \end{bmatrix} \\ \text{DAY:} & \begin{bmatrix} \mathsf{DomDay} & \\ \text{DOM:} & [\,8\,] \end{bmatrix} \end{bmatrix} \\ \text{END:} & \begin{bmatrix} \mathsf{DayMonthDate} & \\ \text{MONTH:} & \begin{bmatrix} \mathsf{MoyMonth} & \\ \text{MOY:} & [\,\boxed{1}\mathsf{Moy}\,] \end{bmatrix} \\ \text{DAY:} & \begin{bmatrix} \mathsf{DomDay} & \\ \text{DOM:} & [\,13\,] \end{bmatrix} \end{bmatrix} \end{bmatrix} \quad \text{(hinreichend spezifiziert)}$$

Abbildung 19: Beispielstrukturen für das Intervall $C - D$

Beispiel (2): Unifikation von Objekthierarchien

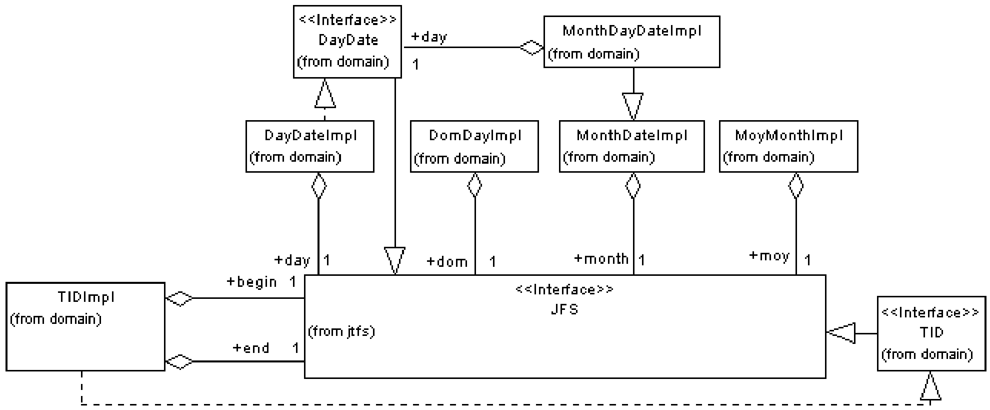

Abbildung 20: Klassenstruktur einer TID

Das UML-Diagramm in Abb.20 stellt eine Aggregation dar, die Objekthierarchien vom Typ TID spezifiziert. Die Klassen implementieren Subinterfaces von JFS, die, um das Diagramm überschaubar zu halten, nur am Beispiel von DayDate und dargestellt sind.

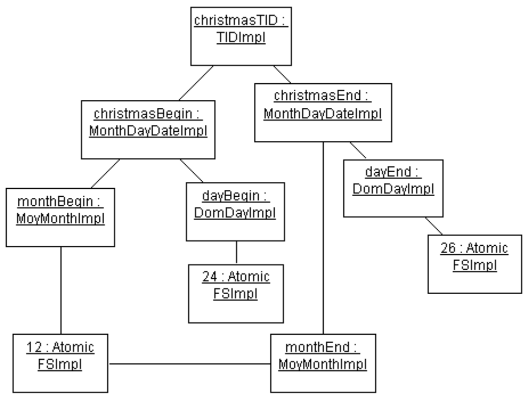

Abbildung 21: Objektmodell von "christmas"

Eine Instanz der Klassenstruktur ist in Abb.21 als Objektmodell dargestellt, wobei die Verbindungen[44] zwischen den Objekten durch die Aggregatbeziehungen der Klassenstruktur vorgegeben sind. Die Assoziation zwischen den Objekten monthBegin und 12 sowie zwischen monthEnd und 12 repräsentiert eine Koreferenz.

[44]Engl.: *Links*

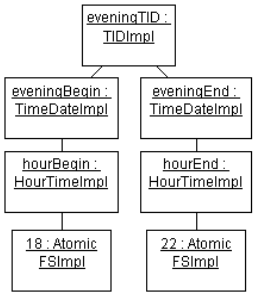

Abbildung 22: Objektmodell von "evening"

Abb.22 stellt ein Objektmodell dar, das die Information "evening" repräsentiert. Die Objektmodelle der Abb.21 und Abb.22 sind kompatibel, so daß sie unifizieren. Das UML-Diagramm in Abb.23 stellt das Objektmodell des Unifikators dar.

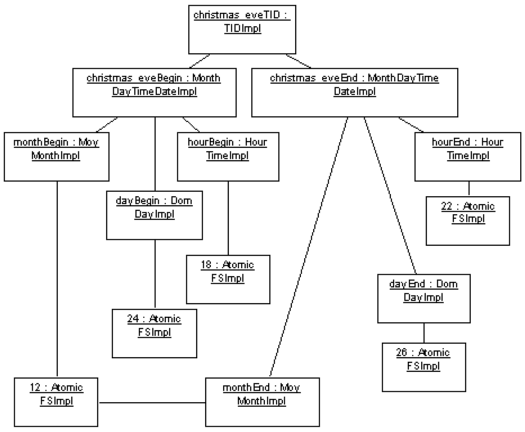

Abbildung 23: Objektmodell des Unifikators von "evening" und "christmas"

3 Temporal Expression Evaluator (TEE)

Der folgende Abschnitt beschreibt die Funktion und Struktur der Komponente TEE, die basierend auf der Modellierung der Information temporaler Ausdrücke domänenspezifische Inferenzoperationen realisiert. TEE implementiert eine Methode zur Konstruktion von Objekthierarchien des Typs TID, die in einem Speicher abgelegt werden. Auf diesen Strukturen werden Inferenzoperationen wie Vergleiche und Konsistenzüberprüfung durchgeführt.

3.1 Konstruktion von TID

Um Inferenzoperationen auf temporalen Ausdrücken durchführen zu können, müssen von diesen Ausdrücken Instanzen von TID konstruiert werden. Die Konstruktionsoperation wird als Übersetzung von $\mathcal{L} \subset \mathcal{L}(ZG)$ nach IdFS realisiert, wobei unterschiedliche Informationsquellen als Kontext verwendet werden. Die Übersetzung ist nur auf einer Teilmenge von $\mathcal{L}(ZG)$ definiert, da Ausdrücke in $\mathcal{L}(ZG)$ existieren, die nicht sinnvoll als Zeitintervall modelliert werden können. Bei der Übersetzung wird die Konsistenz des gegebenen ZG-Ausdrucks überprüft. Im folgenden Abschnitt werden zunächst mögliche Quellen der Inkonsistenz vorgestellt, bevor das Verfahren der Übersetzung und gleichzeitiger Konsistenzprüfung informell erläutert wird. Es folgt eine Auflistung der mit einzelnen Kategorien der ZG assoziierten Übersetzungsoperationen.

3.1.1 Inkonsistenzen in $\mathcal{L}$

Eingangs wurde erwähnt, daß die ZG unterschiedliche Ausdrücke mit gleichem Gehalt zuläßt. Untersuchte Dialoge weisen neben häufig unvollständig spezifizierten Zeitausdrücken, die eventuell Referenzresolution auf der Basis von kontextuellem Wissen notwendig machen, auch Redundanzen auf. Neben leeren Referenzen als Ursache für Inkonsistenz führen vom Typ her redundante Ausdrücke durch Inkompatibilität der Werte eine weitere Möglichkeit für inkonsistente Terminbeschreibungen ein, die bei einer Übersetzung berücksichtigt und eventuell korrigiert werden müssen.

Unter leeren Referenzen sind all die Ausdrücke zu verstehen, die in sich konsistent und syntaktisch korrekt sind, für die jedoch keine kalendarische Entsprechung existiert. Dies ist z.B. der Fall bei:

> "Der 31. Februar"

Für diese Klasse von Ausdrücken lassen sich Instanzen von TID konstruieren, die nicht auf ein Element des Kalendermodells abzubilden sind. Dies wird nach der Übersetzung überprüft, so daß keine unerfüllbaren Ausdrücke generiert werden.

Redundante Ausdrücke sind all diejenigen, in denen sowohl ein Teilausdruck der Kategorie `POD` als auch `TOD` vorkommt, auch wenn der `TOD`-Ausdruck schon eindeutig ist, wie z.B.:

"morgens so gegen 18:00 Uhr"

Die Übersetzung weist inkonsistente Ausdrücke dieser Art zurück, da TID durch Unifikation konstruiert werden. Es besteht jedoch die Möglichkeit, für konsistente Teilmengen Instanzen von TID zu generieren und damit eine Grundlage für etwaige Reparaturen des Ausdrucks zu schaffen.

3.1.2 Übersetzung

Das hier vorgestellte Schema einer Abbildung von $\mathcal{L}$ auf IdFS besteht im wesentlichen darin, schrittweise die im ZG-Ausdruck enthaltenen Teilausdrücke basierend auf unterschiedlichen Wissensquellen und Vorgehensweisen zu interpretieren. Resultat ist jeweils eine Merkmalstruktur, die den Teilausdruck repräsentiert. Die so konstruierten JFS werden sukzessive mit einer zunächst unspezifizierten TID id unifiziert. Schlägt die Unifikation fehl, ist der zu interpretierende ZG-Ausdruck inkonsistent. Sonst ist id das Resultat der Übersetzung und enthält alle aus der Eingabe extrahierbaren Informationen.

Die Übersetzung wird im wesentlichen durch die Interpretationsfunktion I:$\mathcal{L} \longrightarrow$ IdFS geleistet, die jedem hinreichend einfachen Ausdruck $t \in \mathcal{L}$ eine Intervallbeschreibung zuordnet oder sonst t in Teilausdrücke zerlegt und diese in eine Menge von Intervallbeschreibungen überführt und unifiziert. Ist t selbst ein Ausdruck, der ein Intervall beschreibt, muß bei seiner Dekomposition darauf geachtet werden, daß die interpretierten Teilausdrücke jeweils nur dem END-Datum bzw. BEG-Datum zugeordnet werden. Das bedeutet bei tieferen Verschachtelungen, daß die maximal- bzw. minimal-Werte ausgewählt werden müssen.

Teilausdrücke t' und t'' von t beziehen sich auf das gleiche temporale Objekt wie t, so daß die JFS $id' = \mathrm{I}(t')$ und $id'' = \mathrm{I}(t'')$ unifizieren. Redundante und konsistente Zeitausdrücke werden durch TIDU korrekt auf das enthaltene und damit "spezifischere" Intervall abgebildet. Es werden unscharfe Ausdrücke, wie z.B. $afternoon$, auch als Intervalle interpretiert und als Resultat wird das eingeschränkte Intervall berechnet, wenn jene zusammen mit spezifischeren Angaben in einem redundanten Zeitausdruck vorkommen. Dadurch wird eine bezüglich der Eingabe maximal spezifizierte Intervallbeschreibung berechnet.

3.1.3 Wissensquellen

Bei der Abbildung von Ausdrücken der ZG auf Intervallbeschreibungen werden unterschiedliche Wissensquellen genutzt. Dabei wird die Strategie verfolgt, möglichst wenig Inferenzen während der Übersetzung zu leisten und eher sehr unterspezifizierte TID zu generieren.

- Eine Tabelle implementiert eine heuristische Abbildung der `POD`-[45] und `POY`-Terme und auf Zeitintervalle.
- Der Kalender fungiert als das der Interpretation von TID zugrundeliegende Modell. Er ordnet Feiertagen Datumsangaben zu und assoziiert Monate im Jahr mit Woche im Jahr usw.
- Der Dialogkontext liefert einen Zeitstempel für die aktuelle Äußerung und ermöglicht den Zugriff auf bisher im Diskurs interpretierte Zeitausdrücke.

3.1.4 Die Elemente von ZG und ihre Übersetzung

Einfache Ausdrücke

Oft ergibt erst eine Kombination[46] von "einfachen Ausdrücken" die Information, die einen Wert vollständig definiert. So ist z.B. die Angabe von Jahreswoche (`woy`) und Wochentag (`dow`) äquivalent zur Angabe von Monat (`moy`) und Tag im Monat (`dom`). Zur Interpretation solcher Ausdrücke wird gezielt alle nötige Teilinformation aus der Eingabe extrahiert, um zum gegebenen Verarbeitungsschritt den Wert auf der Grundlage des Kalenders berechnen zu können.

Die einfachen Ausdrücke der ZG sind unterteilt in die Kategorien `POINT` und `POINTLIKE`. Unter `POINTLIKE` werden verschiedene Jahres- und Tageszeiten sowie Monats- und Wochenangaben zusammengefaßt, die alle eine zeitpunktartige Auffassung zulassen, jedoch auf Zeitspannen referieren. `POINT` ist die Kategorie der Tageszeitangaben, die aber erst in Kombination mit `am` bzw. `pm` – Ausdrücken der `POD`-Kategorie (`POINTLIKE`) – eindeutig interpretierbar ist.

Unter `POINTLIKE` fallen folgende Kategorien mit den zugehörigen Übersetzungsoperationen bzw. Inferenzschritten:

`POD` (`morning, beforenoon, ...`) werden auf der Basis einer Heuristik direkt in Intervalle abgebildet, die in Form einer Tabelle, die Anfangs- und Endpunkte angibt, implementiert ist. `am` und `pm` werden zur Desambiguierung von Uhrzeitangaben verwendet sowie alle `POD`s, die als Zeitspannen eindeutig unter `am` bzw. `pm` fallen. Wie das folgende Beispiel in Abb.24 demonstriert, geschieht dies durch Unifikation der generierten Teilausdrücke.

`DOW` (`mon | ... sun| workday | weekend`) definieren in Kombination mit der Woche im Jahr (`woy`) eindeutig die Attribute MONTH und DAY. Sonst dienen sie nur der Überprüfung der Konsistenz des Zeitausdrucks. Die deiktischen Ausdrücke wie `today` etc. unter den `DOW`s sind nur bei gegebenem aktuellem Äußerungszeitpunkt zu interpretieren.

[45] `evening, afternoon, ...`
[46] Evtl. verschiedener ZG-Kategorien.

POY (spring | ...) werden ähnlich der PODs direkt auf Intervalle abgebildet.

WOY setzt eine Zeitspanne durch das Monat-im-Jahr (Moy) und Tag-im Monat-Attribut (Dom).

DOM, MOY, YEAR, WOM definieren das Attribut des jeweiligen Typs.

HOLIDAY definieren im Zusammenhang mit der Jahreszahl mindestens die Attribute vom Typ Moy und Dom, in Ausnahmefällen auch die Uhrzeit (neujahr).

Intervalle

Die LIMIT- und BOUNDARIES-Ausdrücke lassen sich direkt als offene und geschlossene Intervalle darstellen, wobei ein BOUNDARIES-Ausdruck nur mit der Semantik von point_between dargestellt wird, da from_to explizit die ganze Dauer bedeutet. Hierfür müßte der Formalismus insofern erweitert werden, so daß Spezialisierung als Unifikation partiell unzulässig wäre – etwa durch Einführung eines Attributs und der Anpassung des Unifikationsalgorithmus.

Wenn ein kalendarischer Bezugspunkt gegeben ist, sind Anfangs- und Endpunkt von DURATIONs (Zeitspannen) direkt berechenbar und können so auf IdFS abgebildet werden. Sonst wird die durch einen Ausdruck der Kategorie DURATION bezeichnete Distanz mittels der Nachfolgerfunktion *succ* und Variablen dargestellt. Bei späterer Instanziierung der Variablen kann der Wert dann berechnet werden[47].

Ausdrücke mit Referenzzeitpunkt

Bei gegebenem kalendarischem *Anker* werden die durch verschiedene COUNTER beschriebenen Zählanweisung durchgeführt und so die Eckpunkte des Intervalls inferiert.

Ausdrücke mit Abzählung

Diese Ausdrücke teilen sich grob in die Klassen Zeitspannen, wie z.B. counted(2, week)[48] und kalendarisch beschreibbare Daten, die durch die Bestimmung eines Referenzzeitpunktes und der Berechnung auf unterschiedliche Weise gegebener Abzählschritte inferiert werden.

[47] Dabei muß auf Wertüberläufe geachtet werden s.S.24.

[48] " ... die zweite Woche ..."

Modifikation und "Fuzzyfikation"

Die Interpretation der Modifikatoren `MOD` (`early | late ...`) führt auf der Basis einer Heuristik zu einer Einschränkung bzw. Einengung des modifizierten Intervalls, die des `FUZZIFIER`s `about` zu einer Erweiterung.

Quantoren

Allquantifizierte Ausdrücke können nicht interpretiert werden, wenn sie Frequenzen beschreiben, wie z.B.:

"jeder (dritte) Montag im Monat"

Der Formalismus spezifiziert keine Relationen zwischen TID, die über den Vergleich der temporalen Information hinausgehen. Um *jeder (dritte) Montag im Monat* zu repräsentieren, müßten Strukturen definierbar sein, die aus Instanzen von TID bestehen. Allerdings kann zumindest vordergründig eine Lesart von Ausdrücken der Form `all([], [moy:jan])` (*Immer im Januar*) als der gesamte Zeitraum repräsentiert werden. Intuitiv ist dann aber die Semantik der Unifikation als Spezifizierung nicht mehr korrekt, so daß von einer Repräsentation des Allquantors Abstand genommen wird.

Dahingegen läßt sich der Existenzquantor leicht darstellen, da seine Semantik direkt der Unterspezifiziertheit einer Strutur entspricht. So wird z.B. `any([dow:mon], [moy:may])` in eine Struktur mit variablem DOM-Wert übersetzt.

ao - Listen

Auswahllisten werden als Listen von TID-Strukturen dargestellt; die Erweiterung des Basisalgorithmus zur Verarbeitung von Listen ist kanonisch.

3.1.5 Ein Beispiel für die Konstruktion durch Unifikation

S: *das Treffen beginnt um 3:30, Montag Nachmittag*

ZG: `[tod:3:30, dow:mon, pod:afternoon]`

$\mathcal{T}(\texttt{[tod:3:30]}) := id_1$:

$$\text{BEG:}\left[\begin{array}{ll} \multicolumn{2}{l}{\boxed{1}\,\mathsf{TimeDayMonthYearDate}} \\ \text{YEAR:} & \left[\mathsf{Year}\right] \\ \text{MONTH:} & \left[\mathsf{Month}\right] \\ \text{DAY:} & \left[\mathsf{Day}\right] \\ \text{TIME:} & \left[\begin{array}{l} \mathsf{MinuteHourTime} \\ \text{HOUR:}\{\left[15\right], \left[3\right]\} \\ \text{MINUTE:} \left[30\right] \end{array}\right] \end{array}\right], \text{ END: }\left[\boxed{1}\,\mathsf{TimeDayMonthYearDate}\right]$$

$\mathcal{T}(\texttt{[dow:mon]}) := id_2$

$$\text{BEG:}\left[\begin{array}{ll} \multicolumn{2}{l}{\boxed{1}\,\mathsf{DayMonthYearDate}} \\ \multicolumn{2}{l}{\ldots} \\ \text{DAY:} & \left[\begin{array}{l} \mathsf{DowDay} \\ \text{DOW:} \left[\mathsf{mon}\right] \end{array}\right] \end{array}\right], \text{ END: }\left[\boxed{1}\,\mathsf{DayMonthYearDate}\right]$$

$\mathcal{T}(\texttt{[pod:afternoon]}) := id_3$

$$\text{BEG:}\left[\begin{array}{ll} \multicolumn{2}{l}{\mathsf{TimeDayMonthYearDate}} \\ \text{YEAR:} & \left[\boxed{1}\,\mathsf{Year}\right] \\ \text{MONTH:} & \left[\boxed{2}\,\mathsf{Month}\right] \\ \text{DAY:} & \left[\boxed{3}\,\mathsf{Day}\right] \\ \text{TIME:} & \left[\begin{array}{l} \mathsf{HourTime} \\ \text{HOUR:} \left[13\right] \end{array}\right] \end{array}\right], \text{END:}\left[\begin{array}{ll} \multicolumn{2}{l}{\mathsf{TimeDayMonthYearDate}} \\ \text{YEAR:} & \left[\boxed{1}\,\mathsf{Year}\right] \\ \text{MONTH:} & \left[\boxed{2}\,\mathsf{Month}\right] \\ \text{DAY:} & \left[\boxed{3}\,\mathsf{Day}\right] \\ \text{TIME:} & \left[\begin{array}{l} \mathsf{HourTime} \\ \text{HOUR:} \left[18\right] \end{array}\right] \end{array}\right]$$

$id_1 \sqcup id_2 \sqcup id_3 := id$:

$$\text{BEG:}\left[\begin{array}{ll} \multicolumn{2}{l}{\boxed{1}\,\mathsf{TimeDayMonthYearDate}} \\ \multicolumn{2}{l}{\ldots} \\ \text{DAY:} & \left[\begin{array}{l} \mathsf{DowDay} \\ \text{DOW:} \left[\mathsf{mon}\right] \end{array}\right] \\ \text{TIME:} & \left[\begin{array}{l} \mathsf{HourMinuteTime} \\ \text{HOUR:} \left[15\right] \\ \text{MINUTE:} \left[30\right] \end{array}\right] \end{array}\right], \text{END: }\left[\boxed{1}\,\mathsf{TimeDayMonthYearDate}\right]$$

Abbildung 24: Konstruktion einer TID-Struktur

Das Beispiel in Abb.24 illustriert die Rolle der Unifikation von Teilausdrücken bei der Konstruktion von TID-Strukturen. Wie oben beschrieben zerlegt das Verfahren die Eingabe in Form eines Ausdrucks der ZG in Teilausdrücke, die direkt übersetzt werden können. Unfikation leistet dann die Komposition der Teilübersetzungen. Durch die speziell für Intervalle definierte Unfikation werden unterspezifizierte Teilstrukturen weiter spezifiziert, wie hier anhand der Repräsentation von "3:30" und "Nachmittag" zu sehen ist.

3.2 Inferenzen

Die hier zu leistenden Inferenzen haben zum Ziel, Relationen zwischen Repräsentationen zeitlicher Information, wie sie im Laufe eines Dialogs generiert werden, zu bestimmen. Die Operationen basieren auf dem im Abschnitt[49] Unifikation vorgestellten Algorithmus zur Berechnung eines Teilunifikators zweier Strukturen, der dort definierten Relation before und Typinformation. Es werden keine Referenzresolutionen diskutiert, die über die während der Übersetzung geleisteten hinausgehen. Dies ist jedoch ein interessanter Punkt zur Erweiterung des Systems.

3.2.1 Inkonsistenzen

Wie im vorangegangenen Kapitel beschrieben, werden inkonsistente Terminbeschreibungen bei der Übersetzung erkannt. Es wird dann entschieden, ob diese weiter bearbeitet und repariert werden oder die Eingabe von TEE sie zurückgweist. Dadurch wird vor allem sichergestellt, daß keine inkonsistenten TID generiert werden.

Die Frage nach der Konsistenz, bzw. Kompatibilität zweier Strukturen ist äquivalent zu der Frage, ob die Strukturen unifizierbar sind. Unifikation stellt sich als konjunktive Vereinigung zweier Formelmengen dar, wobei Individuenkonstanten und Variablen identifiziert werden. Damit ist die konsistente Vereinigung eine – eventuell informativere – Beschreibung des gleichen Objektes. Ist die so erzeugte Formelmenge inkonsistent, d.h. schlägt Unifikation fehl, bedeutet dies, daß beide Strukturen nicht Repräsentationen des gleichen Objektes sein können. Eine Möglichkeit für eine Reparaturmethode besteht darin, von den als inkonsistent erkannten TID durch Generalisierung eine Instanz zu inferieren, die die inkonsistenten Objekte subsumiert.

3.2.2 Vergleiche

Die Unifikation von Intervallstrukturen, aufgefaßt als Spezialisierung, erfordert eine Operation, die einen Teilunifikator zusammmen mit einem eventuell leeren

[49] S.S.31.

Restpaar berechnet. Diese Operation teilt damit die zugrundeliegenden Formelmengen in einen konsistenten Teil und in ein Paar von zusammengenommen inkonsistenten Teilmengen. Die durch die lineare Ordnug Spec vorgegebene Reihenfolge der Unifikation der Teilstrukturen gewährleistet, daß die Unifikation an dem unspezifischsten Attribut[50] fehlschlägt.

Damit kann der Teilunifikator als Referenzrahmen interpretiert werden, innerhalb dessen die durch die nicht unifizierbaren Restmengen beschriebenen temporalen Objekte zu vergleichen sind. Zusammen mit den durch die Typinferenz eingeführten und substituierten Variablen stellt der Teilunifikator die notwendige Bedingung für den Vergleich der Reststrukturen dar.

Um zu bestimmen, ob ein Fokuswechsel im Dialogverlauf stattgefunden hat, kann das Fehlschlagen der Unifikation als new bzw. das Gelingen als given interpretiert werden:

$$\mathsf{new}\colon \mathsf{IdFS} \times \mathsf{IdFS} \longrightarrow \mathsf{BOOL}$$
$$\mathsf{given}\colon \mathsf{IdFS} \times \mathsf{IdFS} \longrightarrow \mathsf{BOOL}$$

Außerdem ermöglicht die induktive Konstruktion einer Intervallbeschreibung das Testen von beliebigen Teilausdrücken der ZG auf ihr given/new-Verhältnis zu einer gegebenen Struktur. Hierdurch lassen sich die Teilausdrücke identifizieren, die bereits Gesagtes in einem Dialog wieder aufnehmen und solche, die neue Information hinzufügen.

Es ergeben sich die Allenschen Zeitrelationen direkt aus der Relation before[51] und den jeweiligen durch die partielle Unifikation berechneten Restmengen. Hierbei können Variablen dazu führen, daß Relationen nicht eindeutig bestimmbar sind, so daß Alternativen in Form von Mengen angegeben werden müssen.

3.2.3 Spezifizität

Um Äußerungen klassifizieren zu können, ist oft auch ein Vergleich in Hinsicht auf den Zugewinn an Information wichtig. Beschreibt eine TID id_1 ein eingeschränktes Intervall als eine TID id_2, kann deren Relation als Spezialisierung interpretiert werden. id_1 ist spezifischer als id_2 in Hinsicht auf die Inklusion der beschriebenen Intervalle. Diese Spezifitätsrelation wird durch die Relation dur modelliert.

Um für Intervallbeschreibungen $id_{1,2}$ zu entscheiden, ob id_1 spezifischer in Hinsicht auf den Typ als id_2 ist, müssen die Typen der Teilobjekte von id_1 und id_2 auf Subsumption getestet werden. id_1 subsumiert id_2, wenn die Werte der Attribute unifizieren und die Typen der Werte von id_1 die von id_2 subsumieren. TID id_1 ist spezifischer als id_2, gdw.:

[50] Das Attribut mit der gröbsten Granularität.

[51] S.S. 27.

- $(id_1,\, id_2) \in \mathsf{dur}$ oder

- unify$(id_1,\, id_2) \neq fail$ und $(id_2,\, id_1) \notin \mathsf{dur}$ und für alle Pfade P sowie Typen $\boldsymbol{d}_{1_P}$, $\boldsymbol{d}_{2_P}$ mit:

$$\delta(P,\, id1) : \boldsymbol{d}_{1_P}$$
$$\delta(P,\, id2) : \boldsymbol{d}_{2_P}$$

gilt:

$$\boldsymbol{d}_{2_P} \sqsubseteq \boldsymbol{d}_{1_P}$$

3.3 Architektur und Implementierung

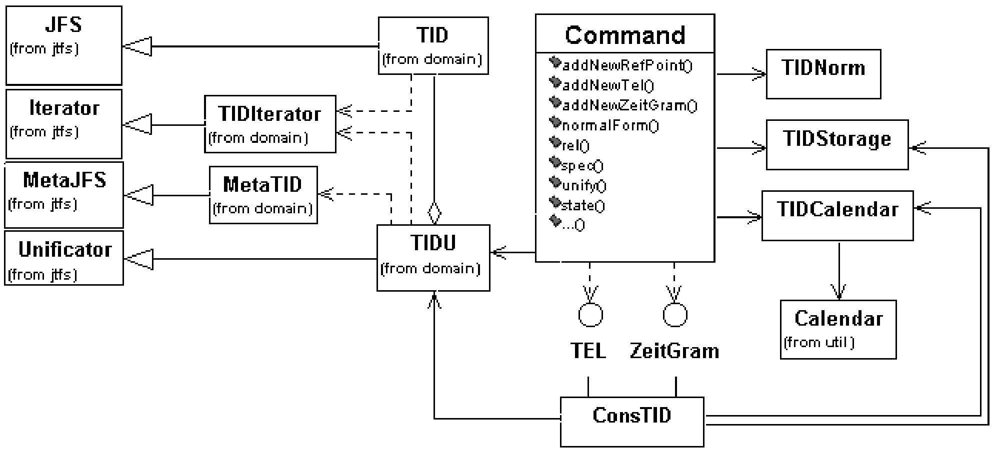

Abbildung 25: Architektur von TEE

Abb.25 skizziert die interne Architektur von TEE. TEE ist als Server in VERBMOBIL integriert und verfügt über eine TCP/IP Schnittstelle zur Kommunikation mit einem Client, die in dem Diagramm nicht dargestellt ist. Ein Interpreter erwartet die Nachrichten an der Schnittstelle und bildet die Anfragen des Client auf Methodenaufrufe von Command ab. Das Resultat einer Operation von Command wird als String in einer Variable state gespeichert, worauf die Methode state() zugreift und den Wert an den Client weitergibt.

TIDNorm implementiert eine Methode, die eine Normalform in der Syntax der Zeitbeschreibungssprache TEL aus TID generiert. Die Implementierung nutzt das *Visitor* Entwurfsmuster um aus Teilobjekten der durch TID spezifizierten Objekthierarchie sukzessive eine Zeichenkette von TEL zu generieren.

Instanzen von TID werden in einem Hashtable gespeichert, das von TIDStorage implementiert ist. TIDStorage bietet Operationen an, um die gespeicherten Objekte zu verwalten. Hierzu gehören Methoden zum Erzeugen von neuen Schlüsseln, das Löschen einzelner Einträge und verschiedene Methoden des Zugriffs auf Elemente der Tabelle.

Die semantische Interpretation der TID wird von TIDCalendar durch eine Abbildung auf Instanzen des `java.util.Calendar` realisiert. TIDCalendar nutzt das API von `java.util.Calendar`, um die Korrektheit von Instanzen der TID zu überprüfen und um eine Kalenderarithmetik zu implementieren.

Der Unifikator TIDU implementiert neben der beschriebenen Unifikationsoperation auch den Subsumptionstest für TID, da die Methode eine Variante der Unifikationsoperation darstellt. Auch beim Test der Subsumption werden zwei Strukturen parallel durchlaufen, es wird nur keine neue Struktur erzeugt, sondern jeweils die Typen der Teilstrukturen auf Subsumption getestet. TIDU nutzt die Spezialisierungen MetaTID und TIDIterator von JTFS , die eine Anpassung von JFS an die TID realisieren. Die Anpassung betrifft die atomaren Typen, die um einen Nachfolgertyp SuccAtomic und einen Listentyp erweitert sind, und TID, der auch um einen Listentyp erweitert ist. Die Spezialisierungen von JFS verhalten sich bei der Unifikation anders als JFS, so daß TIDU dies in Fallunterscheidungen realisiert.

TIDIteratoren geben Auskunft über den Typ des Objektes, von dem sie erzeugt wurden, und werden in Fallunterscheidungen verwendet. Die Spezialisierung MetaTID von MetaJFS definiert die Methode glb() für den Nachfolge- und Listentyp.

ConsTID implementiert die Regeln zur Konstruktion von TID aus temporalen Ausdrücken. Es werden von ConsTID aus Gründen der Abwärtskompatibilität die zwei Interfaces TEL und ZeitGram implementiert, da im System neben der Erweiterung TEL der ZG die ZG auch weiterhin verwendet wird. Dem Design von ConsTID liegt das *Builder* Entwurfsmuster zugrunde. ConsTID nutzt den Unifikator TIDU als Speicher der zu konstruierenden Struktur. TIDU speichert die Ergebnisstruktur einer Unifikation und ermöglicht es, diese durch sukzessive Aufrufe des Unifikators zu erweitern. Das inkrementelle Vorgehen von ConsTID beim Bau einer Struktur wird durch *Visitor*-Objekte gesteuert, die die Parsebäume der jeweiligen Grammatik TEL oder ZG durchwandern und Methoden von ConsTID mit Teilausdrücken des Zeitausdrucks der Eingabe aufrufen. Die Visitoren haben dabei die Rolle des Director in dem *Builder* Entwurfsmuster.

Die Grammatiken der Zeitbeschreibungssprachen liegen als BNF vor, so daß die Werkzeuge *JavaCC*[52] und *JTB*[53] zur automatischen Generierung der Parser und Interpreter der Zeitausdrücke verwendet werden. JTB erzeugt eine Objektstruktur als Repräsentation der Parsebäume einer kontextfreien Grammatik und einen *Visitor* dieser Struktur. Der generierte *Visitor* stellt eine Basisklasse dar, von der durch Spezialisierung *Visitoren* für konkrete Aufgaben

[52] *Java Compiler Compiler*, `http://www.suntest.com/JavaCC`

[53] *Java Tree Builder*, `http://www.cs.purdue.edu/homes/taokr/jtb`

erzeugt werden. Aus dem *Visitor* einer der Zeitbeschreibungssprachen $\mathcal{L}$(ZG) und $\mathcal{L}$(TEL) wird durch Spezialisierung ein *Director* zur Steuerung des *Builder* ConsTID, indem innerhalb der überladenen Methode visit() Methoden von ConsTID aufgerufen werden, die Instanzen von TID generieren und diese unifizieren.

4 Schlußbemerkung

Vor allem das für die Übersetzung gesprochener Sprache typische Problem der unvollständigen, partiellen Information motiviert, JTFS zur Repräsentation der temporalen Domäne zu verwenden. Die Domäne wird durch ein Typsystem modelliert, wobei Intervalle als Basiselemente durch den Typ TID beschrieben werden. Die Komponente TEE implementiert domänenspezifische Inferenzoperationen für den Typ TID.

Ein erweiterter Unifikationsalgorithmus für JFS wird vorgestellt, der eine Trennung von konsistenter und inkonsistenter Information ermöglicht und in TIDU implementiert. Die angegebene Ordnung Spec, die einer Rangfolge von Kalendereinheiten entspricht, kann als Ordnung der Granularität des bestehenden Beschreibungssystems und damit der Eigenschaften der Objekte der modellierten Domäne aufgefasst werden. Die Steuerung der Unifikationsoperatiopn durch Spec gewährleistet, daß der als inkonsistent erkannte Teil der Beschreibung eines Objektes sich auf die spezifischeren Eigenschaften bezieht. Es wurde gezeigt, wie diese Erweiterung der Unifikation in vollständiger und korrekter Weise die Anreicherung von TID mit Information ermöglicht.

In Verbindung mit dem Typsystem läßt sich durch die partielle Unifikation ein Ähnlichkeitsmaß zwischen Objekten berechnen. Damit kann eine Form von Klassifikation realisiert werden, welche die Granularität der Objekteigenschaften explizit einbezieht.

So ergibt sich aus der Typinformation, die aus der Kombination des Typs des Teilunifikators und der Typen der inkompatiblen Reste besteht, ein Kriterium zur Beurteilung der resultierenden Schlüsse. Die Unvollständigkeit des Wissens ist damit klar repräsentiert und kann für Revisionsmechanismen verwendet werden.

Das Verfahren ist nicht auf die vorgestellte temporale Domäne beschränkt, da es durch Anpassen der Ordnung der Eigenschaften der zu modellierenden Objekte für weitere Anwendungsgebiete mit ähnlichen Anforderungen adaptiert werden kann.

Die Darstellung temporaler Information als Intervalle wurde von Allen ([Allen, 1983], [Allen, 1984]) ausführlich untersucht. Hier werden Intervalle selbst nicht weiter strukturiert, sondern als atomar betrachtet. Vielmehr wird eine Menge von Relationen einer gegebenen Menge von Zeitintervallen als bekannt vorausgesetzt und für weitere Inferenzen genutzt. TEE inferiert hingegen ausgehend von dem Informationsgehalt der Beschreibung eines Zeitintervalls eine Menge von möglichen Relationen zwischen Intervallen. An dieser Stelle besteht eine interessante Erweiterungsmöglichkeit von TEE; denn durch Allens Intervallkalkül besteht die Möglichkeit, die berechneten Relationen zu desambiguieren.

A Abkürzungen

API *application programming interface*

AVM *attribute value matrix*

FS Merkmalstruktur (*feature structure*)

JFS Name des Interfaces und Name des Typs des OOD der FS (*java feature structure*)

JFSImpl Name der Implementierenden Klasse des OOD der FS (*java feature structure implementation*)

JTFS Name des Systems der in Java implementierten getypten Merkmalstrukturen (*java typed feature structures*)

$\mathcal{L}(G)$ die durch die Grammatik G beschriebene Sprache als Menge der wohlgeformten Ausdrücke.

MetaJFS Name des Interfaces und Name des Typs der Metaklasse von JFS

MetaJFSImpl Name der implementierenden Klasse des Interfaces MetaJFS

MOP *meta object protokol* in Java durch die Reflection API realisiert

TFS getypte Merkmalstruktur (*typed feature structure*)

TID *Time Interval Description*

TEL *Temporal Expression Language*

UML *Unified Modeling Language*

VIT *Verbmobil Interface Term*

ZG *ZeitGram*

Ausdrücke der ZeitGram

`ao(LIST_OF_DATE_EXPRS+)` ist Auswahlliste von Datumsausdrücken.

`BOUNDARIES` ist ein geschlossener Intervall gegeben durch einen Anfangs- und Endpunkt sowie eine Charakterisierung ob ein Punkt innerhalb des Intervalls oder der ganze Intervall gemeint ist.

`COUNTER` können Zahlenangaben oder deiktische Ausdrücke wie "dieser" "letzter" etc. sein.

`DATE_EXPR` ist ein allgemeiner evtl. quantifizierter Datumsausdruck oder eine Auswahlliste von Datumsausdrücken.

`DOM` *Day Of Month*

`DOW` *Day Of Week*

`DURATION` ist eine Dauer.

`FUZZYFIER` (`::= about`) charakterisiert Unschärfe.

`LIMIT` ist ein offener Intervall gegeben durch einen Anfangs- oder Endpunkt.

`MOD` (`early | late ...`)

`MOY` *Month Of Year*

`POD` *Period Of Day*

`POINT` ist ein Zeitpunkt also eine Uhrzeitangabe.

`POINTLIKE` ist eine Zeitspanne, die sich als "Punkt" konzeptionalisieren läßt.

`POY` *Period Of Year*

`TEMPOBJ` sind nichtquantifizierte Ausdrücke.

`TOD` *Time Of Day*

`WOM` *Week Of Month*

`WOY` *Week Of Year*

B Berechenbarkeit von *UnifyMod*

Terminierung

Der Algorithmus *unifyMod* terminiert bei Eingabe von JFS $f_{1,2}$, gdw. in endlich vielen Schritten eine der Abbruchbedingungen erreicht wird. Sei die Tiefe $\mathcal{D}(f)$ einer JFS gegeben als die Länge des längstmöglichen Pfades von der Wurzel von f zu einer atomaren Teilstruktur. $\mathcal{D}(f)$ ist endlich, da keine zyklischen Strukturen zugelassen werden und damit keine unendlich langen Pfade existieren. Die Überprüfung auf Freiheit von Zyklen der zu unifizierenden Strukturen wurde aus Effizienzgründen unterlassen, da Date offensichtlich keine rekursiven Typen enthält, so daß die korrekte Typisierung der Variablen hinreicht, damit keine zyklischen Strukturen als Eingabe auftreten können. Durch einfaches Umbenennen der Variablen der zu unifizierenden Strukturen muß vor der Unifikation gewährleistet werden, daß diese keine gemeinsamen Variablen enthalten.

Die Terminierung läßt sich durch Induktion über $n = \max(\mathcal{D}(f_1), \mathcal{D}(f_2))$ zeigen. Hierzu sollen Variablen $\boxed{x}$ so interpretiert werden, daß $\mathcal{D}(\boxed{x}) = \mathcal{D}(\Theta(\boxed{x}))$.

So gilt bei $n = 1$ für JFS $f_{1,2}$ einer der Fälle 1.a, 1.b oder 1.c. Funktionale Werte sind auf die Nachfolgefunktion *succ* beschränkt, so daß im Falle von 1.a *unifyFun* und damit *unifyMod* immer terminiert. Im Falle von 1.b terminiert der Algorithmus offensichtlich, da nur Atome verglichen werden. Trifft 1.c zu, erfolgt ein erneuter Aufruf von *unifyMod*, wobei die Variable(n) durch Θ ersetzt werden. Hierbei werden ungebundene Variablen zu $[\top]$ und gebundene durch das jeweilige Atom ersetzt. *unifyMod* terminiert also, da nun Fall 1.b gilt.

Für $n = k + 1$ betrachten wir folgende Fälle:

1. $f_{1,2}$ ist (sind) Variable mit $\mathcal{D}(\Theta(f_{1,2})) > 1$, dann erfolgt in 1.(c).i ein Aufruf mit den Bindungen der Variablen oder eine Variable wird gebunden, d.h. es erfolgt ein Aufruf der Form *unifyMod*$(\Theta(f_1), \Theta(f_2))$ mit $n = \max(\mathcal{D}(\Theta(f_1)),\mathcal{D}(\Theta(f_2)))$. n bleibt also unverändert, es kommt nunmehr aber nur noch der folgende Fall 2. in dem erneuten Aufruf in Betracht.

2. Falls kein Subtyp d für $f_{1,2}$ (bzw. $\Theta f_{1,2}$ s.o.) in 2. inferiert werden kann bricht der Algorithmus ab. Sonst gilt die Behauptung für *unifyMod*(q_1, q_2) aus 3. offensichtlich, da $\max(\mathcal{D}(d_1), \mathcal{D}(d_2)) = \max(\mathcal{D}(f_1), \mathcal{D}(f_2)) - 1 = k$. Ferner terminiert die Schleife über die Punkte 3. 4. 5., da diese bei fehlgeschlagener Unifikation in Punkt 4. verlassen wird oder die verarbeiteten Teilstrukturen gelöscht werden.

□

Korrektheit

Der Algorithmus *unifyFun* ist korrekt, gdw. gilt, daß wenn *unifyFun* bei Eingabe von JFS $f_{1,2}$ mit $(d, \mathrm{R}) =$ *unifyFun*(f_1, f_2) terminiert, d Teilunifikator und $\mathrm{R} \in$ Rest ist. Die Behauptung, daß $\mathrm{R} \in$ Rest ist, hängt eng mit der Definition des

Typs Date und der Auswahlfunktion *selectA* zusammen. Im allgemeineren Falle kann Rest als Menge nicht-unifizierender Strukturen und somit unabhängig von der Ordnung Spec definiert werden.

Es ist also zu zeigen, daß wenn nach n Schritten der Algorithmus durch "return" beendet wird, die zurückgegebenen Werte korrekt sind.
Für $n = 1$ bei Eingabe von JFS $f_{1,2}$ terminiert *unifyMod* bei 1.(a), 1.(b) und 2. Funktionen sind hier auf die Nachfolgefunktion *succ* für atomare Typen beschränkt, deren Korrektheit direkt ersichtlich ist. Werden allgemein Funktionen zugelassen, muß im einzelnen die Korrektheit überprüft werden. Bei den Fällen 1.(b) und 2. ist der Wert $([\top], (f_1, f_2))$. $[\top]$ ist Teilunifikator beliebiger Paare von JFS. Im allgemeinen Fall gilt, daß $(f_1, f_2) \in \mathsf{Rest}$, da $f_{1,2}$ entweder inkompatible Atome (1.(b)) sind oder kein gemeinsamer Subtyp inferiert werden kann. In Hinsicht auf die in *selectA* implementierte Ordnung Spec und die Typhierarchie von Date terminiert *unifyMod* frühestens bei $n = 2$, d.h. bei der Unifikation der Substruktur unter YEAR. Offensichtlich gilt für JFS $f_{1,2}$ vom Typ Date mit inkompatiblen Werten unter YEAR $max_{\mathsf{Spec}}(f_1) \neq max_{\mathsf{Spec}}(f_2)$ und damit $(f_1, f_2) \in \mathsf{Rest}$.

Für $n = k + 1$ kommen zusätzlich zu 1.(a),(b) und 2. die Fälle 1.(c)iii. und 4. sowie 5. in Betracht. Die Behauptung gilt für $n = k$ und damit ist (d, R) in 1.(c).i. und auch $(\boxed{\mathrm{x}}d, \mathrm{R})$ korrekt. Falls für $f_{1,2}$ kein Subtyp existiert, ist das Paar (f_1, f_2) nichtunifizierender Rest der Eingabe; es gilt jedoch nicht notwendig $max_{\mathsf{Spec}}(f_1) \neq max_{\mathsf{Spec}}(f_2)$. Also muß die Definition von Rest geändert werden, sobald die Typhierarchie von Date nicht mehr gewährleistet, daß für korrekt getypte Eingaben immer ein Subtyp inferiert werden kann. Andernfalls wird der gemeinsame Subtyp in 2. inferiert und der Teilunifikator d initialisiert. In 3. wird d_q sowie R für die nach Spec ausgewählten Teilstrukturen in k Schritten berechnet. Aus der Korrektheit von *subType* (2.) folgt, daß d für das ausgewählte Attribut A definiert ist und damit $\delta(A, d) := d_q$ einen Teilunifikator erzeugt. Da während der Schleifendurchläufe (3.,4.,5.) in 4. sukzessive die Eingabestrukturen gemäß Spec reduziert werden, sind $f_{1,2}$ korrekt in Bezug auf die Spezifikation von Rest. Also ist auch return(d, R) in 4. und return$(d, ())$ korrekt. □

Vollständigkeit

Der Algorithmus *unifyMod* ist vollständig, gdw. gilt, daß, falls $(d,\ \mathrm{R})$ für ein Paar von JFS $f_{1,2}$ existiert, wobei d Teilunifikator und R der zugehörige Rest ist, *unifyMod*$(f_1,\ f_2)$ das Paar $(d,\ \mathrm{R})$ berechnet.

Seien $f_{1,2}$ JFS vom Typ Date, d leerer Teilunifikator und somit $\mathrm{R} = (f_1,\ f_2)$. Nach Definition von Rest ist $max_{\mathsf{Spec}}(f_1) \neq max_{\mathsf{Spec}}(f_2)$ und offensichtlich *unifyMod*$(f_1,\ f_2) = (d,\ \mathrm{R})$, da der Algorithmus durch *selectA* gesteuert dem Pfad der maximal unspezifischen Attribute folgend die als erste zu unifizierende Atome $max_{\mathsf{Spec}}(f_{1,2})$ identifiziert und so die Rekursion mit dem Wert ($[\ \top\ ]$, $(max_{\mathsf{Spec}}(f_1),\ max_{\mathsf{Spec}}(f_2))$) in 1. abbricht. In 4. ist also die Bedingung $\mathrm{R} = ()$ nie erfüllt, so daß keine Teilstrukturen aus der Eingabe gelöscht werden können und somit R korrekt berechnet wird.

Angenommen d ist für JFS $f_{1,2}$ vom Typ Date nicht leer und Rest $\mathrm{R} = (r_1,\ r_2)$. $r_{1,2}$ sind echte Teilstrukturen von $f_{1,2}$, wobei nach Definition von Rest $max_{\mathsf{Spec}}(r_1) \neq max_{\mathsf{Spec}}(r_2)$. Der Teilunifikator d ist JFS vom Typ Date und enthält alle Attribute aus $f_{1,2}$, die unspezifischer als $max_{\mathsf{Spec}}(r_{1,2})$ sind.
d ist Teilunfikator, also ist t = *subType*$(f_1,\ f_2)$ mit t:d definiert. Aus dem gleichen Grund ist auch für alle Unterknoten von d mit Typ t' = *subType*$(f_1',\ f_2')$ definiert, wobei $f_{1,2}'$ das unifizierte Paar von eventuell leeren Teilstrukturen aus $f_{1,2}$ ist. Der Teilunifikator wird also im Schritt 2. und der Schleife 3.4.5. korrekt berechnet, da durch *selectA* die unspezifischen Attribute gewählt und, wie gezeigt, der entprechende Subtyp inferiert und initialisiert werden kann. Unter 4. werden nach erfolgreicher Unifikation der Teilstrukturen diese aus der Eingabe gelöscht, so daß bei einem “Fehlschlagen” der Unifikation die verbliebenen Strukturen aus der Eingabe die entsprechenden Reste $r_{1,2}$ darstellen. □

C API Dokumentation von TEE

Interface Command

public interface Command

Spezifikation der Operationen von TEE. Das Interface wird von der Klasse MsgInterpreter benutzt, um die entsprechenden Kommandos zu realisieren. Alle Methoden sind vom Typ void und schreiben Rückgabewerte als String in eine Variable. Diese können mit der Methode state() abgefragt werden. Um Operationen auf temporalen Ausdrücken der Grammatiken TEL und ZeitGram durchführen zu können, müssen diese zunächst von einem Parser der jeweiligen Grammatik analysiert werden. Ist der temporale Ausdruck syntaktisch korrekt, wird eine TID (Time Interval Description) generiert und unter einem Referenzschlüssel gespeichert.

See Also: MsgInterpreter

Methoden

public abstract String state()

Gibt das Resultat der letzten Operation zurück. Dieser wird als Seiteneffekt durch Aufruf einer der Methoden gesetzt. Initialwert ist ök".

public abstract void addNewTel(String key, String telExpr)

Konstruiert TID aus telExpr. Diese wird unter der Referenz key in ein Hashtable eingefügt. Wenn key bereits belegt ist, wird die erzeugte Struktur mit neu generiertem newKeyäbgelegt. Tritt ein Syntaxfehler auf, wird mit parserErroräbgebrochen.

Parameters:

- key ist eindeutiger Bezeichner.
- telExpr ist Ausdruck der Grammatik TEL der Kategorie DATE.

Returns: "konsistent", ïnkonsistent", newKeyöder parserError"

public abstract void addNewZeitGram(String key, String zgExpr)

Konstruiert TID aus zgExpr. Dabei wird zusätzlich ein TEL-Ausdruck generiert. Die konstruierte Struktur wird unter der Referenz key in ein Hashtable eingefügt. Bei einem Syntaxfehler wird nichts eingetragen und bei belegtem key erfolgt ein neuer Eintrag unter newKey".

Parameters:

- key ist eindeutiger Bezeichner.
- zeitGramExpr Ausdruck der Grammatik ZeitGram der Kategorie LIST-OF-DATE-EXPR

Returns: "konsistent", ïnkonsistent", newKeyöder parserError"

public abstract void addNewRefPoint(String telExpr)
TID wird aus Zeitausdruck konstruiert und als Referenzpunkt gespeichert. Der Referenz-Zeitpunkt wird zur Interpretation weiterer Eingaben verwendet. Bei Syntaxfehler wird abgebrochen.

Parameters:

- telExpr ist Ausdruck der Grammatik TEL der Kategorie DATE.

Returns: "konsistentinkonsistentöder parserError"

public abstract void unify(String key1,String key2)
Unifiziert zwei TIDs. Die unter key1 und key2 gespeicherten Strukturen werden unifiziert und das Resultat unter neuer Referenz newKey"gespeichert. Wenn keine Struktur unter key1 oder key2 eingefügt wurde, bricht die Methode mit notExistäls Resultat ab.

Returns: newKeyfailöder notExist"

public abstract void rel(String key1,String key2)
Berechnet Allensche Relation ALLEN_REL zwischen zwei TIDs. Die Relation der unter key1 und key2 gespeicherten Strukturen wird berechnet. Wenn keine Struktur unter key1 oder key2 eingefügt wurde, bricht die Methode mit notExistäls Resultat ab.

```
ALLEN_REL::=  "overlaps" | "meets" | "before" |
"contains" | "finished_by" | "starts"|
"equal" | "started_by" | "during" |
"finishes" | "after" | "met_by" | "overlapped_by"
```

Returns: ALLEN_REL oder notExist"

public abstract void before(String key1,String key2)
Testet ob die before-Relation zwischen zwei TIDs gilt. Die unter key1 und key2 gespeicherten TIDs werden (partiell) unifiziert und der Rest auf before getestet. Wenn keine Struktur unter key1 oder key2 eingefügt wurde, bricht die Methode ab.

Returns: true", falseöder notExist"

public abstract void spec(String key1,String key2)
Testet ob die Spezieller-Relation zwischen zwei TIDs gilt. Die Typen der unter key1 und key2 gespeicherten TIDs werden in Hinsicht auf genereller/spezieller getestet. Wenn keine Struktur unter key1 oder key2 eingefügt wurde, bricht die Methode ab.

Returns ßame", "gen", ßpec", ündefinedöder notExist"

public abstract void getUnitValue(String unit,String key)
Gibt Wert der TID für angegebene Unit zurück.

Returns: "(Value1,Value2)öder notExist"

public abstract void normalForm(String key)
Gibt TID in Normalform aus. Dabei ist die Normalform ein Ausdruck der Grammatik TEL von der Kategorie Intervall, der aus Kalender-Kategorien Jahr Monat etc. aufgebaut ist.

Returns: NormalFormöder notExist"

public abstract void del(String key)
Entfernt TID aus dem Speicher. Resultat ist notExistfalls key nicht existiert.

Returns: ököder notExist"

public abstract void dateExpr(String key)
Gibt TEL-Ausdruck zu der referenzierten TID zurück.

Returns: "DATE-EXPRöder notExist"

public abstract void show(String key)
Öffnet Fenster und druckt referenzierten TID darin aus.

Returns: ököder notExist"

public abstract void delAll()
Entfernt alle TIDs aus dem Speicher.

Returns: ök"

public abstract void newTID()
Erzeugt neue leere TID. Diese wird unter neuem key in Hashtable gespeichert.

Returns: "key"

public abstract void newKey()

Erzeugt neuen key. Der neu erzeugte key kann zum Speichern von TIDs verwendet werden.

Returns: "key"

Class MsgInterpreter

```
java.lang.Object
 |
 +----MsgInterpreter
```

---

public class MsgInterpreter extends Object implements MsgInterpreterIfc

Nachrichteninterpretierer für TEE (Temporal Expression Evaluator). Realisiert flache Analyse der Eingabe und ruft Command-Objekt mit Argumenten auf.

BNF der Eingabe

- *DATEXPR* ist Termin- bzw. Datumsausdruck und wird von der BNF der jeweiligen Zeitgrammatik (ZeitGram bzw. TEL) beschrieben.
- *KEY* ist beliebiger String ohne

  `"()[],\#[\t\n\r "`

  (z.B. aus VitID generiert)

```
MSG ::= ADD "(" "tempex" "(" KEY "," DATEXPR ")" ")" |
CMD2 "(" KEY "," KEY ")" |
CMD1 "(" KEY ")" | CMD0 "(" ")"

ADD ::= "TEEaddNewTel" |
"TEEaddNewZeitGram" |
"TEEaddNewRefPoint"
```

```
CMD2 ::= "TEEunify" |
"TEErel" |
"TEEbefore" |
"TEEspec"

CMD1 ::= "TEEnormalForm" |
"TEEdel" |
"TEEdateExpr"|
"TEEgetUnitValue"|
"TEEshow"

CMD0 ::= "TEEdelAll" |
"TEEnewTID" |
"TEEnewKey "
```

See Also: Command

Variable

protected Command thisCommand
Referenz auf das Command-Objekt, das die Methoden von TEE ausführt.

protected String thisDelimiters
Zeichenkette, die Teile der Eingabe begrenzt.

protected MsgInterpreter.CmdTable thisCmdTable
Tabelle der verfügbaren Methoden.

Konstruktor

public MsgInterpreter(Command command)
Initialisiert die Referenz thisCommand mit command.

Methoden

public String result()
Holt Ergebnis des Commandobjekts nach dem Aufruf einer Methode.

public void interpretMsg(String msg)
Interpretiere Eingabezeichenkette msg. Die Zeichenkette wird nach Begrenzern (Delimiters) zerlegt. Die Teile werden durch eine Tabelle als

Kommandos mit Argumenten interpretiert, die als Methoden des Commandobjekts aufgerufen werden.

Abbildungsverzeichnis

Literatur

[Alexanderson *et al.*, 1997] J. Alexanderson, N. Reithinger, E. Maier, *Insights into the Dialogue Processing of* VERBMOBIL, Verbmobil-Report 191, 1997.

[Allen, 1983] J.F. Allen,*Maintaining Knowledge About Temporal Intervals*, Communications of the ACM 26(11), 832-843, 1983.

[Allen, 1984] J.F. Allen, *Toward a general theory of action and time*, Artificial Intelligence 23: 123-154, 1984.

[Burkhardt, 1997] R. Burkhardt, *UML – Unified Modeling Language*, Addison Wesley, 1997.

[Carpenter, 1992(1)] B. Carpenter, *ALE – The Atribute Logic Engine: User's guide*, Technical Report, Laboratory for Computational Linguistics, Philosophy Department, Carnegie Mellon University, Pittsburgh, PA 1992.

[Carpenter, 1992(2)] B. Carpenter, *The Logic of Typed Feature Structures*, Cambridge University Press, 1992.

[Eisele, Dörre, 1990] A.Eisele, J.Dörre, *Disjunctive Unification*, IBM IWBS Report 124, 1990.

[Endriss, 1998] U.Endriss, *Semantik zeitlicher Ausdrücke in Terminvereinbarungsdialogen*, Verbmobil-Report 227, 1998.

[Eriksson, Penker, 1998] H-E. Erikson, M.Penker, *UML Toolkit*, John Wiley & Sons, Inc., 1998.

[Flanagan, 1997] , D. Flanagan, *Java in a Nutshell*, O'Reilly, 1997.

[Gamma *et al.*, 1995] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Pattern, Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995.

[Gazdar, Mellish, 1989] G. Gazdar, C. Mellish, *Natural Language Processing in LISP, An Introduction to Computational Linguistics*, Addison Wesley, 1989.

[Krieger, Schäfer, 1994] H-U. Krieger, U. Schäfer, *TDL–A Type Descrition Language for Constraint-Based Grammars*, COLING–94, 1994.

[Krieger, 1995] H-U. Krieger, *TDL–A Type Descrition Language for Constraint-Based Grammars, Foundations, Implementation and Applications*, Dissertation, Universität des Saarlandes, 1995.

[Küssner, Stede, 1995] U. Küssner, M. Stede, *Zeitliche Ausdrücke: Repräsentation und Inferenz* Verbmobil Memo 100, 1995.

[Laffra, 1997] C. Laffra, *Advanced Java*, Prentice Hall, 1997.

[Doug Lea] Doug Lea, *Draft Java Coding Standard*, `http://gee.cs.oswego.edu/dl/html/javaCodingStd.html`.

[Martelli, Montanari, 1982] A. Martelli, U. Montanari, *An Efficient Unification Algorithm*, ACM Transactions on Programming Languages and Systems, 4(2):258-282, 1982.

[Martin, Odell, 1998] J. Martin und J. J. Odell, *Object Oriented Methods, A Foundation* (zweite Ausgabe), Prentice Hall, 1998.

[Odell, 1988] J. J. Odell, *Advanced Object-Oriented Analysis & Design Using UML*, SIGS Books & Multimedia, 1998.

[Rational, 1998] Rational Software Corporation, *UML Summary*, `http://www.rational.com/uml/`, 1998.

[Stede, Haas, Küssner, 1998] M. Stede, S. Haas, U. Küssner *Understanding and tracking temporal descriptions in dialogue*, Konvens, Bonn 1998.

[UML1.1] www.omg.org/library/schedule/Technology_Adoptions.htm