Universität Hamburg
Fachbereich Informatik

Studienarbeit

Ein Typsystem zur Analyse dialogorientierter Workflows in kooperativen Informationssystemen

eingereicht bei:
Prof. Dr. Florian Matthes
Arbeitsbereich Softwaresysteme
TU HAMBURG HARBURG

vorgelegt von:
Patrick Hupe
Frickestraße 85
20251 Hamburg
Tel: 040 / 48 14 96

Hamburg, den 19.11.98

Inhaltsverzeichnis

1. EINLEITUNG 5

1.1 ZIEL DER ARBEIT 5
1.2 GLIEDERUNG DER ARBEIT 5

2. DAS MODELL DER BUSINESS CONVERSATIONS 6

2.1 RÜCKBLICK AUF DAS BISHERIGE SYSTEM 8
2.2 PROBLEME DER VALIDIERUNG DES CODES 9
2.3 ZUSÄTZLICHE ANFORDERUNGEN AN DEN TYPPRÜFER 10

3. MODELLBILDUNG 12

3.1 FORMALISIERUNG DER ABLÄUFE ZWISCHEN KUNDE UND DIENSTLEISTER 12
3.1.1 Die Spur (trace) 13
3.1.2 Die Spursicht (traceview) 15
3.1.3 Semantik der Spursichten 15
3.2 PROZEßORIENTIERTE MODELLIERUNG 17
3.2.1 Das Erzeugen von Nachfolgedialogen und Abfragen 17
3.2.2 Die Modellrepräsentation 20
3.2.3 Das Prozeßmodell 22
3.2.4 Der Prozeßevaluator 24
3.2.5 Semantische Bedingungen der vier Fälle 30
3.2.6 Schleifen 33
3.3 STRUKTURORIENTIERTE MODELLIERUNG 37
3.3.1 Das Strukturmodell 37
3.3.2 Algorithmus zur Namensauflösung 40
3.4 KOPPLUNG VON STRUKTURORIENTIERTER UND PROZEßORIENTIERTER ANALYSE 40

4. IMPLEMENTIERUNG 41

4.1 DIE GLIEDERUNG DES SET 42
4.2 DAS SET-OBJEKTMODELL 44
4.2.1 Die Modellrepräsentation: TBCModel 44
4.2.2 Die Modellrepräsentation: Teil ModelRep 45
4.2.3 Die Historienklassen 46
4.3 DER SET–CHECKER 48
4.3.1 Konstrukte 48
4.3.2 Checker 52
4.3.3 Der Strukturevaluator 53
4.3.4 Der Prozeßevaluator 54
4.4 PROGRAMMIERSPRACHEN (PRL) – KONZEPTE 56

5. EINBETTUNG DES SET IN DIE TBC 59

6. VERGLEICH VON MODELLIERUNG UND IMPLEMENTATION 62

ANHANG A 63

DIE ABSTRAKTE SET - SPRACHE 63
DIE KONKRETE UMSETZUNG DER SET – SPRACHE IN TYCOON 2 64

ANHANG B 65

AGENTENPRÜFER (AGENT CHECKER) 65

ANHANG C 67

PAKETDIAGRAMME SET 67

ANHANG D 69

GENERIERTE BEISPIELKLASSEN FÜR SPUR (TRACE) UND SPURSICHT (TRACEVIEW) 69

LITERATURVERZEICHNIS 73

1. Einleitung

Am Arbeitsbereich Softwaresysteme der TU Hamburg - Harburg wurde das Modell der Business Conversations zur Kooperation und Koordination von Agenten entwickelt.
Bei den Agenten kann es sich sowohl um Softwareagenten als auch um Menschen handeln. Die Business Conversations definieren einen Kontext, in dem zwei Agenten langfristige Konversationen führen, in der sie durch Austausch von Dialogen miteinander kommunizieren. Eine Spezifikation der Konversation definiert für beide Agenten, welche Dialoge existieren, welche Abfolgen von Dialogen möglich sind und welche Inhalte (*dialog contents*) in den Dialogen ausgetauscht werden können. Die in der Spezifikation definierten Dialogabläufe werden bei Softwareagenten durch sogenannte Regeln (*business rules*) implementiert. Es gibt bisher keine Möglichkeit, zu prüfen, ob der Programmcode der Regeln der Konversationsspezifikation entspricht.

1.1 Ziel der Arbeit

Ziel der Studienarbeit ist es, zu untersuchen, inwieweit der Code der Regeln auf korrekte Implementierung der Spezifikation geprüft werden kann und einen um Zustandsinformation erweiterten Typüberprüfer (*state enriched typechecker, SET*) zu implementieren, der die im ersten Teil dargestellten Überprüfungen durchführt. Normale Typprüfer, wie sie für Programmiersprachen üblich sind, sind nicht dazu in der Lage, da in ihnen dynamische Agentensysteme nicht formalisiert sind.

Das dynamische Agentensystem basiert auf der Implementation der Business Conversations [Mat97,Wegn98] in Tycoon 2, einer am Arbeitsbereich und bei der Firma Higher-Order entwickelten Programmierumgebung für persistente, verteilte Systeme, welche anwenderorientierte Informationssysteme in offenen Umgebungen bereitstellen [Wahl98].

Tycoon 2 ist reflexiv, d.h., daß Scanner, Parser und Compiler der Tycoon-Sprache TL2 in Anwendungen genutzt werden können und das System zur Laufzeit erweitert werden kann. Dieses hat eine direkte Herangehensweise und Konzentration auf die Problemstellung ermöglicht und die Lösung der Aufgabe wesentlich erleichtert.

1.2 Gliederung der Arbeit

In Kapitel 2 wird das Modell der Business Conversations vorgestellt. Es wird der bisherige Stand des in Tycoon 2 implementieren Systems kurz dargestellt und anhand von Beispielen die Probleme erläutert, die die Entwicklung des Typprüfers begründen.
In Kapitel 3 wird das Modell des Typprüfers (SET) formalisiert.
Kapitel 4 beschäftigt sich mit der Implementierung des Typprüfers in Tycoon 2.
Kapitel 5 stellt die Einbettung des Tycoon 2-SET in die Implementation der Business Conversations in Tycoon 2 dar.
Abschließend werden in Kapitel 6 die Modellbildung und die Implementierung verglichen.

2. Das Modell der Business Conversations

Das Modell der Business Conversations beschreibt die Koordination von Akteuren in verteilten Systemen [Mat97]. Als Akteure werden autonome, unabhängige Geschäftseinheiten verstanden, die mit anderen Akteuren in langfristigen Geschäftsprozessen arbeiten. Die Akteure besitzen eigenes Wissen, ein „Gedächtnis“ und eigene Fähigkeiten, die sie bei der Interaktion mit anderen Akteuren in den Geschäftsprozessen nutzen. Der Begriff des Akteurs umfaßt sowohl menschliche Akteure als auch Softwareakteure.

Zwei Akteure (ein Kunde und ein Dienstleister (*Customer und Perfomer*) kommunizieren miteinander nach dem Modell der „*conversation for action*“ von Winograd/Flores [WiFl87].

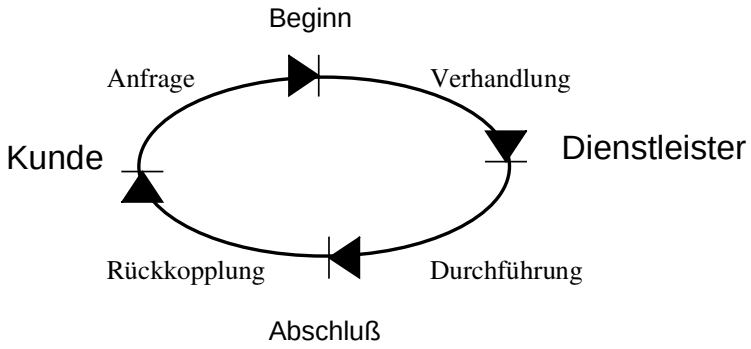


Abbildung1: Das Phasenmodell der Business Conversations

Anfragephase (request phase): Der Kunde macht sich beim Dienstleister bekannt und stellt diesem sein (noch allgemein formuliertes) Ziel für die folgende Konversation dar.

Verhandlungsphase (negotiation phase): Kunde und Dienstleister verhandeln miteinander in Form von Sprechakten mit dem Ziel, die Wünsche/Anforderungen des Kunden und die Dienstleistungen des Lieferanten zu verfeinern mit dem Ziel, eine für beide Seiten akzeptable Dienstleistung zu finden, welche sich neben dem eigentlichen Ziel des Kunden auch durch QoS - Parameter definiert.
Die Folge der Sprechakte (Dialoge) ist nicht willkürlich von beiden Seiten wählbar, sondern wird durch den Austausch der Spezifikation der Konversation festgelegt.

Durchführungsphase (performance phase): Der Dienstleister erbringt die vereinbarte Leistung. Er kann dabei dem Kunden darüber berichten, wie weit sein Auftrag bereits durchgeführt ist. In dieser Phase kann es zu weiterem Informationsaustausch zwischen dem Kunden und dem Dienstleister kommen.

Rückkopplungsphase (feedback phase): Der Kunde kann seine Zufriedenheit mit der vom Dienstleister erbrachten Leistung formulieren. Diese Phase umfaßt auch die ggf. vorhandene Bezahlung der Dienstleistung durch den Kunden.

Jeder Sprechakt zwischen Kunde und Lieferant wird über einen Dialogschritt realisiert. Ein Dialogschritt zwischen Lieferant und Kunden besteht immer aus zwei Phasen: Der Lieferant schickt dem Kunden einen Dialog, welcher einem elektronischen Formular mit Einträgen entspricht, von denen der Lieferant bereits einige ausgefüllt haben kann. Der Kunde füllt den Dialog aus und schickt ihn dem Lieferanten zurück, zusammen mit einer an diesem Dialogschritt zulässigen Anfrage (*request*), wie die Konversation weitergeführt werden soll. Sowohl die Dialoge als auch die zulässigen Anfragen des Kunden werden in der Spezifikation der Konversation definiert [Joh97].
Jede Konversation beginnt mit einem ausgezeichneten initialen Dialog, den der Kunde zu bearbeiten hat und er dem Lieferanten schickt. Eine Konversation endet, wenn beide Seiten über das Ende der Konversation befinden.

Die Inhalte der Dialoge (*contents*) sind hierarchisch gegliederte Dokumente, die Attribute enthalten. Diese werden genau wie die Dialoge in der Spezifikation definiert. Ein Attribut kann entweder ein Behälter für andere Attribute oder ein atomares Attribut sein. In Anlehnung an kommerzielle Systeme (u.a. SAP) sind wichtigste Klassen für Geschäftsprozesse (Currency, Date, ...) als atomare Typen bereits im System definiert.

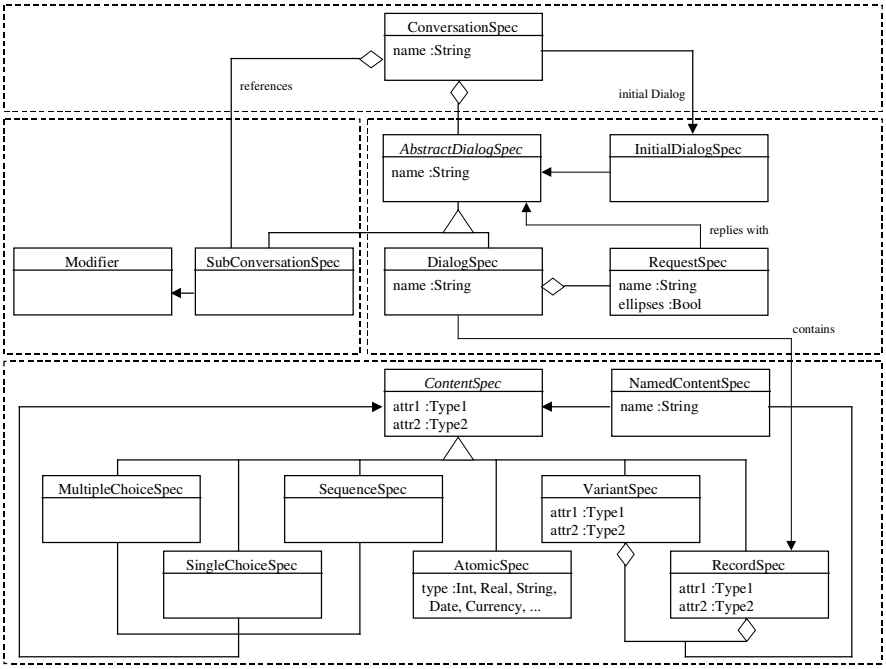


Abbildung 2: Objektmodell der Business Conversation Spezifikation

Die vorliegende Arbeit beschäftigt sich mit der Frage, wie der Code, der die Rollen des Kunden und Dienstleisters in Regeln implementiert, auf Konsistenz mit der Spezifikation der Konversation geprüft werden kann. Dabei geht es zunächst darum, daß Modell der Spezifikation darzulegen und die Bedingungen, die die Spezifikation an den Programmcode stellt, zu formalisieren. Daraus wird der (zustandserweiterte) Typproüfer (*typechecker*) konstruiert, welcher diese Bedingungen prüfen kann, ähnlich einem Typprüfer einer Programmiersprache. Das Prüfen des Codes gegen die Spezifikation wird im Verlauf ausgedehnt auf weitere Aufgabenbereiche. Insbesondere soll dabei untersucht werden, ob es sich um einen allgemeinen Ansatz handelt, der auch außerhalb des Modells der Akteursysteme angewendet werden kann.

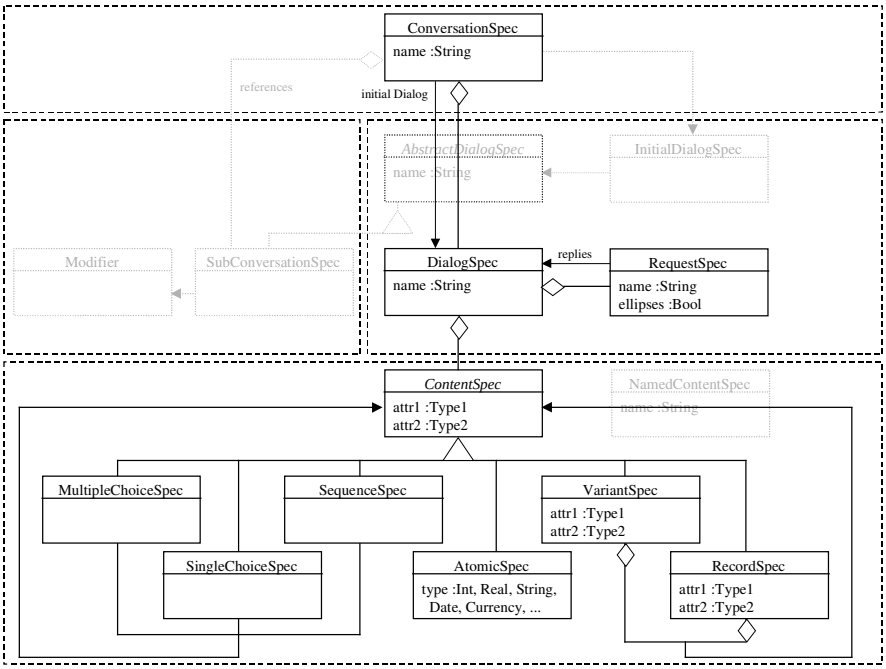


Abbildung 3: Objektmodell der Tycoon2 - Business Conversation (TBC) Spezifikation (reale Implementierung)

2.1 Rückblick auf das bisherige System

Um zu verstehen, welche Schwierigkeiten bei der Benutzung des bisherigen Systems [Joh97, Ripp98, Wegn98] auftauchen, machen wir es zunächst dem Leser vertraut.

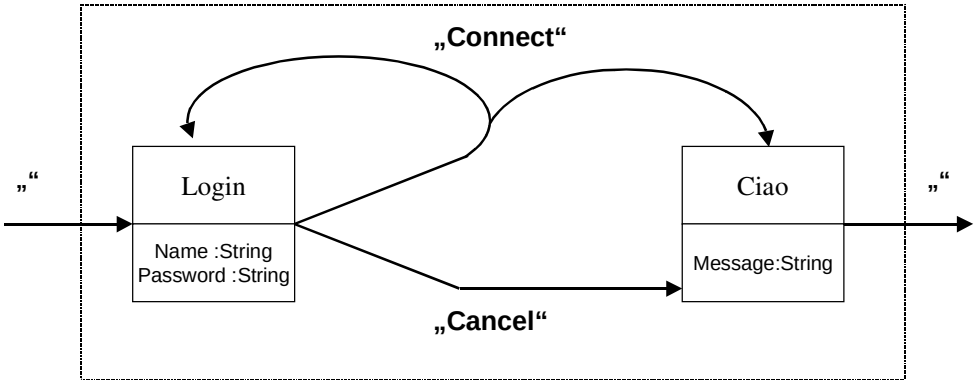


Abbildung 4: Beispielkonversation

Im Beispiel, welches in Abbildung 4 dargestellt ist, läuft die Konversation folgendermaßen: Der Kunde erhält vom System die Spezifikation der Konversation des Performers, in der definiert ist, daß die Dialoge „`Login`“ und „`Ciao`“ existieren, welche die Attribute (bzw. Variablen) „`Name`“ und „`Password`“ bzw. „`Message`“ enthalten. Vor Beginn der eigentlichen Konversation erzeugt der Dienstleister in der initialen Regel den Dialog „`Login`“ und preinitialisiert ggf. die Attribute. Der Dialog wird vom System an den Kunden gereicht, welcher die Attribute des Dialogs ausliest und bei Bedarf verändern kann. In dieser Beispielkonversation muß der Benutzer als Customer seinen Namen und sein Passwort in die dafür vorgesehen Attribute eintragen. Nachdem dies erfolgt ist, hat er laut Spezifikation die Wahl zwischen den Anfragen (*requests*) „`Connect`“ und „`Cancel`“. Die Anfrage „`Cancel`“ würde ihn auf jeden Fall in den Dialog „`Ciao`“ führen, wohingegen die Anfrage „`Connect`" ihn entweder zum Dialog „`Login`“ zurückführen würde (im Falle, daß das Passwort falsch ist und er es neu eingeben soll), oder aber Name und Passwort waren korrekt und er landet am Dialog „`Ciao`“. Die Wahl trifft nicht der Kunde und auch nicht das Business Conversation System, sondern der Dienstleister. Er erhält vom System den Dialog „`Login`“ und die Anfrage „`Connect`“ und kann nun wählen, ob er als nächsten Dialog „`Login`“ oder „`Ciao`“ erzeugt. Im Falle der Anfrage „`Cancel`“ kann der Dienstleister nicht wählen, die Spezifikation schreibt vor, daß er als nächsten Dialog „`Ciao`“ erzeugen und an das System reichen muß. Dabei kann das Attribut „`Message`“ bereits von ihm initialisiert werden. Gegeben der Fall, daß bei der Anfrage „`Connect`“ Name und Passwort korrekt waren oder der Kunde die Anfrage „`Cancel`“ gewählt hat, erzeugt der Dienstleister den Dialog „`Ciao`“ und reicht ihn an das System. Dieses schickt den Dialog an den Kunden, welcher den Inhalt des Dialogs wiederum sondiert und auch verändern kann. Der Spezifikation nach muß er nun den leeren Request wählen, welcher die Konversation enden läßt.

Anhand des Beispiels wird deutlich, daß zwischen zwei Dialogen jeweils einmal der Customer und einmal der Perfomer zum Zuge kommt.

Die Aufteilung in Phasen, in denen der Customer und solche, in denen der Performer agiert, sind in Abbildung 5 dargestellt.

Da die Konversation phasenweise abläuft, ist auch der Code, der in den Phasen ausgeführt wird, nicht in einem Stück geschrieben, sondern er liegt in vielen Fragmenten vor. Die Fragmente sind die Regeln (*business rules*). Eine Rule enthält jeweils den Code, der innerhalb einer Phase von einem Akteur ausgeführt werden muß.

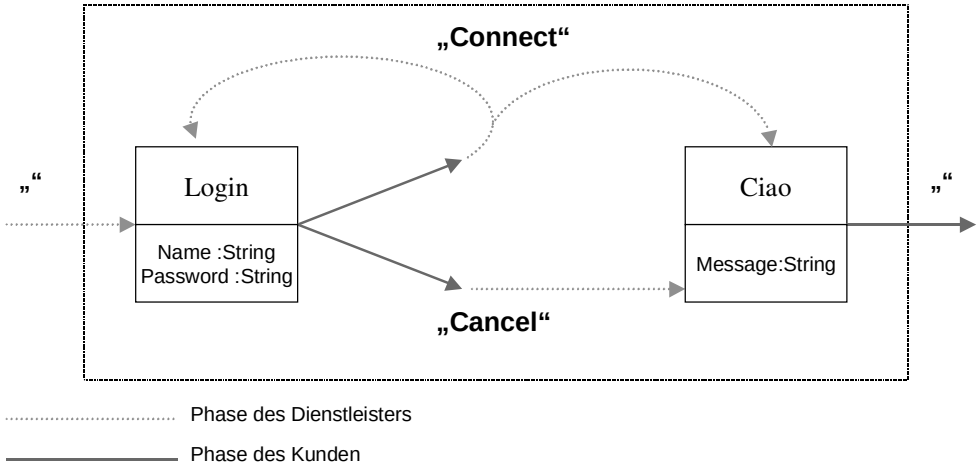


Abbildung 5: Aufteilung der Konversation in Kunde- / Dienstleisterphasen

Die Regeln von Kunde und Dienstleister unterscheiden sich. Die Dienstleisterregel (*performer rule*) erhält den aktuellen Dialog und die vom Kunden gewählte Anfrage (*request*) und muß einen neuen Dialog generieren. Die Kundenregel (*customer rule*) erhält den aktuellen Dialog und muß einen Request generieren.

Abbildung 6 zeigt die Zuordnung der Phasen zu den Regeln.

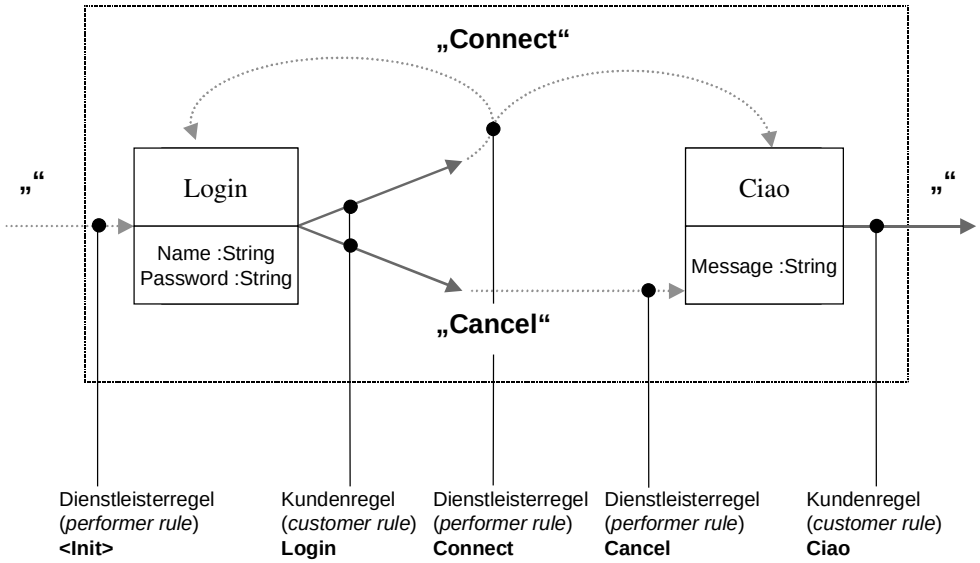


Abbildung 6: Zuordnung der Phasen zu den Regeln (r*ules*)

2.2 Probleme der Validierung des Codes

Probleme entstehen, wenn der Code der Rules nicht der Spezifikation entspricht. Bislang gibt es keine Möglichkeit, den Code gegen die Spezifikation zu prüfen. Es können mehrere Arten von Problemen auftreten:

- Es wird auf Inhalt von Dialogen (d.h. Variablen von Dialogen) zugegriffen, der nicht existiert.
- Auf den Inhalt der Dialoge wird nicht typrichtig zugegriffen.
- Es werden unzulässige Nachfolgedialoge / Requests erzeugt.

Alle Fehler können bisher nur zur Laufzeit erkannt werden. Erstrebenswert ist es jedoch, diese Fehler bereits zur Übersetzungszeit zu finden.

Für jeden der drei Fälle folgt ein kurzes Beispiel in TL2, der Programmiersprache von Tycoon 2.

Zugriff auf nicht-existente Dialoginhalte

Folgendes Codefragment aus der Dienstleisterregel „`Connect`“ zeigt dies:

```
namePw := "Login Name=" +
          dialog.content["Name"].str +
          " Password=" +
          dialog.content["Passwort"].str,
```

Codebeispiel 1: nicht-existente Variable „Passwort“ im Dialog „Login“

Der String namePw wird aus den Literalen "`Login Name`", "` Password=`" und Attributen des aktuellen Dialogs erzeugt. Dabei werden die Variablen über Stringbezeichner gewählt. Zu beachten ist, daß das Attribut "`Passwort`" nicht existiert, da es als "`Password`" deklariert ist. Dieser Fehler kann nicht zur Übersetzungszeit erkannt werden.

Typinkorrekter Zugriff auf Dialoginhalte

Der Typ der Variablen kann nicht aus dem Variablenbezeichner inferiert werden, da es sich hierbei um einen Stringbezeichner handelt. Aus diesem Grund muß der Typ im Ausdruck explizit angegeben werden, im Beispiel "`.str`", welches definiert, daß die Attribute „`Name`“ und „`Password`“ vom Typ String sind. Diese Typisierung wird allerdings vom Programmierer der Rules angegeben und kann der Typisierung in der Spezifikation widersprechen.

```
let id = dialog.content["Name"].int,
```

Codebeispiel 2: Typinkorrekter Zugriff auf die Stringvariable „Name“

Die Variable Name ist nach Spezifikation vom Typ String, wird hier jedoch als Integer angesprochen, was zu einem Laufzeitfehler führt.

Erzeugung unzulässiger Nachfolgedialoge / Anfragen

Laut Spezifikation kann auf die Anfrage „`Cancel`“ des Kunden nur der Dialog „`Ciao`“ folgen. Im Code wird jedoch der unzulässige Dialog „`Login`“ erzeugt, was erst durch eine Überprüfung zur Laufzeit festgestellt werden kann.

```
let d = conv.newDialog("Login"),
```

Codebeispiel 3: Generieren des falschen Nachfolgedialogs: „Login“ statt „Ciao“

Derselbe Fehler ist beim Generieren einer Anfrage auf Seiten des Kunden möglich.

2.3 Zusätzliche Anforderungen an den Typprüfer

Bei der Analyse zur Entwicklung des SET (*state enriched typecheckers*) wurden am Arbeitsbereich weitere Ideen entwickelt, die erst durch das Vorhandensein eines solchen Typprüfers umgesetzt werden können. Wir unterteilen sie in prozeßorientierte und strukturorientierte Ideen:

Prozeßorientiert

- Es soll ein problemloser Zugriff auf Variablen von bereits bearbeiteten Dialogen möglich sein. Dabei soll nach Möglichkeit der Zugriff ohne Benennung des beinhaltenden Dialogs möglich sein. Wenn

Programmcode direkt auf Variablen verweisen kann, die im Laufe der Konversation bereits definiert wurden, bringt dies einen Gewinn an Flexibilität mit sich, da einerseits die Reihenfolge der Dialoge und anderseits die Zugehörigkeit der Variablen zu den Dialogen verändert werden kann, ohne daß der Code, der sich auf die Variablen bezieht, nachträglich geändert und neu übersetzt werden muß.

- Die Wertbelegungen der Variablen, die bereits verwendet wurden, soll im nachhinein historisch sondiert werden können. Dabei soll die Belegung der Variablen am Ende jeder Phase festgehalten werden. Die früheren Variablenbelegungen sollen nicht nachträglich verändert werden können.

- Die Variablen der Dialoge sollen über Kurznamen ansprechbar sein, z.B. um auf tief verschachtelte Variablen leicht zugreifen zu können. Dies ist ein weiterer Gewinn an Flexibilität, da so Variablenhierarchien verändert werden können, ohne den Code ändern zu müssen.

Strukturorientiert

- Information über die Struktur der Variablenhierarchie soll nach Möglichkeit verwendet werden. Als Beispiel ist zu nennen, daß die Dialoginhalte eine Variantenstruktur kennen. Wird in einer solchen Struktur eine Ausprägung der Variante benutzt, so sollen die anderen Ausprägungen als nicht zulässig erkannt werden.

In der Arbeit wurden vorwiegend in der prozeßorientierten Richtung gearbeitet, die strukturorientierte Idee ist nur teilweise umgesetzt worden.

3. Modellbildung

Dieses Kapitel beschäftigt sich mit der Formalisierung des Typprüfers.

3.1 Formalisierung der Abläufe zwischen Kunde und Dienstleister

Zunächst bedarf es einer genauen Beschreibung der Phasen von Kunde und Dienstleister, aus der die Formalisierung entwickelt wird.

Bei der Betrachtung einer Konversation sind mehrere Punkte zu untersuchen:

- Wie können die nacheinander ablaufenden Phasen einer Konversation sinnvoll in eine zeitliche Achse gebracht werden?

  Jeder Dialog wird in drei Phasen verarbeitet. Zunächst wird der Dialog vom Dienstleister als Nachfolgedialog (*next dialog*) des aktuellen Dialogs erzeugt. In der nächsten Phase wird dieser vom Kunden als aktueller Dialog (*actual dialog*) bearbeitet, an den Dienstleister zurückgereicht und kann dort in der dritten Phase als aktueller Dialog verarbeitet werden. Nach diesen drei Phasen wird der Dialog in der Historie der Konversation (*history*) abgelegt.

- Wie beginnt eine Konversation, wie endet sie?

  Eine Konversation beginnt immer beim Dienstleister. In der einleitenden Phase existiert kein aktueller Dialog, es muß als Nachfolgedialog der initiale Dialog der Konversation erzeugt werden.
  Eine Konversation endet beim Kunden. Er bearbeitet den aktuellen Dialog und generiert die beendende Anfrage (*final request*).

- Welche Schritte müssen in den Phasen ablaufen, welche sind wahlfrei?

  In einer Dienstleisterphase muß ein Nachfolgedialog erzeugt werden, es kann der Inhalt des aktuellen und des Nachfolgedialogs sondiert und verändert werden; der Inhalt der Historie kann sondiert werden.
  In einer Kundenphase muß eine Anfrage generiert werden, es kann der Inhalt der aktuellen Dialogs verändert und sondiert werden, der Inhalt der Historie sondiert werden. Das Erzeugen eines Nachfolgedialogs ist nicht zulässig.

- Wie kann ein Zugriff auf Variablen aussehen, der vom enthaltenen Dialog abstrahiert?

  Die Variablen werden ohne den Namen des Dialogs angesprochen.
  Beispiel: Variable `x` des Dialogs `dialog1` heißt `x`.

  Es muß unterschieden werden, ob eine Variable zum aktuellen Dialog, zum Nachfolgedialog oder zur Historie gehört. Dafür werden zugehörige logische Bezeichner (*identifiers*) eingeführt.

`dialog`	bezeichnet eine Variable als zum aktuellen Dialog gehörig
`next`	bezeichnet eine Variable als zum Nachfolgedialog gehörig
`history`	bezeichnet eine Variable als zur Historie (zu bereits bearbeiteten Dialogen) gehörig

Definition 1: Logische Variablenbezeichner

  Beispiel: `dialog.x` ist die Variable `x` des aktuellen Dialogs, `history.y` ist die Variable `y`, die innerhalb des bisherigen Ablaufs der Konversation definiert wurde.

- Was muß bei einer prozeßorientierten (temporalen) Sicht auf Variablen besonders untersucht werden, welche Fragen stellen sich, die beim „zeitlosen“ Variablenbegriff nicht auftreten?

  Bei der prozeßorientierten Sichtweise kann es vorkommen, daß Variablen nicht auf allen Pfaden zum aktuellen Dialog deklariert werden. Zugriffe auf solche Variablen müssen durch Bedingungen (*guards*), die sicherstellen, daß bestimmte Pfade in der Konversation durchlaufen wurden, eingefaßt werden.

Das Zusammenfassen aller Variablen von Dialogen, die bereits abgelaufen sind, führt zum Modell des Trace. Der Trace nimmt den Ablauf der Konversation in Phasen auf.

3.1.1 Die Spur (*trace*)

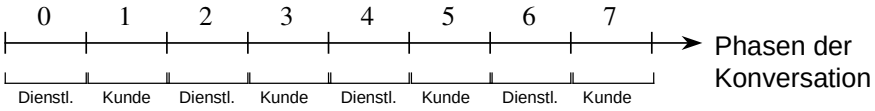
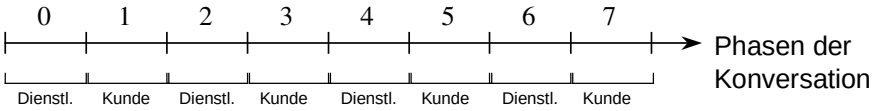


Abbildung 7: Die Spur (*trace*) einer Konversation

Die Spur nimmt alle in der Konversationsspezifikation als Dialoginhalte (*contents*) definierten Variablen auf und ersetzt dadurch die Inhalte der Dialoge. In der Spur werden die Werte der Variablen einer jeden Phase festgehalten, d.h., daß die Spur ein Behälter für alle Variablen der Konversation in ihrem zeitlichen Verlauf ist. Dabei stammen eingeklammerte Variablenbelegungen aus vorherigen Phasen und sind in der aktuellen Phase nicht verändert worden (siehe Abbildung 8).



x :Int	3	(3)	2	(2)	1	(1)	0	(0)
y :Int	0	(0)	1	(1)	2	(2)	3	(3)
Message :String	nil	(nil)	invalid	(invalid)	invalid	(invalid)	valid	(valid)
Name :String	nil	Holm	(Holm)	(Holm)	(Holm)	(Holm)	(Holm)	(Holm)
Password :String	nil	Holm	a83f	SWT	fef8	STS	38e1	(38e1)

Abbildung 8: Die Spur als Variablenbehälter

Die Phasen der Konversation reichen in der bisherigen Modellierung nicht aus, um einen korrekten Zugriff auf Variablen zu gewährleisten, da die vorhergenden und die nachfolgenden Phasen weder über fixe Nummern noch über relativen Offsets zur aktuellen Phasen adressiert werden, sondern durch die logischen Bezeichner dialog, next und history. Die Verwendung der logischen Bezeichner wird an einem Beispiel, in dem nacheinander die Dialoge 1, 2, 3 und 4 bearbeitet werden, verdeutlicht (siehe Abbildung 9).

(actual) dialog	-	Dialog 1	Dialog 1	Dialog 2	Dialog 2	Dialog 3	Dialog 3	Dialog 4
next (dialog)	Dialog 1	-	Dialog 2	-	Dialog 3	-	Dialog 4	-
history	-	-	-	Dialog 1	Dialog 1	Dialog 1,2	Dialog 1,2	Dialog 1,2,3

Abbildung 9: Bedeutung der Bezeichner dialog, next und history in der Spur

Der logische Bezeichner `history` wird von intuitiven Begriff in der Semantik so abgewandelt, daß die Variablenbelegung des aktuellen Dialogs auch in der Historie auftaucht, d.h., daß die Phase des aktuellen Dialogs auch in die Historie aufgenommen wird. Diese Veränderung rührt daher, daß die Variablenbelegung des

aktuellen Dialogs der Kundenphase vom Dienstleister in der vorigen Phase bereits gesetzt worden sein kann. In diesem Fall ist es sinnvoll, diese Belegung in der Historie wiederzufinden.

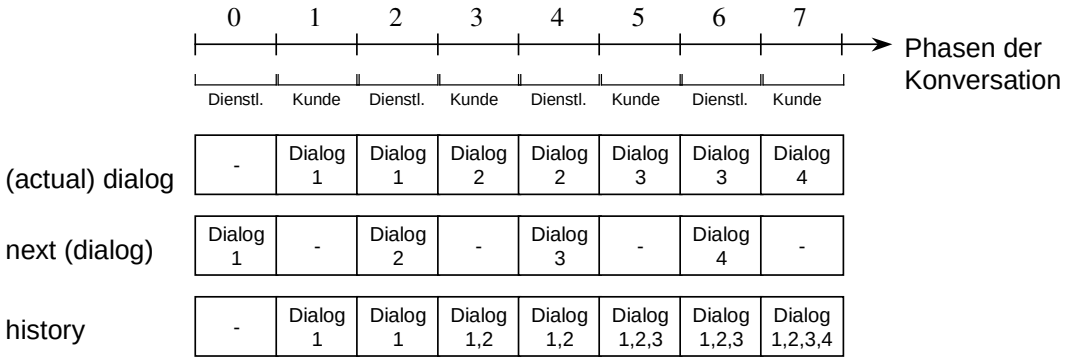


Abbildung 10: Veränderte Semantik des Bezeichers history

Eine weitere Veränderung ist notwendig. In der Dienstleisterphase können die Variablenbelegungen des aktuellen Dialogs und die des Nachfolgedialogs gesetzt werden. Wenn in beiden Dialoge dieselbe Variable gesetzt wird, z.B. weil beide Dialoge dieselbe Spezifikation haben, so ist in späteren Phasen nicht mehr erkennbar, welche Belegung zum hier aktuellen und zum Nachfolgedialog gehört. Aus diesem Grund wird die Dienstleisterphase in zwei Phasen geteilt: Die Phase des aktuellen Dialogs und die Phase des Nachfolgedialogs.

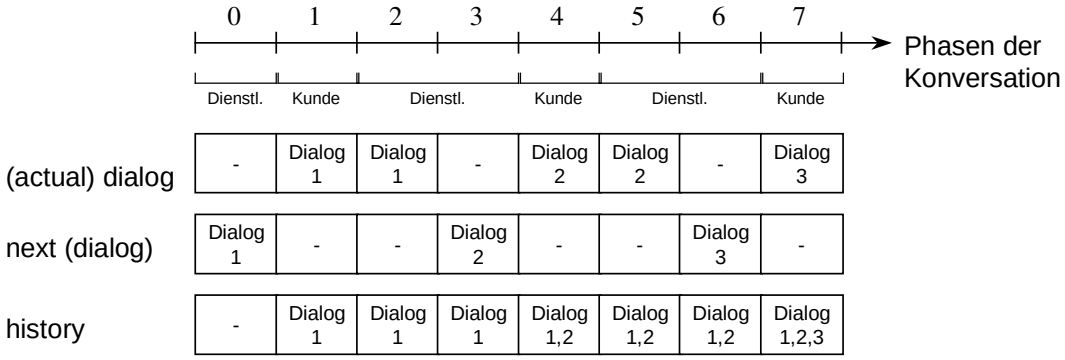


Abbildung 11: Unterteilung der Dienstleisterphase

Es ist wichtig, daß ältere Wertbelegungen der Variablen nicht im nachhinein verändert werden können. Aus diesem Grund führt die Spur eine Schreibschwelle mit, d.h. den Index der niedrigsten Phase, in der schreibender Zugriff erlaubt ist.

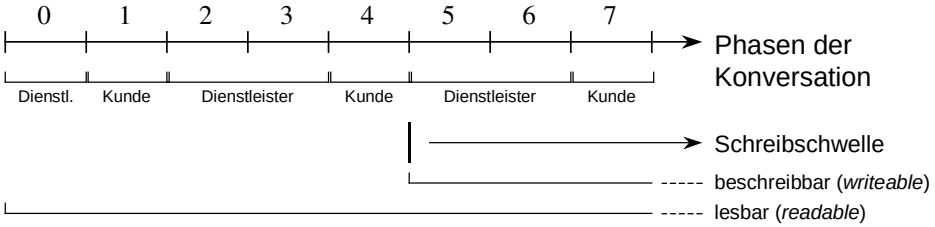


Abbildung 12: Schreibschwelle der Spur

Es folgt die Formalisierung des Zugriff auf die Spur (*trace*) mittels der logischen Bezeichner dialog, next und history. Hierfür wird der Begriff der Spursicht (*traceview*) eingeführt.

3.1.2 Die Spursicht (*traceview*)

Die logische Bezeichner dialog, next und history definieren Sichten auf die Spur der Konversation, die für jede Phase unterschiedlich ist. Gleichzeitig definieren unterschiedliche logische Bezeichner verschiedener Phasen Sichten auf dieselben Variablenbelegungen, teils mit dem Unterschied, daß die Belegungen sondierbar und/oder veränderbar sind (lesender/schreibender Zugriff). Es ist nicht sinnvoll, die logischen Bezeichner als Teil der Spur zu modellieren; vielmehr müssen sie als Sichten (*views*) auf diese verstanden werden.

Eine Spursicht ist eine Sicht auf die Spur einer Konversation. Sie definiert, welche Phasen der Spur sichtbar sind und in welchen davon lesender bzw. schreibender Zugriff auf die Variablen erlaubt ist.

Eine Spursicht gehört immer zu einer Phase einer Konversation. Schreibender Zugriff auf die Spur ist nur in dieser Phase möglich und nur dann, wenn die Schreibschwelle der Spur diese Phase als beschreibbar markiert. Für den lesenden Zugriff definiert die Sicht eine obere und untere Schwelle, in der lesender Zugriff möglich ist.

3.1.3 Semantik der Spursichten

Jede Dienstleisterregel kennt drei, jede Customerrule zwei Sichten: dialog, next und history bzw. dialog und history. Zugriff aus den Regeln auf die Spur ist ausschließlich über die Spursichten möglich. (siehe Abb. 13 und 14).

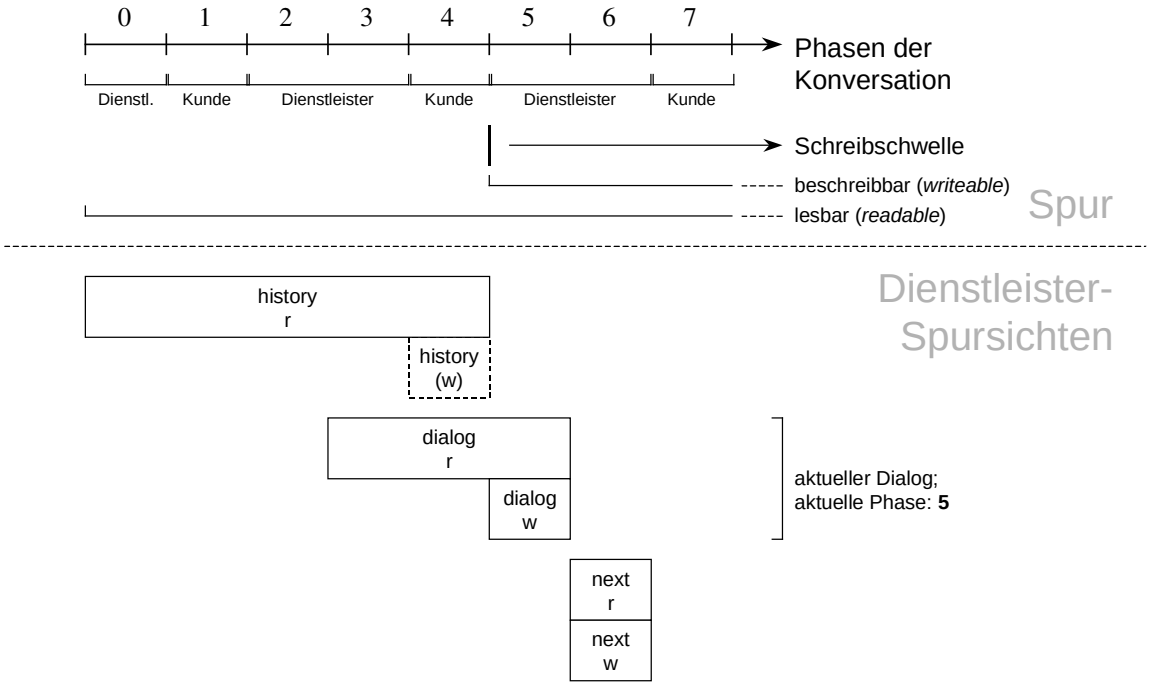


Abbildung 13: Dienstleisterspursichten (*performer traceviews*)

Über die Sicht history können die Werte der vorigen Phasen sondiert werden, dialog bzw. next ermöglichen den lesenden und schreibenden Zugriff der Werte der aktuellen bzw. folgenden Phase. Zu beachten ist einerseits, daß history keinen schreibenden Zugriff erlaubt, da die Schreibschwelle der Spur den verändernden Zugriff auf Werte voriger Phasen verbietet; andererseits, daß der lesende Zugriff des aktuellen Dialogs (dialog) auch die zwei vorhergehenden Phasen umfaßt, da in diesen der Dialog bereits bearbeitet wurde (in der Phase t-2 vom Dienstleister als Nachfolgedialog und in der Phase t-1 vom Kunden als aktueller Dialog).

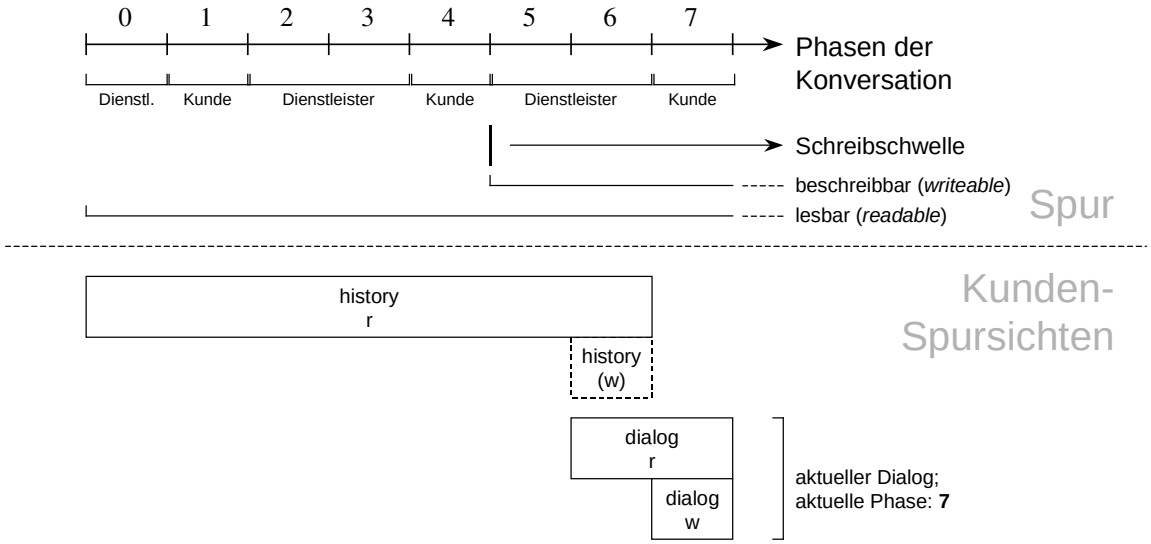


Abbildung 14: Kundenspursichten (*customer traceviews*)

Über die View history kann wie beim Dienstleister auf die vorigen Phasen lesend zugegriffen werden. Der aktuelle Dialog kann in der aktuellen Phase verändert und die Variablenbelegungen dieser und der vorigen Phase können sondiert werden. In der Phase t-1 wurde der Dialog bereits vom Dienstleister als Nachfolgedialog verwendet.

VarName ::=	*Variablenname*
PerformerViewName ::=	**history** | **dialog** | **next**
CustomerViewName ::=	**history** | **dialog**
Lesender Zugriff vom Dienstleister ::=	PerformerViewName "." VarName
Schreibender Zugriff vom Dienstleister ::=	PerformerViewName "." VarName ":=" *Value*
Lesender Zugriff vom Kunden ::=	CustomerViewName "." VarName
Schreibender Zugriff vom Kunden ::=	CustomerViewName "." VarName ":=" *Value*

Definition 2: Formalisierung des Zugriffs auf Variablen

Der Zugriff auf Variablen reicht nicht aus, um die Konsistenz von Spezifikation und Implementation zu gewährleisten. Es fehlen bislang drei wichtige Aspekte:

- Das korrekte Erzeugen von Nachfolgedialogen bzw. Anfragen
- Das korrekte Referenzieren von Variablen, deren Definiertheit nicht garantiert ist
- Der Zugriff auf Variablen über Kurznamen bei Abstraktion der Variablenhierarchie

Die beiden oberen Aspekte fallen unter den Begriff der prozeßorientierten Modellierung, der letzte gehört zur strukturorientierten Modellierung.

3.2 Prozeßorientierte Modellierung

Nach der Definition und Formalisierung des Zugriffs auf Variablen geht es nun um die prozeßorientierte Modellierung. Bei der prozeßorientierten Modellierung geht es um Aspekte, die den zeitlichen Verlauf der Konversation betreffen:

- Das korrekte Erzeugen von Nachfolgedialogen bzw. Anfragen
- Das korrekte Referenzieren von Variablen, deren Definiertheit nicht garantiert ist

3.2.1 Das Erzeugen von Nachfolgedialogen und Abfragen

Das Erzeugen von Nachfolgedialogen und Anfragen wurde schon an einer früheren Stelle beschrieben. Es sei noch einmal kurz wiederholt:

Der Kunde erzeugt am Ende der Bearbeitung des aktuellen Dialogs eine Anfrage, d.h. eine Anweisung, wie die Konversation weiterverlaufen soll. In der folgende Phase erzeugt der Dienstleister unter Berücksichtigung dieser Anfrage den Nachfolgedialog.

DialogName ::=	*Name eines in der Spezifikation genannten Dialogs*
RequestName ::=	*Name einer in der Spezifikation genannten Anfrage (requests)*
NextDialog ::=	**nextDialog** "(" DialogName ")"
NextRequest ::=	**nextRequest** "(" RequestName ")"

Definition 3: Erzeugung von Nachfolgedialogen und Anfragen

Beim Erzeugen von Nachfolgedialogen (im weiteren genannt Erzeugen von Dialogen) und dem Erzeugen von Anfragen muß die Korrektheit der Ausdrücke untersucht werden. In einem ersten Schritt kann die Korrektheit von Dialogen und Anfragen durch die Verkettung der Dialoge untereinander spezifiziert werden.

Ein Dialog *next* ist genau dann ein gültiger Nachfolgedialog des aktuellen Dialogs *dialog*, wenn in der Konversationspezifikation (*CvSpec*) eine Anfrage *req* des Dialogs *dialog* zum Dialog *next* führt.

Eine Anfrage (*request*) *req* ist genau dann am Dialog *dialog* gültig, wenn in der CvSpec eine Anfrage *req* am Dialog *dialog* existiert.

Definition 4: Vorläufige Definition des gültigen Nachfolgedialogs / Anfragen

Diese Definition ist nicht ausreichend, was an einem Beispiel gezeigt wird:

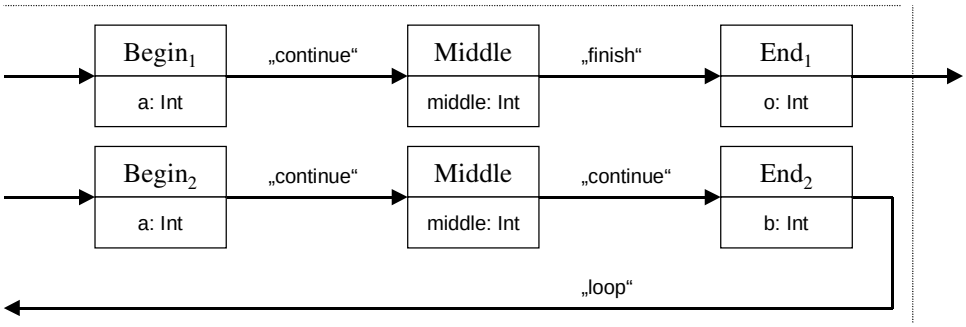


Abbildung 15: Beispiel einer (Teil-)Konversation (1)

Der Dialog „`Middle`“ taucht in der Konversation zweimal auf. Es existiert eine Kundenregel pro Dialog, in der die folgende Anfrage generiert wird. Die zulässige Anfrage kann nun aber nicht allein aus dem aktuellen Dialog

gewonnen werden; die Vorgeschichte des Dialogs ist die einzige Möglichkeit, zu entscheiden, welches die korrekte Anfrage ist, d.h., daß die Vorgeschichte des Dialogs als Bedingung für die Anfrage formalisiert werden kann. Daraus folgt: Der Befehl NextRequest muß um eine Bedingung (*guard*) erweitert werden.

Ein ähnliches Beispiel kann für den Befehl NextDialog gefunden werden. Auch der Befehl NextDialog muß um eine Bedingung erweitert werden.

Der Zugriff auf Variablen über Spursichten ist im prozeßorientierten Kontext ebenfalls nicht ohne Kenntnis der Vorgeschichte des aktuellen Dialogs möglich.

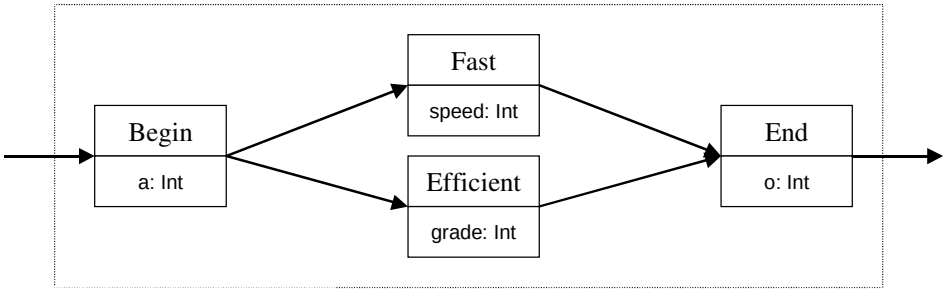


Abbildung 16: Beispiel einer Konversation (2)

Im Dialog „End“ ist der Zugriff auf die Variablen „speed“ und „grade“ nicht möglich, da es vom Ablauf der Konversation abhängt, welcher Dialog zuletzt ausgetauscht wurde. Nur durch Wissen der Vorgeschichte ist ein korrekter Zugriff auf die Variablen der Historie möglich. Es ist leicht zu sehen, daß auch der Zugriff auf die Variablen des Nachfolgedialogs vom Wissen des Ablaufs abhängig ist.

Sowohl das Erzeugen von Nachfolgedialogen / Abfragen als auch der Zugriff auf Variablen kann nur korrekt sein, wenn diese Anweisungen durch Bedingungen, genauer Definiertheitsbedingungen, eingefaßt werden.

Große Erkenntnis 1

In der Arbeit wurden diese Definiertheitsbedingungen so formalisiert, daß sie als boolsche Ausdrücke darüber, ob Dialoge bereits ausgetauscht bzw. Variablen definiert wurden, verstanden werden.

Variable ::=	*Variablenname*
DefiniertheitEinerVariable ::=	**defined** "(" Variable ")"
Dialog ::=	*Dialogname*
AustauschEinesDialogs ::=	**executed** "(" Dialog ")"
BoolscherAusdruck ::=	DefiniertheitEinerVariable | AustauschEinesDialogs | ¬BoolscherAusdruck | BoolscherAusdruck ∧ BoolscherAusdruck | BoolscherAusdruck ∨ BoolscherAusdruck
Definiertheitsbedingung ::=	BoolscherAusdruck

Definition 5: Formalisierung der Definiertheitsbedingung

Es bleibt zu formalisieren, wie die Definiertheitsbedingung beim Erzeugen von Dialogen / Anfragen und beim Zugriff auf Variablen im Code der Regeln notiert wird. Eine Methode wäre es, bei jeder Anweisung, die eine Definiertheitsbedingung benötigt, die Anweisung durch die Bedingung einzufassen. Da aber zumeist mehrere Anweisungen unter derselben Bedingung stehen, erscheint es sinnvoll, die Definiertheitsbedingung von den Anweisungen zu trennen und Bedingungen für Anweisungsblöcke zu formalisieren. Dazu wird das Konstrukt whex (*when exists*) eingeführt.

Anweisungsblock ::=	*Anweisungsblock der Sprache*
Whex ::=	**whex** "(" Definiertheitsbedingung ")" **do** Anweisungsblock **else** Anweisungsblock **end**

Definition 6: Syntax des whex-Konstrukts

Die Semantik des whex-Konstrukts ist folgende: Wenn die Bedingung zu wahr evaluiert, führe den Anweisungsblock nach dem do aus, andernfalls führe den Block nach dem else aus. Die Bedingung des whex-Statements gilt für alle Anweisungen innerhalb des do Blocks. Die negierte Bedingung gilt für alle Anweisungen des else Blocks. Whex-Konstrukte können, wie die Programmiersprachkonstrukte if, loop und while, geschachtelt werden. Die Bedingungen von geschachtelten whex-Konstrukten werden als Konjunktion der Bedingungen verstanden (durch logisches Und verknüpft). D.h., daß für eine Anweisung in einem mehrfach geschachtelten whex die Bedingungen aller whex-Konstrukte gilt.

Die Formalisierung ist noch nicht vollständig, denn es fehlt eine Bedingung für den Zugriff auf Variablen des Nachfolgedialogs.

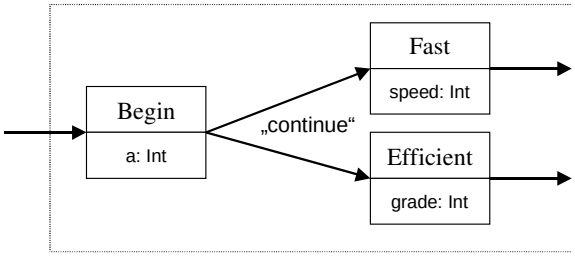


Abbildung 17: Beispiel einer (Teil-)Konversation (3)

In der Dienstleisterregel, die den Nachfolgedialog von „`Begin`“ erzeugt, werden die Variablen des Nachfolgedialogs über die Sicht next referenziert. Die Variablen „`speed`“ bzw. „`grade`“ können nicht korrekt referenziert werden, solange nicht bekannt ist, welcher Nachfolgedialog erzeugt wurde, d.h., daß der Zugriff auf Variablen vom Nachfolgedialog abhängig ist. Dies ist eine andere Form von Bedingung, da der Nachfolgedialog nicht als Bedingung über den bisherigen Ablauf der Konversation verstanden werden kann, d.h., daß diese Bedingung nicht auf den Begriff der Definiertheitsbedingung zurückgeführt werden kann. Die neue Bedingung wird formalisiert als Erzeugungsbedingung.

ErzeugungsBedingung ::=	**created** "(" Dialog ")"

Definition 7: Erzeugungsbedingung

Da diese Bedingung immer mit der NextDialog-Anweisung zusammenfällt, ist es sinnvoll, die NextDialog-Anweisung als eigenes Konstrukt zu formalisieren.

NextDialog ::=	**nextDialog** "(" DialogName ")" **do** Anweisungsblock **end**

Definition 8: Syntax des NextDialog-Konstrukts

Die Semantik des NextDialog-Konstrukts ist folgende: Erzeuge als Nachfolgedialog den Dialog `DialogName` und führe alle Anweisungen im Anweisungsblock aus. Diese Anweisungen haben als Bedingung die das NextDialog einfassende Bedingung $\wedge$ ErzeugungsBedingung des NextDialog-Konstrukts. Um eine einheitliche Bedingung zu erhalten, werden die Definiertheits- und Erzeugungsbedingungen zu einer Bedingung zusammengefaßt.

Bedingung ::=	ErzeugungsBedingung | Definiertheitsbedingung | ErzeugungsBedingung $\wedge$ Definiertheitsbedingung

Definition 9: Bedingung einer Anweisung

Da keine Anweisung innerhalb einer Kundenregel abhängig von der erzeugten Anfrage ist, wird das NextRequest-Konstrukt nicht verändert.

NextRequest ::=	**nextRequest** "(" RequestName ")"

Definition 10: Syntax des NextRequest-Konstrukts

Es gilt nun, ein Modell zu finden, mit dem die Bedingungen geprüft werden können. Die beiden weiter hinten in diesem Kapitel vorgestellten Evaluatoren, der Prozeß- und der Strukturevaluator, prüfen genau diese Bedingungen und arbeiten dabei auf der SET-eigenen Repräsentation der Konversation, genannt Modellrepräsentation.

3.2.2 Die Modellrepräsentation

Die Bedingungen, welche Anweisungen in den Regeln einfassen, zu evaluieren, ist Aufgabe der in diesem Kapitel beschriebenen Evaluatoren. Vor der Darstellung der Evaluatoren wird zunächst die Repräsentation der Konversationspezifikation beschrieben, auf der diese arbeiten.

Das Modell der Konversation, wie sie in der Konversationsspezifikation definiert wird, ist aus mehreren Gründen nicht geeignet, als Modell für die Evaluatoren zu dienen:

- Die Konversationsspezifikation erlaubt es nicht, auf geeignete Weise mehrere Dialoge, die derselben Dialogspezifikation folgen, d.h. mehrere Instanzen einer Dialogspezifikation, zu unterscheiden,
- die Bedingungen umfassen Variablen und Dialoge gleichermaßen. Es ist sinnvoll, diese Mengen zu einem Begriff zusammenzufassen, um ein allgemeineres Modell zu erhalten.

Unter Beachtung dieser Gründe wird das Modell der Konversation in eine geeignete Repräsentation überführt, die Modellrepräsentation.
In der Modellrepräsentation werden verschiedene Dialoge einer Dialogspezifikation als Dialogknoten (*nodes*) unterschieden. Ein Node ist ein Exemplar eines Dialogs.
Die Modellrepräsentation ist definiert als gerichteter Graph, dessen Knoten die Dialogknoten (*nodes*) und dessen Kanten die Anfragen (*requests*) sind. Es existiert ein Startknoten (*start node*), welcher den initialen Dialog repräsentiert. Die Endknoten (*end nodes*) sind diejenigen Knoten, welche Dialoge repräsentieren, an denen die aus der Konversation hinausführende Anfrage (*final request*) definiert ist. Um die in die Konversation hinein- und herausführenden Anfragen als Kanten repräsentieren zu können, werden zwei zusätzliche inhaltslose Dialoge eingeführt: Pre- und PostDialog. Sie werden in der Modellrepräsentation durch eigene Knoten, genannt Pre- und Postnode, repräsentiert. Anfragen, die zu mehr als einem Dialog führen, werden durch mehrere Kanten repräsentiert.

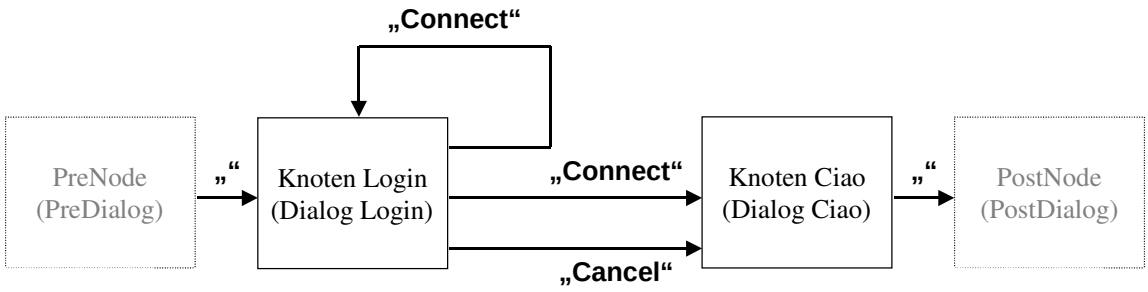


Abbildung 18: Modellrepräsentation der Beispielkonversation aus Kapitel 2.1 (Abb. 4)

Der Prozeßevaluator arbeitet auf der Modellrepräsentation und überprüft, ob in einer Konversation eine Variable definiert bzw. ein Dialog ausgetauscht wurde. Die Menge der zu prüfenden Objekte wird als Menge der VDN-Objekte (Variablen/Dialoge/Nodes) bezeichnet.

VDNObjekte (VDN) ::= Variablen ∪ Dialoge ∪ Knoten

Definition 11: VDNObjekt

Es muß als nächster Schritt die Semantik der Begriffe Definiertheit einer Variablen bzw. Austauschen eines Dialogs geklärt werden. Die beiden Begriffe werden zunächst zu einem Begriff vereint, dem Begriff der Sichtbarkeit (*scope*) eines VDNObjekts.

always(o)	ein VDNObjekt o ist immer sichtbar
possibly(o)	ein VDNObjekt o kann sichtbar sein
never(o)	ein VDNObjekt o ist nie sichtbar

Eine VDNObjekt einer Konversation hat zunächst einen Scope an einem Knoten. Die Sichtbarkeit eines VDNObjekts v an einem Knoten n definiert, ob v auf allen Pfaden zum Knoten n definiert wurde. Im Beispiel in Abbildung 19 ist die Sichtbarkeit des Dialogs „`Login`“ am Knoten „`Ciao`“ always, d.h., daß der Dialog „`Login`“ auf allen Pfaden zum Knoten „`Ciao`“ ausgetauscht wurde. Zu beachten ist weiterhin, daß keine Unterscheidung der Sichtbarkeit zwischen Dialog,Knoten und Variablen mehr gemacht wird.

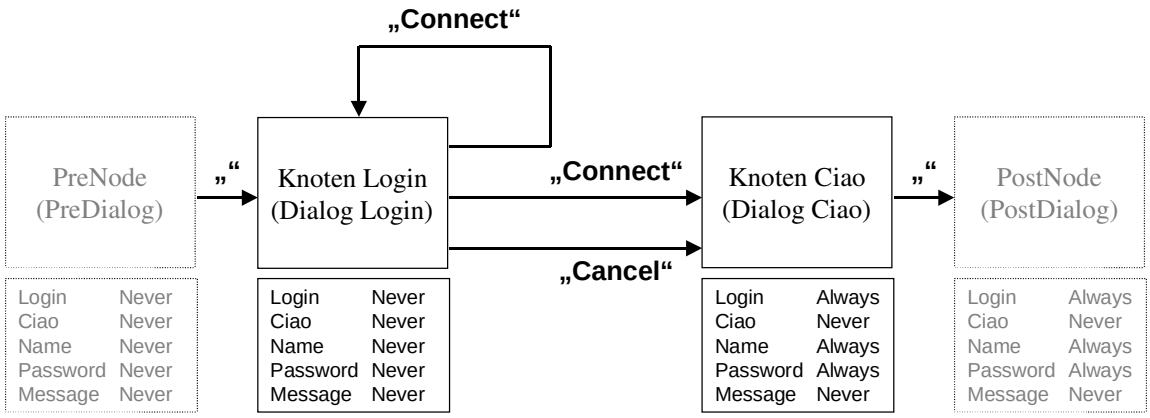


Abbildung 19: Sichtbarkeit an einem Knoten

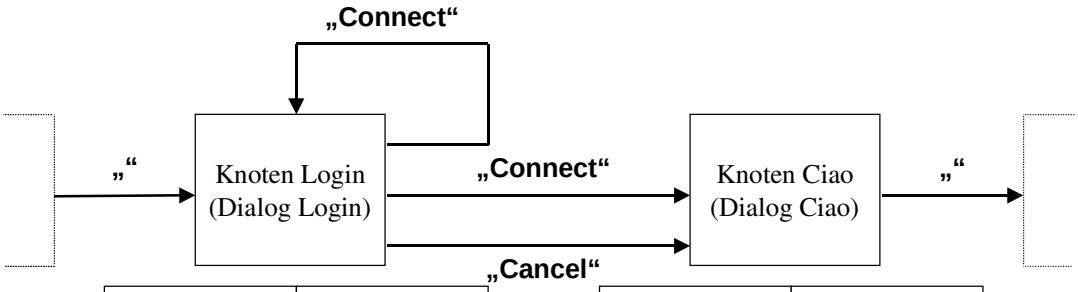
Dieses Modell ist ermöglicht bereits die Evaluation der Sichtbarkeit an einem Knoten. Die Bedingungen der whex- / NextDialog- und NextRequest-Konstrukte können jedoch nicht auf die Sichtbarkeit eines VDNObjekts an einem Knoten reduziert werden. Im Abschnitt Prozeßevaluator wird gezeigt, daß jede Bedingung in der Modellrepräsentation evaluiert werden kann, wenn die Sichtbarkeit von VDNObjekten zwischen je zwei Knoten definiert ist.

always(o,n_i,n_j)	ein VDNObjekt o ist auf allen Pfaden von Knoten i zu Knoten j immer sichtbar
possibly(o,n_i,n_j)	ein VDNObjekt o ist auf einigen Pfaden von Knoten i zu Knoten j sichtbar
never(o,n_i,n_i)	ein VDNObjekt o ist auf keinem Pfad von Knoten i zu Knoten j sichtbar

Definition 12: Sichtbarkeit (*scope*)

Jedem Triple (VDNObjekt,Knoten,Knoten) wird eine Sichtbarkeit zugewiesen, d.h. die Menge VDN x N x N wird in die Menge S abgebildet. Dabei definiert die Abbildung $f(vdn,n_1,n_{end}) \rightarrow s$ die Sichtbarkeit des VDNObjekts auf allen Pfaden von n_1 nach n_{end}.

Der Anschaulichkeit halber wird diese Abbildung in Tabellen dargestellt. Jedem Knoten n_j werden i Tabellen (i = 0,1,...,n) mit Paaren (vdn,s) zugeordnet, welche die Sichtbarkeit des Objekts vdn vom Knoten n_i zum Knoten n_j enthalten.



v. Knoten PreNode		vom Knoten Login	
Login	Possibly	Login	Always
Ciao	Never	Ciao	Never
Name	Possibly	Name	Always
Password	Possibly	Password	Always
Message	Never	Message	Never

vom Knoten Ciao		v. Knoten PostNode	
Login	Never	Login	Never
Ciao	Never	Ciao	Never
Name	Never	Name	Never
Password	Never	Password	Never
Message	Never	Message	Never

v. Knoten PreNode		vom Knoten Login	
Login	Always	Login	Always
Ciao	Never	Ciao	Never
Name	Always	Name	Always
Password	Always	Password	Always
Message	Never	Message	Never

vom Knoten Ciao		v. Knoten PostNode	
Login	Never	Login	Never
Ciao	Never	Ciao	Never
Name	Never	Name	Never
Password	Never	Password	Never
Message	Never	Message	Never

Abbildung 20: Scope zwischen je zwei Nodes

3.2.3 Das Prozeßmodell

Das Prozeßmodell definiert den Algorithmus zur Berechnung der Sichtbarkeit für alle Tripel (vdn,n_i,n_j). Für das Prozeßmodell muß die Sichtbarkeit um einen temporären Wert erweitert werden: undefined.

always(o,n_i,n_j)	ein VDNObjekt o ist auf allen Pfaden von Knoten i zu Knoten j immer sichtbar
possibly(o,n_i,n_j)	ein VDNObjekt o ist auf einigen Pfaden von Knoten i zu Knoten j sichtbar
never(o,n_i,n_j)	ein VDNObjekt o ist auf keinem Pfad von Knoten i zu Knoten j sichtbar
undefined(o,n_i,n_j)	die Sichtbarkeit eines VDNObjekts o zwischen Knoten i und Knoten j ist nicht definiert

Definition 13: Erweiterung der Sichtbarkeit für das Prozessmodell

Algorithmus zur Berechnung des Scopes von VDNObjekten zwischen je zwei Knoten

Für alle Knoten n_j führe aus
Setze die Sichtbarkeit s(vdn,n_i,n_i) für alle n_i auf *undefined*
Trage den Knoten n_j in die Liste der noch zu bearbeitenden Knoten ein

Wiederhole, solange die Liste der noch zu bearbeitenden Knoten nicht leer ist
Für den ersten Knoten n_{next} in der Liste der noch zu bearbeitenden Knoten führe aus
Für allen Kanten r_k, die zum Knoten n_{next} führen, führe aus
Erzeuge die Sichtbarkeit aller VDNObjekte vdn an der Kante r_k:
s(r_k,vdn) = *always*, wenn vdn ein Attribut des Dialogs des Knoten n_{next} ist, oder wenn vdn der Dialog des Node n_{next} ist
= s(vdn,n_{next},n_j) sonst.
Am Knoten n_{first}, von dem die Kante r_k ausgeht, verknüpfe für alle VDNObjekte vdn die Sichtbarkeit s(vdn,nfirst,nj) mit s(r_k,vdn) durch Und-Verknüpfung.
Hat sich die Sichtbarkeit s(vdn,n_{first},n_j) für ein vdn geändert, trage den Knoten n_{first} in die Liste der noch zu bearbeitenden Knoten ein.
Entferne den bearbeiteten Knoten von der Liste der noch zu bearbeitenden Knoten
Friere die Sichtbarkeit jedes VDNObjekts an jedem Knoten ein

Das Verknüpfen der Sichtbarkeit ist definiert als das Anwenden der Und-Verknüpfung auf die beiden Sichtbarkeiten. Das Einfrieren der Sichtbarkeit ist definiert als das Umwandeln von undefined in never.

Die Verknüpfungen ist über folgende Tabellen definiert:

AND	Always	Possibly	Never	Undefined
Always	Always	Possibly	Possibly	Always
Possibly	Possibly	Possibly	Possibly	Possibly
Never	Possibly	Possibly	Never	Never
Undefined	Always	Possibly	Never	Undefined

Tabelle 1: Und-Verknüpfung auf Sichtbarkeit von VDNObjekten

OR	Always	Possibly	Never	Undefined
Always	Always	Always	Always	Undefined
Possibly	Always	Possibly	Possibly	Undefined
Never	Always	Possibly	Never	Undefined
Undefined	Undefined	Undefined	Undefined	Undefined

Tabelle 2: Oder-Verknüpfung auf Sichtbarkeit von VDNObjekten

S	Freeze(s)
Always	Always
Possibly	Possibly
Never	Never
Undefined	Never

Definition 14: Einfrieren von Sichtbarkeit

Beschreibung des Algorithmus

Der Algorithmus basiert darauf, die Sichtbarkeit $s(vdn,n_i,n_j)$ für jeden (Ziel-)Knoten n_j separat zu berechnen. Er sucht rekursiv alle Pfade zum Knoten n_j ab. Dazu werden die Kanten rekursiv entgegen ihrer Richtung verfolgt.
An jeder Kante r_k gilt, daß die Sichtbarkeit eines jeden VDNObjekts an r_k die Sichtbarkeit des Objekte am Knoten n_{next} ist, zu dem die Kante r_k führt. Ausnahmen davon sind die Variablen, die Attribute des Dialogs des Knoten n_{next} sind und der Dialog des Knoten n_{next} selbst: Sie sind auf jeden Fall an der Kante r_k sichtbar (Sichtbarkeit: always).
Die Sichtbarkeit an der Kante r_k wird benutzt, um die Sichtbarkeit des Knoten n_{first} zu berechnen. Der Node n_{first} ist derjenige Knoten, von dem die Kante r_k abgeht. Die Sichtbarkeit aller VDNObjekte an den Knoten n_i mit $i \neq j$ ist durch Initialisierung zunächst always. Diese Sichtbarkeit kann im Verlauf des Algorithmus durch die Sichtbarkeit an Kanten r_k nur eingeschränkt werden.
Die Sichtbarkeit des Knoten n_{first} wird durch Und-Verknüpfung mit der Sichtbarkeit an der Kante r_k verknüpft. Hat sich die Sichtbarkeit am Knoten n_{first} verändert, so muß diese Veränderung durch alle Kanten, die an n_{first} enden, propagiert werden. Das geschieht durch Eintragen des Knoten in die Liste der noch zu bearbeitenden Knoten.
Der Algorithmus terminiert, wenn keine Knoten mehr in der Liste der zu bearbeitenden Knoten stehen.
Der Grund, weshalb always nicht als initialer Scope an den Knoten stehen kann, ist, daß es nicht unterscheidbar wäre, ob die Sichtbarkeit eines VDNObjektes vom Knoten n_i zum Knoten n_j always ist, weil es keinen Pfad gibt, auf dem das VDNObjekt nicht sichtbar ist, oder ob die Sichtbarkeit always ist, weil es keinen Pfad vom Knoten n_i zum Knoten n_j gibt. Aus diesem Grund wurde undefined eingeführt.
Da der Prozeßevaluator nicht auf undefined arbeitet, wird am Ende die Sichtbarkeit eingefroren, d.h. undefined wird durch never ersetzt.

Formulierung negativer Bedingungen

In Vorwegnahme der Beschreibung des Prozeßevaluators ist es erforderlich, in der Modellrepräsentation die Sichtbarkeit von VDNObjekten unter Weglassen einer Menge N' von Knoten zu bestimmen. In diesem Fall überführt das Prozeßmodell die Modellrepräsentation M in eine neue Modellrepräsentation M'. Aus M' werden die Knoten N' und alle Kanten, die mit Knoten aus N' verbunden waren, entfernt. Der Algorithmus der Prozessmodells wird über M' ausgeführt. Der Prozeßevaluator arbeitet nachfolgend auf M'.

3.2.4 Der Prozeßevaluator

Der Prozeßevaluator überprüft die Zulässigkeit von Anweisungen, d.h. den Zugriff aus Regeln auf Variablen über Sichten und das Erzeugen von Dialogen und Anfragen unter einer Bedingung. Dazu wird die Bedingung in Aussagen über den Sichtbarkeit von VDNObjekten überführt. Die Aussagen können in der Modellrepräsentation evaluiert werden.

Zunächst müssen die zu überprüfenden Anweisungen und die Bedingungen formalisiert werden. Die Formalisierung der Anweisungen und der Bedingungen ist im Gegensatz zur Formalisierung der Sprachkonstrukte feiner.

Es werden zuerst die Bedingungen und die Anweisungen formalisiert. Anschließend wird die Semantik der Sprachkonstrukte formalisiert. Nach den Formalisierungen wird das Deduktionsgerüst vorgestellt, mit dem die Bedingungen reduziert werden.

3.2.4.1 Formalisierung der Bedingung von Anweisungen als Umgebung

In der Beschreibung der Sprachkonstrukte wurde bereits dargelegt, daß jede Anweisung einer Dienstleister- oder Kundenregel in eine Bedingung eingefaßt ist, die die bereits abgelaufene Konversation konkretisiert. Das heißt, daß jede Anweisung in einem bestimmten Kontext von Variablen und Dialogen abläuft. Dieser Kontext wird als Umgebung (*environment*) env formalisiert. Eine Umgebung definiert einen Kontextraum, in dem Anweisungen stehen können. Jede Anweisung anw steht demnach in einer Umgebung $env(_{anw})$.
Die Bedingung einer Dienstleisteranweisung besteht aus der Definiertheitsbedingung und einer Erzeugungsbedingung. Die Bedingung einer Kundenanweisung umfaßt hingegen nur die Definiertheitsbedingung. Die unterschiedlichen Bedingungen werden als unterschiedliche Umgebungen formalisiert.

Vdef(v) ::=	*Variable definiert*
Ddef ::=	*Dialog definiert*
Ndef ::=	*Node definiert*
VDNdef ::=	Vdef | Ddef | Ndef
Rdef ::=	*Anfrage (request) definiert*
Definiertheitsumgebung (dEnv) ::=	VDNdef | ¬dEnv | dEnv ∧ dEnv | dEnv ∨ dEnv | ε
Erzeugungsumgebung (eEnv) ::=	Ddef | ε
Dienstleisterumgebung (pEnv) ::=	(dEnv, eEnv)
Kundenumgebung (cEnv) ::=	dEnv

Definition 15: Formalisierung der Umgebung

Beispiel: pEnv(Vdef(a) ∧ Ddef(b) ∧ ¬Ddef(c), Ddef(e)) bedeutet, daß in der Umgebung die Variable a definiert, der Dialog b ausgetauscht, der Dialog c nicht ausgetauscht und der Dialog e als Nachfolgedialog erzeugt wurde.

3.2.4.2 Formalisierung von Anweisungen

Eine zu prüfende Dienstleisteranweisung ist ein Variablenzugriff oder das Erzeugen des Nachfolgedialogs. Eine zu prüfende Kundenanweisung ist ein Variablenzugriff oder das Generieren der Anfrage. Ein Variablenzugriff besteht aus der benutzten Sicht (*traceview*), der Variable der Spur und einem Wert, der angibt, ob es sich um einen lesenden oder schreibenden Zugriff handelt. Das Erzeugen eines Nachfolgedialogs umfaßt den Nachfolgedialog, das Generieren einer Anfrage diese Anfrage. Die Formalisierung von Dienstleisteranweisungen umfaßt den aktuellen Dialog und die Anfrage (*request*), die der Kundenanweisungen den aktuellen Dialog. Wie oben beschrieben, hat jede Kunde- und Dienstleisteranweisung eine Umgebung (*environment*). Damit ist die Formalisierung der Anweisungen vollständig.

V ::=	*Variable*
D ::=	*Dialog*
R ::=	*Request*
aktDialog ::=	*aktueller Dialog der Regel*
aktRequest ::=	*aktuelle Anfrage (request) der (Dienstleister-)Regel*
RW ::=	**read | write**
pAccess ::=	(View,V,RW,pEnv,aktDialog,aktRequest)
cAccess ::=	(View,V,RW,cEnv,aktDialog)
pDialog ::=	(D,pEnv,aktDialog,aktRequest)
cRequest ::=	(R,cEnv,aktDialog)
pAnw ::=	pAccess | pDialog
cAnw ::=	cAccess | cRequest

Definition 16: Formalisierung der Anweisungen

Beispiel:
pAnw(history,V(„`Password`“),read,pEnv(Vdef(„`Name`“) ∧ Ddef(„`Login`“)),D(„`Ciao`“),R(„`Connect`“)) bedeutet, daß in der Dienstleisterregel mit aktuellem Dialog „`Ciao`“ und Anfrage (*request*) „`Connect`“ eine Anweisung geprüft wird, welche die Variable „`Password`“ über die Sicht history liest, unter der Bedingung, daß die Variable „`Name`“ definiert und der Dialog „`Login`“ ausgetauscht wurde.

3.2.4.3 Formalisierung der Konstrukte

Das whex-Konstrukt definiert eine Definiertheitsumgebung dEnv und zwei Anweisungsblöcke blk+, blk-. Das Konstrukt definiert aus der äußeren Umgebung und der Umgebung dEnv die neuen Umgebungen env+ und env-.

whex(dEnv, blk+, blk-) definiert zwei neue Umgebungen env+ und env- mit Umgebung (blk+) = env+ und Umgebung (blk-) = env-.

Umgebung(whex) = (dEnv(whex), eEnv (whex))

env+ ::=	(dEnv(whex) ∧ dEnv, eEnv(whex))
env- ::=	(dEnv(whex) ∧ ¬dEnv, eEnv(whex))

Definition 17: Formalisierung des whex-Konstrukts

Das NextDialog-Konstrukt definiert eine Erzeugungsumgebung eEnv und einen Anweisungsblock blk. Das Konstrukt definiert aus der äußeren Umgebung und der Umgebung eEnv eine neue Umgebung env' für den Anweisungsblock, wenn die äußere Umgebung noch kein eEnv definiert.

nextDialog(eEnv, blk) definiert eine neue Umgebung env' mit Environment(blk) = env.

Umgebung(NextDialog) = (dEnv(NextDialog), eEnv(NextDialog))

env ::=	(dEnv(NextDialog), eEnv),	wenn eEnvironment(NextDialog) = ε,
	ERROR	sonst

Definition 18: Formalisierung des NextDialog-Konstrukts

Das NextRequest-Konstrukt definiert keine Umgebung, sondern eine Anfragedefinition.

nextRequest(Rdef)

Definition 19: Formalisierung des NextRequest-Konstrukts

Anmerkung: Die Formalisierung der NextDialog-Anweisung ist unabhängig von der Semantik des Konstrukts, den Dialog zu erzeugen.

3.2.4.4 Reduktion der Anweisungen auf umgebungsrelevante Teile

Beim Prüfen einer Anweisung wird diese zunächst um Teile, die nicht von der Umgebung abhängig sind, reduziert. Darunter fällt der lesende und schreibende Zugriff und die Sichten.

- Der lesende und schreibende Zugriff auf Variablen des aktuellen Dialogs ist immer erlaubt.
  pAccess(dialog,V,RW,pEnv,aktDialog,aktRequest) ⊨ Always
  cAccess(dialog,V,RW,cEnv,aktDialog) ⊨ Always

- Der schreibende Zugriff auf Variablen der Historie ist nicht erlaubt.
  pAccess(history,V,write,pEnv,aktDialog,aktRequest) ⊨ Never
  cAccess(history,V,write,pEnv,aktDialog) ⊨ Never

Durch weitere Reduktion ergeben sich vier Fälle:

- Der lesende Zugriff auf Variablen der Historie. Er ist in Kunden- und Dienstleisterregeln gleich, da er nicht vom eEnv abhängig ist.
  pAccess(history,V,read,pEnv,aktDialog,aktRequest)
  cAccess(history,V,read,pEnv,aktDialog)

- Das Erzeugen des Dialogs.
  pDialog(D,pEnv,aktDialog,aktRequest)

- Das Generieren des Request.
  cRequest(R,cEnv,aktDialog)

- Der Zugriff auf Variablen über die Sicht next in Dienstleisterregeln.
  pAccess(next,V,RW,pEnv,aktDialog,aktRequest)

Nur der letzte Fall braucht die Erzeugungsumgebung eEnv zur Evaluation. Es ist daher sinnvoll, dEnv von eEnv zu trennen und einen allgemeinen Umgebungsbegriff zu schaffen.

```
Env ::= dEnv
Env(cEnv) ::= dEnv(cEnv)
Env(pEnv) ::= dEnv(pEnv)
```

Definition 20: Umgebung als dEnv; eEnv getrennt

Die Umgebung kann als Bedingung für alle zulässigen Pfade durch die Konversation aufgefaßt werden. Die Semantik der Umgebung wird übertragen auf die Modellrepräsentation und kann formalisiert werden als:

Die Anweisung anw kann in der Umgebung env am Dialog aktDialog ausgeführt werden, wenn
∀ Pfade p vom initialen Dialog zum aktDialog unter env die semantische Bedingung sb(anw) der Anweisung erfüllt wird.

(sb(anw),aktDialog,env) ⊨ true

Definition 21: Semantische Bedingung

Die semantische Bedingung der Anweisung definiert, was für alle Pfade zwischen initialem und aktuellem Dialog gelten muß, damit die Anweisung ausgeführt werden darf. Wir unterscheiden die vier oben genannten Fälle.

Lesender Zugriff auf eine Variable v der Sicht history:
∀ p vom initialen Dialog zum aktDialog unter env:
$s(v,initialerDialog,aktDialog_j)$ = always

Das Erzeugen eines Dialogs d_{next}:
∀ p vom initialen Dialog zum aktDialog unter env:
$\exists\ d_{new}\ \exists\ r \mid r = (aktDialog_j,d_{new}) \wedge Name(r) = Name(aktRequest)$

Das Generieren einer Anfrage (*request*) r_{next}.
$\forall$ p vom initialen Dialog zum aktDialog unter env:
$\exists\ d_{new}\ \exists\ r_{next} = (aktDialog,d_{new})$

Zugriff auf eine Variable v über die Sicht next:
$\forall$ p vom initialen Dialog zum aktDialog unter env:
$\forall\ d_{new} : \exists\ r \mid r(aktDialog,d_{new}) \wedge Name(r) = Name(aktRequest) :$
$s(aktDialog,d_{new},v) =$ always

Diese formal nicht korrekte Definition der semantischen Bedingungen muß korrekt formalisiert werden. Da die Sichtbarkeit nur zwischen Knoten definiert ist und die Modellrepräsentation nur mit Knoten arbeitet, müssen zunächst die semantischen Bedingungen für Knoten definiert werden. Der Begriff des initialen Dialogs ist ebenfalls nicht in der Modellrepräsentation definiert, er wird ersetzt durch den PreNode.

Lesender Zugriff auf eine Variable v der Sicht history:
$\forall\ n_{aktDialog} \mid Dialog(n_{aktDialog}) = aktDialog$
$\forall$ p vom PreNode zum Knoten $n_{aktDialog}$ unter env:
$s(PreNode,n_{aktDialog},v) =$ always

Das Erzeugen eines Dialogs d_{next}:
$\forall\ n_{aktDialog} \mid Dialog(n_{aktDialog}) = aktDialog$
$\forall$ p vom PreNode zum Knoten $n_{aktDialog}$ unter env:
$\exists\ n_{new} \mid Dialog(n_{new}) = d_{next}: \exists\ r \mid r = (n_{aktDialog},n_{new}) \wedge Name(r) = Name(aktRequest)$

Das Generieren einer Anfrage (*request*) r_{next}.
$\forall\ n_{aktDialog} \mid Dialog(n_{aktDialog}) = aktDialog$
$\forall$ p vom PreNode zum Knoten $n_{aktDialog}$ unter env:
$\exists\ n_{new}\ \exists\ r_{next} = (n_j,n_{new})$

Zugriff auf eine Variable v über die Sicht next:
$\forall\ n_{aktDialog} \mid Dialog(n_j) = aktDialog$
$\forall$ p vom PreNode zum Knoten $n_{aktDialog}$ unter env:
$\forall\ n_{new} : \exists\ r \mid r(n_j,n_{new}) \wedge Name(r) = Name(aktRequest) :$
$s(n_{aktDialog},n_{new},v) =$ always

3.2.4.5 Reduktion des Environments auf Pfade in der Modelrep

Die Umgebung env ist definiert als boolescher Ausdruck über die Definiertheit von VDNObjekten. Der aktuelle Dialog sei d genannt. Die Erfüllung der semantischen Bedingung einer Anweisung sb(anw) in Dialog d unter Umgebung env wird geprüft.

$(sb(anw),d,env) \models true$

Dazu wird zunächst die Umgebung auf Aussagen der Modellrepräsentation zurückgeführt und anschließend werden über diese Aussagen die semantischen Bedingungen der vier Fälle beschrieben.

- Sei env_{DNF} die Umgebung env in disjunktiver Normalform (nicht minimiert).

  $(sb(anw),d,env_{DNF}) \models true \leftrightarrow (sb(anw),d,env) \models true$
  sb(anw) ist erfüllt in d unter env genau dann, wenn sb(anw) erfüllt ist in d unter env in disjunktiver Normalform.

- Sei EnvAnd die Menge aller maximalen Und-SubTerme von env_{DNF}, die nur durch $\vee$ (oder) verknüpft sind.

  $\forall$envAnd : envAnd ist Teilbedingung von env_{DNF} :
  (sb(anw),d,envAnd) $\models$ true $\leftrightarrow$ (sb(anw),d,env_{DNF}) $\models$ true
  sb(anw) ist erfüllt in d unter env_{DNF} genau dann, wenn sb(anw) erfüllt ist in d unter jeder der disjunkten Teilbedingungen von env_{DNF}.

- Jede envAnd läßt sich schreiben als Konjunktion von Literalen und negierten Literalen.

  envAnd = x1 $\wedge$ x2 $\wedge$ x3 $\wedge$ ... $\wedge$ $\neg$xn1$\wedge$ $\neg$xn2 $\wedge$ $\neg$xn3 $\wedge$ ...

  Diese Aussage läßt sich in zwei Teilaussagen teilen:

  envAndPlus = x1 $\wedge$ x2 $\wedge$ x3 $\wedge$ ...
  envAndMinus = $\neg$xn1 $\wedge$ $\neg$xn2 $\wedge$ $\neg$xn3 $\wedge$ ...

  mit

  envAnd = envAndPlus $\wedge$ envAndMinus

Nach der Teilung der Aussagen envAnd werden zwei Ziele verfolgt:

1. Reduktion der negierten elementaren Aussagen über die Definiertheit von VDNObjekten (envAndMinus) auf negierte elementare Aussagen über die Definiertheit von Knoten. Die erhaltenen Knoten bilden die Menge der Knoten N_{ignore}. Diese werden vor Evaluation des Prozessmodells aus der Modellrepräsentation entfernt.

2. Reduktion der positiven elementaren Aussagen über die Definiertheit von VDNObjekten auf elementare Aussagen über die Definiertheit von Knoten. Die erhaltenen Knoten müssen im Prozeß durchlaufen (passiert) werden, um definiert zu sein, und sind deshalb Punkte im Weg von PreNode zu allen Knoten n_d. Damit kann der Weg von PreNode über andere Knoten zu allen Knoten n_d in Teilwege zerlegt werden. Aussagen über Wege zwischen Knoten können in der Modellrepräsentation evaluiert werden.

Die elementaren Aussagen über die Definiertheit von VDNObjekten sind Aussagen über die Definiertheit von Variablen, Dialogen und Knoten. Das Augenmerk wird darauf gerichtet, die Aussagen über die Definiertheit von Variablen und Dialoge durch Aussagen über die Definiertheit von Knoten zu ersetzen (Substitution).

3.2.4.6 Negierte Literale

Der Ausdruck envAndMinus kann durch einen äquivalenten Ausdruck envAndMinusNodes ersetzt werden, welcher eine Konjunktion negierter Knotenliterale ist und keine Variablen- und und Dialogliterale mehr enthält.

(sb(anw),d,envAndPlus $\wedge$ envAndMinus) $\models$ true $\leftrightarrow$
(sb(anw),d,envAndPlus $\wedge$ envAndMinusNodes) $\models$ true.

Substitution von Variablen

Eine Variable v_n, die im Ausdruck envAndMinus negiert vorkommt und daher nicht definiert sein soll, ist genau dann nicht definiert, wenn alle Dialoge d_n, die diese Variable enthalten, nicht definiert sind.

$\forall$ v :Vdef | $\neg$v in envAndMinus :
(sb(anw),d,envAndPlus $\wedge$ envAndMinus) $\models$ true $\leftrightarrow$ (d,envAndPlus $\wedge$ envAndMinus') $\models$ true, mit

envAndMinus' := envAndMinus $\vee$ (envAndMinus $\wedge$ v) $\wedge$ $\bigwedge$($\neg$d :Ddef | v ist Variable von d)
Elimination von v *Einfügen der negierten d*

Substitution von Dialogen

Ein Dialog d_n, der in der Aussage envAndMinus negiert vorkommt und deshalb nicht definiert sein soll, ist genau dann nicht definiert, wenn alle seine Knoten N_{dn} nicht definiert sind.

$\forall$ d :Ddef | $\neg$d in envAndMinus : (sb(anw),d,envAndPlus $\wedge$ envAndMinus) $\models$ true $\leftrightarrow$
(sb(anw),d,envAndPlus $\wedge$ envAndMinus') $\models$ true, mit

envAndMinus' := envAndMinus $\vee$ (envAndMinus $\wedge$ v) $\wedge$ $\wedge$($\neg$n :Ndef | n ist Knoten von d)
Elimination von d *Einfügen der negierten n*

Die Konjunktion von Knoten, die nicht definiert sein sollen, kann geschrieben werden als die Menge N_{ignore}, welche diejenigen Knoten angibt, welche bei der Evaluatiuion der Modellrepräsentation durch das Prozessmodell ausgelassen werden sollen.

3.2.4.7 Positive Literale

Das Verfahren der Ersetzung positiver elementarer Aussagen gestaltet sich schwieriger. Es wäre falsch, eine Variable transitiv durch alle ihre Knoten zu ersetzen, d.h. die Gesamtmenge aller Knoten zu bilden, da ein Prozeß nicht sämtliche Knoten einer Variablen durchlaufen muß, damit diese definiert sind, sondern nur mindestens einen.

(sb(anw),d,envAndPlus $\wedge$ envAndMinusNodes) $\models$ true $\leftrightarrow$
(sb(anw),d,envAndPlusNodes $\wedge$ envAndMinusNodes) $\models$ true, mit envAndPlusNodes = Konjunktion aller gültigen Ersetzungen von v und d durch n in envAndPlus

Die Vorgehensweise ist folgende:

Substitution von Variablen

Die Menge der Variablen wird auf eine Menge von Auswahlen der Dialoge abgebildet, d.h. es werden sämtliche möglichen Abbildungen in die Menge der Dialoge vorgenommen. Jede Auswahl ist eine Aussage über Dialoge und wird mit der Aussage über die restlichen Dialoge und Knoten in envAndPlus verknüpft. Damit wird die Aussage envAndPlus in eine Menge von Aussagen EnvAndPlusNoVariables überführt.

Substitution von Dialogen

Die Menge der Dialoge in einer Aussage envAndPlusNoVariables wird auf eine Menge von Auswahlen der Knoten abgebildet, d.h. es werden sämtliche möglichen Abbildungen in die Menge der Knoten vorgenommen. Jede Auswahl ist eine Aussage über Knoten und wird mit der Aussage über die restlichen Knoten in envAndPlusNoVariables verknüpft. Damit wird die Aussage envAndPlusNoVariables in eine Menge von Aussagen EnvAndPlusNodes überführt.

Substitution des Dialogs, an dem das VDNObjekt sichtbar sein soll

Als letztes wird der Dialog d in die Menge der Knoten, die zu diesem Dialog gehören, abgebildet.

(sb(anw),d,envAndPlusNodes $\wedge$ envAndMinusNodes) $\models$ true $\leftrightarrow$ $\forall n_d$:
n_d ist Knoten von d : (sb(anw),n_d,envAndPlusNodes $\wedge$ envAndMinusNodes) $\models$ true

Die Frage, welche Variablen und Dialoge definiert sein müssen, wurde damit auf die Frage, welche Knoten definiert sein müssen, reduziert.

3.2.4.8 Reihenfolgenbildung der Knoten (Permutation)

Mit der Berechnung derjenigen Knoten, die sichtbar sein sollen, kann noch keine Aussage in der Modellrepräsentation evaluiert werden, da diese nicht mit Mengen von Knoten operieren kann. Die Knoten nüssen noch in Reihenfolge gebracht werden. Dadurch erhält man Pfade, von denen einzelne Knoten explizit benannt sind. Alle Pfade beginnen mit dem Knoten PreNode. Solche Pfade lassen sich in Teilpfade zwischen je zwei Knoten zerlegen, welche sich von den Evaluatoren in der Modellrepräsentation analysieren lassen.

envAndPlusNodes :=
$n_1 \wedge n_2 \wedge n_3$ ...

N_{perm} ::=
{ n | n in envAndPlusNode }

N_{ignore} ::=
{ n | $\neg$n in envAndMinusNode }

$(sb(anw), n_d, envAndPlusNodes \wedge envAndMinusNodes) \models true \leftrightarrow \forall permutation(N_{perm}) :$
$path_{NIgnore}(n_{preNode}, permutation(N_{perm}), n_d)$ is valid path $\wedge$
$s(sb(anw), permutation(N_{perm}), n_d, N_{ignore}) \models true.$

$(sb(anw), permutation(N_{perm}), n_d, N_{ignore}) = (sb(anw), path_{NIgnore}(n_{preNode}, n_1, n_2, n_3, ..., n_i, n_d))).$

$(n_d, envAndPlusNodes \wedge envAndMinusNodes, sb(anw)) \models true \leftrightarrow \forall permutation(N_{perm}) :$
$path_{NIgnore}(n_{preNode}, permutation(N_{perm}), n_d)$ is valid path $\wedge$
$(sb(anw), path_{NIgnore}(n_{preNode}, n_1, n_2, n_3, ..., n_i, n_d))) \models true$

Die semantische Bedingung der Anweisung sb(anw) ist erfüllt in Knoten n auf allen Pfaden über die Knoten in N_{perm} unter Auslassung der Knoten in N_{ignore}, wenn für jede Permutation der Knoten in N_{perm}, die einen gültigen Pfad beschreibt, auf diesem Pfad die semantische Bedingung erfüllt ist unter Auslassung der Knoten in N_{ignore}.

3.2.5 Semantische Bedingungen der vier Fälle

Nachdem die Deduktion der Umgebung auf Pfade der Modellrepräsentation reduziert wurden, werden die semantischen Bedingungen der vier Fälle beschrieben.

3.2.5.1 Lesender Zugriff auf Variablen der Sicht history

$(sb(anw), path_{NIgnore}(n_{preNode}, n_1, n_2, n_3, ..., n_i, n_d)) \models true$
sb(anw) = v Zugriff auf Variable v

Prüfen eines Pfades

$(sb(anw), path_{NIgnore}(n_{preNode}, n_1, n_2, n_3, ..., n_i, n_d)) \models true \leftrightarrow$
$s_{NIgnore}(v, n_{PreNode}, n_1) \vee s_{NIgnore}(v, n_1, n_2) \vee s_{NIgnore}(v, n_2, n_3) \vee ... \vee s_{NIgnore}(v, n_i, n_d) = always$

Eine Variable v ist auf einem gültigen Pfad $n_{preNode}, n_1, n_2, n_3, ..., n_i, n_d$, der eine Permutation von N_{perm} ist, in n_d unter Auslassung der Knoten in N_{ignore} genau dann sichtbar, wenn v in einem der Teilpfade $n_{preNode}$-n_1, n_1-n_2, n_2-n_3, ..., n_i-n sichtbar ist unter Auslassung der Knoten in N_{ignore}.

Die Aussagen $s(v, n_i, n_j)$ können in der Modellrepräsentation evaluiert werden. Damit kann diese semantische Bedingung geprüft werden.

Die Sichtbarkeit von Variablen zwischen Knoten kann zur Sichtbarkeit in Pfaden und zur Sichtbarkeit in Permutationen zusammengefaßt werden. Dieses Vorgehen erlaubt es, Aussagen über die Sichtbarkeit in Permutationen durch Sichtbarkeitsverknüpfungen zu substituieren.

$s_{NIgnore}(sb(anw), path_{NIgnore}(n_{preNode}, n_1, n_2, n_3, ..., n_i, n_d)) ::= s_{NIgnore}(sb(anw),n_{PreNode},n_1) \vee s_{NIgnore}(sb(anw),n_1,n_2) \vee s_{NIgnore}(sb(anw),n_2,n_3) \vee ... \vee s_{NIgnore}(sb(anw),n_i,n_d)$

Definition 22: Sichtbarkeit in einem Pfad

$s_{NIgnore}(v,n_{PreNode},n_1) \vee s_{NIgnore}(v,n_1,n_2) \vee s_{NIgnore}(v,n_2,n_3) \vee ... \vee s_{NIgnore}(v,n_i,n_d) = always \leftrightarrow$
$s_{NIgnore}(v, path_{NIgnore}(n_{preNode}, n_1, n_2, n_3, ..., n_i, n_d)) = always$

$(sb(anw), path_{NIgnore}(n_{preNode}, n_1, n_2, n_3, ..., n_i, n_d)) \models true \leftrightarrow$
$s_{NIgnore}(sb(anw), path_{NIgnore}(n_{preNode}, n_1, n_2, n_3, ..., n_i, n_d)) = always$

Prüfen aller Pfade

$(v,n_d,envAndPlusNodes \wedge envAndMinusNodes) \models true \leftrightarrow$
$\forall permutation(N_{perm}) : path_{NIgnore}(n_{preNode}, permutation(N_{perm}), n_d)$ is valid path $\wedge$
$(v, path_{NIgnore}(n_{preNode}, n_1, n_2, n_3, ..., n_i, n_d))) \models true$

$s_{NIgnore}(sb(anw), permutation(N_{perm}),n_d) ::=$ $s_{NIgnore}(sb(anw), path_{NIgnore}(n_{preNode},n_1,n_2,n_3,...,n_i,n_d))$,
if $path_{NIgnore}(n_{preNode}, permutation(N_{perm}), n_d)$ is valid path
undefined,
otherwise

Definition 23: Sichtbarkeit in einer Permutation

$s_{NIgnore}(sb(anw),N_{perm},n_d) ::= \bigwedge_{permutation(Nperm)} s_{NIgnore}(sb(anw),permutation(N_{perm}),n_d)$

Definition 24: Sichtbarkeit in allen Knotenpermutationen

$(v,n_d,envAndPlusNodes \wedge envAndMinusNodes) \models true \leftrightarrow s_{NIgnore}(v,N_{perm},n_d) = always$

Damit gilt:
$(sb(anw),d,env) \models true \leftrightarrow s_{NIgnore}(v,N_{perm},n_d) = always$

3.2.5.2 Das Erzeugen eines Nachfolgedialogs

$(sb(anw), path_{NIgnore}(n_{preNode}, n_1, n_2, n_3, ..., n_i, n_d)) \models true$
$sb(anw) = d_{next}$ Erzeugen des Dialogs d_{next}

$(d_{next}, path_{NIgnore}(n_{preNode}, n_1, n_2, n_3, ..., n_i, n_d)) \models true \leftrightarrow \exists\ n_{new} \mid Dialog(n_{new}) = d_{next} : \exists\ r \mid r = (n_d,n_{new}) \wedge$
$Name(r) = Name(aktRequest)$

Der Dialog d_{next} darf erzeugt werden, wenn für jeden gültigen Pfad $n_{preNode}, n_1, n_2, n_3, ..., n_i, n_d$, der eine Permutation von N_{perm} ist, unter Auslassung der Knoten in N_{ignore} ein Node n_{new} existiert, der Node von d_{next} ist und wenn am Knoten n_d eine Anfrage (*request*) r zum Knoten n_{new} existiert, deren Bezeichnung mit der der aktelluen Anfrage identisch ist. Beachte, daß die aktuelle Anfrage definiert ist, da das Erzeugen eines Dialogs nur in Dienstleisterregeln erlaubt ist.

3.2.5.3 Das Generieren einer Anfrage (*requests*)

$(sb(anw), path_{NIgnore}(n_{preNode}, n_1, n_2, n_3, ..., n_i, n_d)) \models true$
$sb(anw) = r_{next}$ Erzeugen der Anfrage r_{next}

$(r_{next}, path_{NIgnore}(n_{preNode}, n_1, n_2, n_3, ..., n_i, n_d)) \models true \leftrightarrow \exists\, n_{new} : \exists\, r_{next} = (n_j,n_{new})$

Die Anfrage r_{next} darf erzeugt werden, wenn für jeden gültigen Pfad $n_{preNode}, n_1, n_2, n_3, ..., n_i, n_d$, der eine Permutation von N_{perm} ist, unter Auslassung der Knoten in N_{ignore} eine Anfrage r_{next} am Knoten n_d existiert, die zu einem Knoten n_{new} führt.

3.2.5.4 Zugriff auf eine Variable über die Sicht next

$(sb(anw), path_{NIgnore}(n_{preNode}, n_1, n_2, n_3, ..., n_i, n_d)) \models true$
$sb(anw) = v_{next}$ Zugriff auf eine Variable über die Sicht next

$(v_{next}, path_{NIgnore}(n_{preNode}, n_1, n_2, n_3, ..., n_i, n_d)) \models true \leftrightarrow$
$\forall\, n_{new} : \exists\, r \mid r(n_j,n_{new}) \wedge Name(r) = Name(aktRequest) : s(n_{aktDialog},n_{new},v_{next}) = always$

Der Zugriff auf eine Variable v über die Sicht next ist zulässig, wenn für jeden gültigen Pfad $n_{preNode}, n_1, n_2, n_3, ..., n_i, n_d$, der eine Permutation von N_{perm} ist, unter Auslassung der Knoten in N_{ignore} an allen Dialogen, zu deren Knoten eine Anfrage mit der Bezeichnung des aktuellen Requests vom Knoten n_d führt, die Variable v definiert ist.
Formuliert als Sichtbarkeit bedeutet dies, daß die Sichtbarkeit der Variablen v zwischen dem Knoten n_d und dem Knoten n_{new} always sein muß.

Die Deduktion der Bedingung von Anweisungen auf Aussagen, welche von den Evaluatoren über die Modellrepräsentation ausgewertet werden können, ist damit abgeschlossen.

3.2.6 Schleifen

Bei der eingehenden Betrachtung wird klar, daß der Zugriff auf Variablen nun akzeptabel gelöst ist. Ein Schleifen- oder Wiederholkonstrukt, wie es in Programmiersprachen existiert, wurde aber bislang nicht definiert. Dieses Konstrukt ist notwendig, weil ein Schleifenkonstrukt nicht aus den anderen Konstrukten (whex/NextDialog/NextRequest) konstruiert werden kann. Als erstes muß nun die Semantik des Schleifenkonstrukts im Ablauf einer Konversation definiert werden.

Ein Schleifenkonstrukt ermöglicht, sich wiederholende Teile der Konversation in der zeitlichen Reihenfolge zu betrachten. Dabei muß der Kontext eines sich wiederholenden Teils der Konversation, nämlich die Variablenbelegung, für jeden Durchlauf korrekt dargestellt werden. Das impliziert, daß die Werte von Variablen eines jeden Schleifendurchlaufs konsistent sein müssen. Der Schleifendurchlauf wird als "Time Warp" verstanden, d.h, daß für jeden Schleifendurchlauf die Variablenbelegungen eines Teils der Spur sichtbar gemacht wird.

Ein Schleifenkonstrukt wird definiert über eine Bedingung, die im Verlauf der Konversation mehrmals erfüllt ist. In jedem Durchlauf enthalten die Variablen Werte, die beim Erfüllen der Bedingung aktuell waren.

Der erste Entwurf des Schleifenkonstrukts fox *(foreach exists)* orientiert sich am Konzept des Variablenzugriffs. Die Bedingung über eine Variable ist definiert als die Phase, in denen die Variable im aktuellen Dialog definiert wurde.

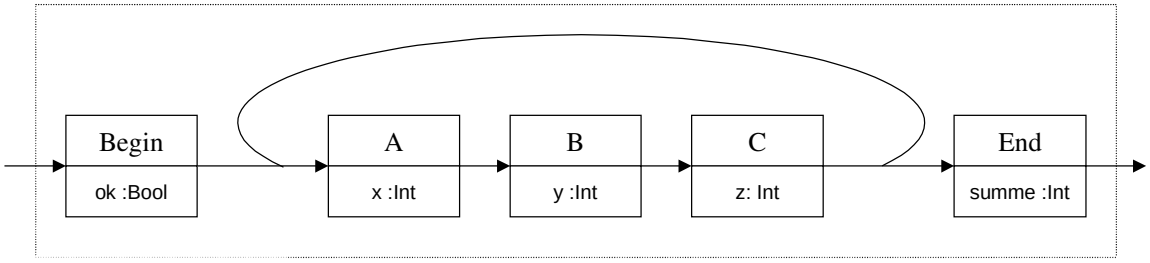


Abbildung 21: Beispielkonversation

```
dialog.summe := 0;
fox( defined( history.y ))
do
  dialog.summe := summe + f(history.x,history.y,history.z);
end
...
```

Codebeispiel 4: Inkorrektes fox Konstrukt: Variable z hat den Wert des vorigen Durchlaufs

Im obigen Codebeispiel einer Regel des Dialogs „End“ wird für jeden Schleifendurchlauf, in dem die Variable „y“ im Dialog „B“ definiert wurde, die Funktion f auf die drei Variablen „x“, „y“ und „z“ angewendet und addiert das Resultat zur „summe“ im Dialog „End“. Wie in Abbildung 22 zu sehen ist, führt das fox-Konstrukt im Beispiel für zwei Schleifendurchläufe zu falschen Resultaten, da die Belegung der Variablen „z“ nicht mit der Belegung der Variablen „x“ und „y“ konsistent ist: Da „z“ nach „y“ definiert wird, wird der passende Wert von „z“ erst im zeitlich folgenden Schleifendurchlauf ermittelt und somit falsch zugeordnet.

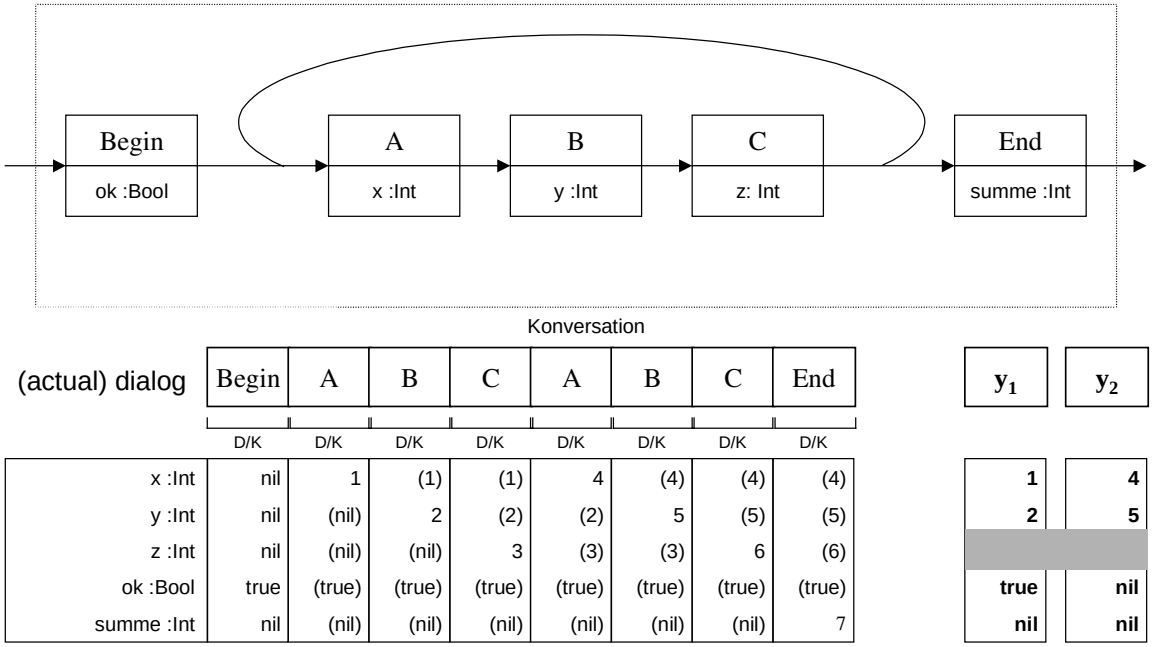


Abbildung 22: (falsche) Variablenbelegung bei Benutzung von VDNObjekten im fox-Konstrukt

Diese Semantik der fox-Konstrukts widerspricht in mehreren Punkten der angestrebten Abstraktion des Prozesses. Eine Konversation ist in mehrere logisch zusammenhängende Teile gegliedert. Die Bedingung ist formuliert in Phasen, die innerhalb eines solchen Teils liegen können. Die Belegungen der Variablen werden nicht in einem Sinnzusammenhang betrachtet, denn Belegungen späterer Phasen werden ausgeschlossen bzw. durch Belegungen, die zu anderen Schleifendurchläufen gehören, verdeckt. Auf der anderen Seite definiert die Bedingung nicht, ob die Belegung diejenige des Kunden oder des Dienstleisters ist.

Es ist daher vonnöten, den Begriff der logisch zusammenhängenden Teile zu formalisieren und darauf das Schleifenkonstrukt anzuwenden.

Teile von Konversationen, die in einem logischen Zusammenhang stehen, sind wiederum Konversationen. Wie jede Konversation müssen sie durch den Dienstleister eingeleitet und durch den Kunden beendet werden. Konversationen in übergeordeneten Konversationen heißen Subkonversationen.

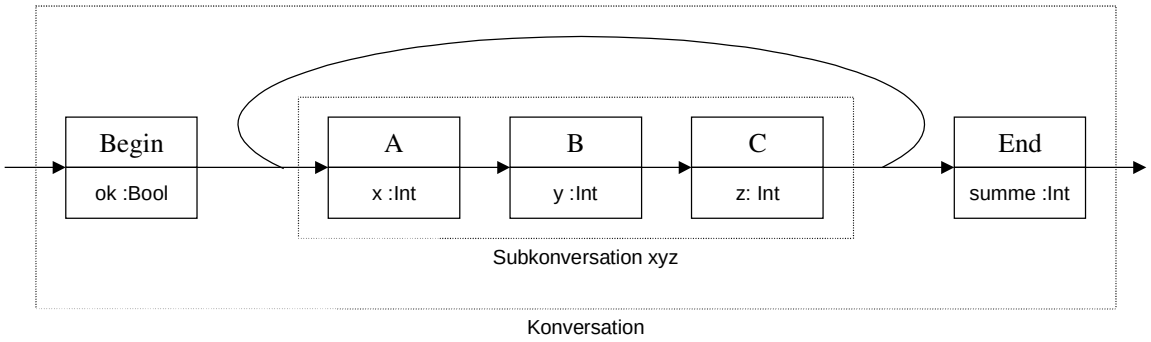


Abbildung 23: Subkonversation xyz

Anmerkung: Der Begriff der Subkonversation existiert auch im Modell der Business Conversations [Wegn98]; zu zeigen, daß der hier verwendete Begriff ein Oberbegriff für die Subkonversationen im Modell der Business Conversations ist, würde den Rahmen der Arbeit sprengen.

Die Namensräume einer Konversation k und einer Subkonversation s von k sind folgendermaßen getrennt:

- Beim Eintreten in s erzeugt der Dienstleister als Dialog einen stellvertretenden initialen Dialog von s. Der Name des Dialogs ist dem Dienstleister dabei nicht bekannt. Dies stellt einen Gewinn an Flexibilität dar, da

beim Veränderung der Subkonversation der Code in den Regeln der Konversation nicht angepaßt werden muß.

- Beim Verlassen von s durch den Kunden erzeugt dieser eine finale Anfrage (*final request*) des aktuellen Dialogs in s. Gibt es mehrere finale Anfragen an einem Dialog, so werden diese durch subkonversationsinterne Namen unterschieden. Finale Anfragen sind all solche Anfragen, die aus der Konversation herausführen. Der Name der Anfrage ist im Normalfall der leere String.

Anmerkung: Die Erweiterung des Prozessablaufs in Konversationen um Verfeinerungs-/Vergröberungskonzepte findet sich bislang nicht im Modell der Business Conversations [Joh97, Wegn98].

Diese Definition stellt eine Subkonversation s in der übergeordneten Konversation k als Dialog von k dar.
Daraus folgt, daß eine Subkonversation wie ein Dialogname in der Bedingung des whex-Konstrukt verwendet werden kann. Im Gegensatz zu normalen Dialogen können Variablen in Subkonversationen über den Namen der Subkonversation referenziert werden.

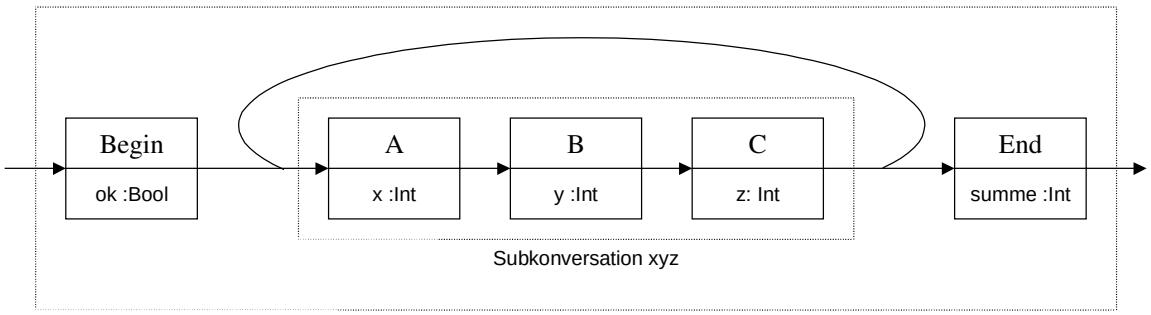
```
whex( executed( xyz ))
do
  resultat := f(history.x,history.y,history.z);
else
...
```

Codebeispiel 5: Adressierung einer Subkonversation in der Bedingung des whex-Konstrukts

Als Bedingung des fox-Konstrukts ist eine Subkonversation bzw. ein Dialog zulässig.

```
dialog.summe := 0;
fox( defined( history.xyz ))
do
  dialog.summe := summe + f(history.x,history.y,history.z);
end
...
```

Codebeispiel 6: Korrektes fox Konstrukt



Konversation

(actual) dialog	Begin	A	B	C	A	B	C	End	xyz_1	xyz_2
	D/K	D/K	D/K	D/K	D/K	D/K	D/K	D/K		
x :Int	nil	1	(1)	(1)	4	(4)	(4)	(4)	**1**	**4**
y :Int	nil	(nil)	2	(2)	(2)	5	(5)	(5)	**2**	**5**
z :Int	nil	(nil)	(nil)	3	(3)	(3)	6	(6)	**3**	**6**
ok :Bool	true	(true)	(true)	(true)	(true)	(true)	(true)	(true)	**true**	**nil**
summe :Int	nil	(nil)	(nil)	(nil)	(nil)	(nil)	(nil)	7	**nil**	**nil**

Abbildung 24: Variablenbelegung bei Benutzung von Subkonversationen im fox Konstrukt

In Codebeispiel 6 wird in einer Regel des Dialogs „End“ über die Subkonversation „xyz“ in jedem Schleifendurchlauf auf die Variablen „x“, „y“ und „z“ zugegriffen. Die Bedingung des fox-Konstrukts impliziert die Refenzierung der korrekten Variablenbelegungen. Im Beispielablauf in Abbildung 24 mit zwei Schleifendurchläufen wird dies nochmals verdeutlicht.

Die Semantik des fox-Konstrukts

Ist als fox-Bedingung eine Subkonversation s angegeben, so gilt:

> Sei h die aktuelle Historie als Sicht history.
> Wiederhole, bis die Subkonversation s in der Historie h nicht durchlaufen wurde:
>
> > Wenn die Subkonversation s in der Historie h mehr als einmal durchlaufen wird, d.h., daß sie als Subkonversationen s1 und s2 in der Spur (*trace*) abgebildet ist, so definiere eine Historie h' als Sicht mit lesendem Zugriff von der nachfolgenden Phase der letzten Phase von s1 bis zur der letzten Phase von s2.
> >
> > Wenn die Subkonversation s in der Historie h nur einmal durchlaufen wird, definere eine Historie h' als Sicht mit lesendem Zugriff von der ersten Phase der Konversation bis zur letzten Phase der Subkonversation s.
> >
> > Führe den Anweisungsblock des fox-Konstrukts unter der Historie h' als Sicht history aus.
> >
> > Ersetze die Historie h durch die Historie h" als Sicht history, die vom Beginn der Konversation bis zur letzten Phase vor der ersten Phase der Historie h' definiert ist.

Ist als fox-Bedingung ein Dialog d angegeben, so gilt:

> Sei h die aktuelle Historie als Sicht history.
> Wiederhole, bis der Dialog d in der Historie h nicht ausgetauscht wurde:
>
> > Wenn der Dialog d in der Historie h mehr als einmal ausgetauscht wird, d.h., daß er als Dialog d1 und d2 in der Spur vorkommt, so definiere eine Historie h' als Sicht mit lesendem Zugriff von der nachfolgenden Phase der letzten Phase, in der d1 aktueller Dialog ist, bis zur letzten Phase von d2 als aktuellem Dialog.
> >
> > Wenn der Dialog d in der Historie h nur einmal durchlaufen wird, definere eine Historie h' von der ersten Phase der Konversation bis zur letzten Phase, in der d aktueller Dialog ist.
> >
> > Führe den Anweisungsblock des fox-Konstrukts unter der Historie h' als Sicht history aus.
> >
> > Ersetze die Historie h durch die Historie h" als Sicht history, die vom Beginn der Konversation bis zur letzten Phase vor der ersten Phase der Historie h' definiert ist.

Die Modellierung des Schleifenkonstrukts fox ist damit abgeschlossen.

3.3 Strukturorientierte Modellierung

3.3.1 Das Strukturmodell

Zur Prozeßanalyse des Konversation gesellt sich ein strukturorientierter Teil. Die beiden Gebiete sind verwandt, wobei die Prozeßorientierung als Ziel hat, mögliche Abstraktionen bei der Abfolge von Dialogen zu bilden, und die Strukturorientierung dementsprechend das Ziel verfolgt, in der Datenstruktur des Inhalts der Dialoge Abstraktionen zu bilden.

Die grundsätzliche Abstraktion im Prozeßablauf ist die, daß beim Zugriff auf Variablen nicht mehr der zugehörige Dialog referenziert wird. Ebenso soll beim Zugriff auf Variablen im Content nicht mehr die Variablenstruktur angegeben werden müssen. Bei der Prozeßorientierung sollten aus dem Ablauf der Konversation Bedingungen abgeleitet werden, die den Zugriff auf Variablen einschränken. Ebenso ist es bei der Strukturorientierung gedacht, den Zugriff auf Variablen von Bedingungen aus dem Zugriff auf andere Variablen einzuschränken.

Die beiden Ziele werden nochmals klar geschildert:

- Abstraktion von der Variablenhierarchie beim Zugriff auf Variablen
- Formulieren von Bedingungen, die den Zugriff auf Variablen vom Zugriff auf andere Variablen einschränken

Die Abstraktion von der Variablenhierarchie wird in dieser Arbeit modelliert als Zugriff auf Variablen über Kurznamen. Bevor der Algorithmus zum Auflösen der Kurznamen beschrieben wird, sollen Entwurfsentscheidungen zur Modellierung kurz dargestellt werden.

Es ist möglich, den (Kurz-)Namen von Variablen sowohl dynamisch zur Laufzeit als auch statisch zur Übersetzungszeit aufzulösen. Die dynamische Namensauflösung bietet den Vorteil, mehrdeutige Kurznamen auch dann auflösen zu können, wenn der statische Lookup die Mehrdeutigkeit nicht zuließe. Der Nachteil liegt darin, daß, weil Variablen gleichen Kurznamens semantisch nicht miteinander verknüpft sind, kein Konzept vorliegt, was dem Subtyppolymorphismus in der objektorientierten Programmierung entspricht. Aus diesem Grund kann bei dynamischer Namensauflösung keine statische Typisierung der Variablen vorgenommen werden. Durch den Typprüfer sollte aber gerade die dynamische Typisierung von Variablen vermieden werden. Um dieses Ziel erreichen zu können, wird auf die dynamische Namensauflösung verzichtet und stattdessen eine statische Namensauflösung zur Übersetzungszeit vorgenommen.

Das zweite Ziel der Strukturorientierung ist, bei der Benutzung von Auswahlkonstrukten aus dem Zugriff auf Variablen in einer Auswahl den Zugriff auf andere Variablen einzuschränken. Die Konstrukte, die eine solche Auswahl realisieren, sind:

- Variante und
- MultipleVariante

Die Auswahlkonstrukte Single- und MultipleChoice werden nicht betrachtet, da sie nur eine Auswahl aus typgleichen Werten realisieren, nicht aber die Auswahl mehrerer Typen.

Die Variante erlaubt die Benutzung der Variablen genau eines Teilbaums, während die MultipleVariante den Zugriff auf keinen bis alle Unterbäume erlaubt, unter der Bedingung, daß diese definiert sind. Im Beispiel in Abbildung 25 erlaubt die Variante somit nur den Zugriff auf den Teilbaum „`standard`“ oder „`extended`“, eine MultipleVariante an dieser Stelle erlaubte den Zugriff auf keinen, einen oder beide Teilbäume.

Zunächst wird die Semantik der Konstrukte erläutert.

3.3.1.1 Variante

Auf eine Variable eines Teilbaums einer Variante kann lesend und schreibend zugegriffen werden, wenn diese oder eine andere Variable desselben Teilbaums definiert ist. Auf die Variable kann nicht lesend zugegriffen werden, wenn eine Variable eines anderen Teilbaums definiert ist. Durch schreibenden Zugriff auf eine Variable

eines Teilbaums einer Variante sind alle Variablen dieses Teilbaums definiert (aber möglicherweise nicht initialisiert), die Variablen der anderem Teilbäume sind anschließend nicht mehr definiert.

3.3.1.2 Multiple Variante

Auf eine Variable eines Teilbaums einer multiplen Variante kann lesend und schreibend zugegriffen werden, wenn diese oder eine andere Variable desselben Teilbaums definiert ist. Durch schreibenden Zugriff auf eine Variable eines Teilbaums einer multiplen Variante sind alle Variablen dieses Teilbaums definiert (aber möglicherweise nicht initialisiert), die Definiertheit der Variablen anderer Teilbäume wird nicht beeinflußt.

```
whex( defined( history.standard ))
do
   ...
   history.breite := 3;
   ...
else
   ...
   history.x := 4;
   ...
end
```

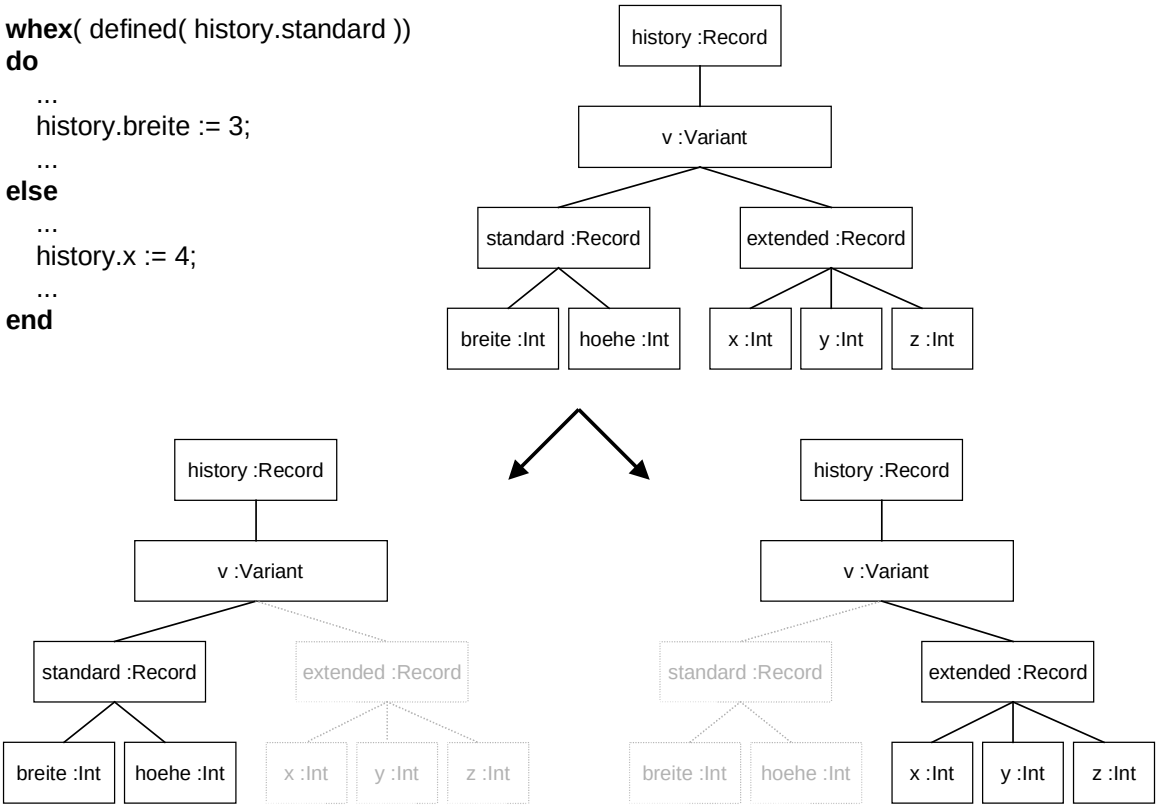


Abbildung 25: Sichtbarkeit von Variablen in Teilbäumen von Varianten

Es folgt für beide Konzepte die Definition eines Konstrukts, welches überprüft, ob eine Variable in einem Teilbaum definiert ist. Das Konstrukt zur Überprüfung ist das whex-Konstrukt, welches bereits im Abschnitt Prozeßmodell beschrieben wurde. Es wird semantisch um die Fähigkeit erweitert, zu überprüfen, ob eine Variable in einem Teilbaum einer Variante / einer multiplen Variante definiert ist. Es bestände die Möglichkeit, zu dieser Überprüfung das Prozeßmodell zu benutzen, um zu überprüfen, welche Variablen der Varianten im Verlauf der Konversation definiert wurden. Das Strukturmodell soll aber bewußt nicht auf dem Prozeßmodell aufsetzen. Aus diesem Grund gilt die Überprüfung der Definiertheit von Variablen der Variante nur lokal in einer Regel. Die strukturelle Bedingung des whex-Konstrukts gilt, anders als die prozeßorientierte Bedingung, nicht für den gesamten eingeschlossenen Anweisungsblock, da bei einer Varianten durch das Schreiben von Variablen eines Teilbaums die Definiertheit anderer Variablen verändert werden kann.

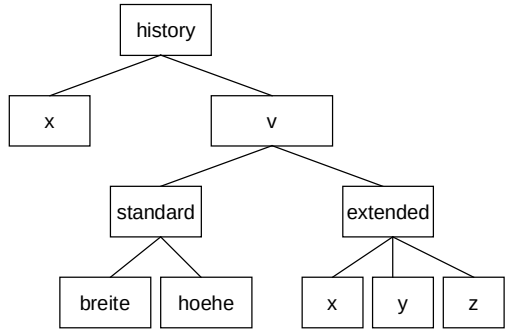
Der Algorithmus zum Überprüfen der Definiertheit von Variablen in Teilbäumen durchläuft sequentiell den Code der Regel. Zu Beginn der Regel ist von allen Varianten kein Teilbaum definiert. Durch whex-Konstrukte und/oder das Schreiben von Variablen werden die Variablen von Variantenteilbäumen definiert und ggf. andere explizit undefiniert.

Dieser Algorithmus zum Prüfen der Definiertheit von Variablen in Variantenteilbäumen wird im Algorithmus zur Namensauflösung verwendet.

3.3.1.3 Sichtbarkeitspunkt (*scopepoint*)

Variablen können über unvollständige Namen referenziert werden. Variablen, welche in der Variablenhierarchie eines Dialogs und damit in der Variablenhierarchie der Spur nahe zusammenstehen, gehören zumeist inhaltlich zusammen, d.h. sie können als semantisch zusammengehörige Gruppe von Variablen verstanden werden. Beim Zugriff auf Variablen über Kurznamen kann es vorkommen, daß die meisten Variablen einer Gruppe durch Kurznamen referenziert werden können, einige jedoch nicht, weil sich eine namensgleiche Variable an höherer Stelle in der Variablenhierarchie der Spur befindet.
Um dieses Problem zu umgehen, wird der Begriff des Sichtbarkeitspunktes (*scopepoints*) eingeführt, der einen sinnvollen Zugriff auf semantisch zusammengehörige Variablen ermöglicht.

Dieser Sichtbarkeitspunkt definiert die Stelle im Variablenbaum, an der der letzte Zugriff auf Variablen, die inhaltlich zusammengehören, stattgefunden hat. Um eine weitere Variable in der Hierarchie der Spur zu finden, wird zunächst die Umgebung des aktuellen Sichtbarkeitspunktes durchsucht, bevor im Rest des Variablenbaums gesucht wird.
Dafür wird zunächst in der Variablenhierarchie unterhalb der Suchwurzel gesucht. Wird dort keine passende Variable gefunden, so wird in der lokalen Umgebung des Sichtbarkeitspunktes gesucht. Wird keine Variable gefunden, so wird diese lokale Umgebung immer weiter ausgedehnt, bis der gesamte Variablenbaum enthalten ist.




```
whex( defined( history.extended ))
do
  history.z := 1;
  history.y := 3;
  history.x := 2; (* history.v.extended.x *)
else
  ...
end
```

Abbildung 26: Anpassung des Namensauflösungsalgorithmus an Lokalität von Variablen

Im Beispiel in Abbildung 26 wird durch den Zugriff auf die Variablen „`history.v.extended.z`“ und „`history.v.extended.y`“ der Sichtbarkeitspunkt auf die Variable „`history.v.extended.y`“ gesetzt, so daß anschließend beim Zugriff auf die Variable „`x`“ die Variable „`history.v.extended.x`“ adressiert wird, und nicht „`history.x`“.

Durch diese lokale Suche wird somit der Zugriff auf zusammengehörende Variablen vereinfacht.

3.3.2 Algorithmus zur Namensauflösung

Der Algorithmus löst Kurznamen von Variablen auf und benutzt dafür den Algorithmus zum Prüfen des Zugriffs auf Variablen in Teilbäumen von Varianten und multiplen Varianten und den Algorithmus des Sichtbarkeitspunktes.

> Sei die Wurzel des Namensraums r der Anfangsknoten bei der Namensauflösung. r ist beim ersten Aufruf gleich der Wurzel der Variablenhierarchie.
>
> Der Sichtbarkeitspunkt ist definiert als die aktuelle Suchwurzel des Algorithmus zur Namensauflösung.
>
> Suche von der Wurzel des Namensraums r zunächst in Breitensuche in allen untergeordneten Knoten nach Variablen, deren Bezeichner der Kurzname ist. Dabei ignoriere Teilbäume, die nicht definiert sind (anhand des Algorithmus zum Überprüfen der Definiertheit von Teilbäumen).
>
> > Finden sich zwei oder mehr Variablen in derselben relativen Tiefe, so ist die Variable nicht eindeutig aufzulösen.
> >
> > Hat sich in der Suche in allen untergeordneten Knoten keine Variable gefunden, so beginne eine neue Suche am Vaterknoten des aktuellen Knotens, wobei der Teilbaum, der den bisher aktuellen Knoten enthält, und alle Teilbäume, die nicht definiert sind, ignoriert werden.
>
> Setze den Knoten, an dem die Variable gefunden wurde, als neue Wurzel des Namensraums.

3.4 Kopplung von strukturorientierter und prozeßorientierter Analyse

Die prozeßorientierte Analyse baut auf der strukturorientierten Analyse auf. Zunächst werden Kurznamen der Variablen aufgelöst und dabei auf strukturorientierte Bedingungen geprüft. In der prozeßorientierten Analyse werden die Variablen dann über die eindeutigen Namen referenziert.

Im diesem Kapitel wurde die prozeß- und strukturorientierte Prüfung von Anweisungen unter Bedingungen beschrieben. Dazu kommt die Definition der abstrakten SET-Sprachkonstrukte whex, fox, nextDialog und nextRequest. Die Sprachdefinition ist nochmals in Anhang A zusammengestellt. Die konkrete Umsetzung der Anweisungsprüfung und die konkrete Tycoon 2 SET-Sprachdefinition, welche sich ebenfalls im Anhang A findet, werden im folgenden Kapitel beschrieben.

4. Implementierung

Teilaufgabe dieser Arbeit war es, einen konkreten zustandserweiterten Typprüfer (*state enriched typechecker*, SET) für eine bestehende Implementation des BC–Modells zu realisieren. Die gewählte Implementation der Business Conversations sind die TBC (Tycoon 2 Business Conversations), welche von Holm Wegner in seiner Diplomarbeit [Wegn98] beschrieben sind.
Die Implementierung des SET hält sich zum größten Teil an die Modellierung aus dem vorigen Kapitel, Abweichungen von dieser werden explizit benannt. Einige Teile der Modellierung sind nicht vollständig implementiert.

Tycoon 2 ist eine Programmierumgebung für persistente, verteilte Systeme, welche anwenderorientierte Informationssysteme in offenen Umgebungen bereitstellen. Diese Programmierumgebung kann am besten mit Java verglichen werden. Sowohl Tycoon 2 als auch Java bestehen aus einer Programmiersprache und einem Laufzeitsystem, in dem der vom Compiler generierte Bytecode von einer virtuellen Maschine ausgeführt wird.
Die besondere Eignung von Tycoon 2 als Programmierumgebung liegt einerseits in der Reflexivität ihrer Programmiersprache TL2, anderseits in der Möglichkeit, in den Übersetzungsprozeß von Quell- zu Zielcode eingreifen zu können, was z.B. Java und C++ nicht bieten. Diese Unterbrechbarkeit des Übersetzungsprozesses hat sich als ein wichtiger Faktor zur zielgerichteten Umsetzung des SET-Konzepts gezeigt, denn in anderen Programmierumgebungen hätten zusätzlich Scanner, Parser und Codegenerator geschrieben werden müssen.

Die hier beschriebene Realisation des SET liegt in TL2 vor, der Programmiersprache von Tycoon 2. Der SET überprüft in TL2 geschriebene Kunden- und Diensleisterregeln von Agenten. Dabei ist der SET in einen sprachabhängigen und sprachunabhängigen Teil untergliedert. Diese Konzeption macht es möglich, durch Anpassung allein des sprachabhängigen Teils Regeln, die in verschiedenen Sprachen programmiert sind, beispielsweise C++, prüfen zu können.

4.1 Die Gliederung des SET

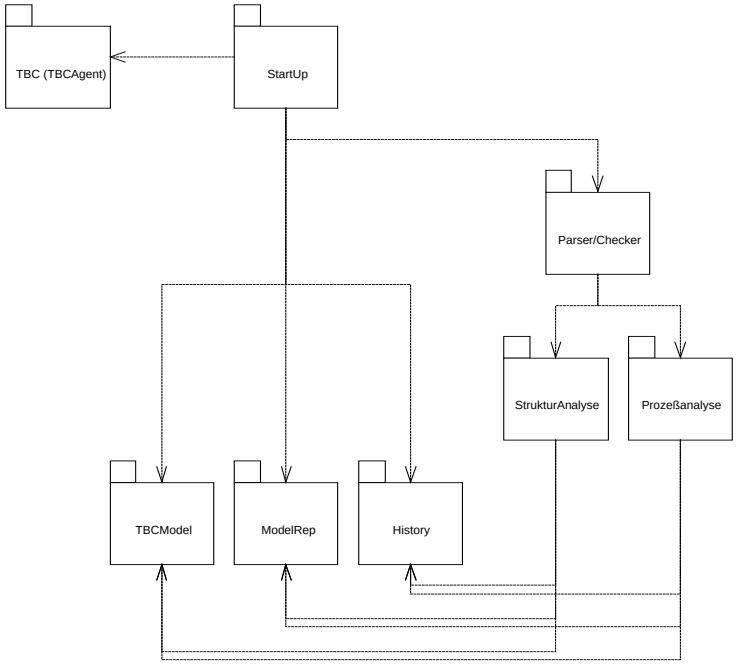


Abbildung 27: Paketdiagramm der SET-Struktur

Der SET (*state enriched typechecker*) kann den Code eines Softwareagenten auf Konformität mit den Konversationsspezifikationen, die dieser in (möglicherweise multiplen) Rollen erfüllen soll, überprüfen. Das Auslesen der Rollen eines Agenten und der darin enthaltenen Konversationsspezifikation wird ausführlich in Kapitel 5 beschrieben.
Dieses Kapitel beschreibt, wie eine Konversationsspezifikation zur Übersetzungszeit im SET-eigenen Objektmodell, der Modellrepräsentation, dargestellt wird, anschließend, wie die Dialoge und deren Variablen im Laufzeitobjektmodell in Historienklassen repräsentiert werden, und wie Parser und Checker respektive Struktur- und Prozeßevaluator auf der Modellrepräsentation bzw. der vom Parser und Checker überprüfte Regelcode auf den Historienklassen arbeitet.

Wie in Abbildung 28 illustriert und in Kapitel 5 ausführlich beschrieben ist, sondiert ein Agentenprüfer (*agent checker*) des Pakets StartUp ein vorhandenes und mit den Regeln initialisiertes Agentenobjekt, wie es in den TBC definiert ist [Wegn98]. Dazu liest er für jede Rolle (*role*) dieses Agenten im ersten Schritt die in den TBC abgelegte Konversationsspezifikation ein und generiert daraus das TBCModel[1]. Anschließend erzeugt der Agentenprüfer die SET-eigenen Erweiterungen der Repräsentation. Nach diesem Schritt ist die Modellrepräsentation (ModelRep) vollständig.

Nun wird der Checker aufgerufen, um die Regeln, die in der Konversation von diesem Agenten ausgeführt werden, zu prüfen. Der Checker ist sprachunabhängig geschrieben; die Prüfung des Programmcodes wird vom sprachabhängigen Parser durchgeführt, welcher die zu untersuchenden Konstrukte an den Checker in einem sprachunabhängigen Format übergibt. Der Parser arbeitet nicht auf dem Quellcode, sondern vielmehr auf dem abstrakten Syntaxbaum (Parsebaum) der Programmiersprache. Aus diesem Grund läßt der Checker vor dem Parsen den Quellcode einer jeden Regel vom TycoonCompilerInterface in einen Parsebaum übersetzen, welcher dann an den Parser gereicht wird.

[1] Das TBCModel ist derjenige Teil der Modellrepräsentation des SET, welcher das TBC-Objektmodell abbildet. Für den SET nötige Erweiterungen des TBC-Modells, beispielsweise Knoten, finden sich im Objektmodell nicht im TBCModel, sondern im Teil ModelRep. Die vollständige SET-Modellrepräsentation umfaßt daher sowohl TBCModel als auch ModelRep.

Die zu prüfenden Konstrukte reicht der Parser in einem sprachunabhängigen Format an den Checker zurück, welcher diese zunächst vom Strukturevaluator und anschließend vom Prozeßevaluator prüfen läßt. Beide Evaluatoren arbeiten auf der Modellrepräsentation.
Die Evaluatoren können die zu prüfenden Konstrukte notwendigerweise anpassen. Die veränderten Konstrukte werden an den Checker zurückgegeben und dieser reicht sie an den Parser zurück. Die Änderungen fließen in den Parsebaum des Regelcodes ein. Die Konsistenz der sprachabhängigen und sprachunabhängigen Konstrukte wird von sogenannten Umhüllern (*wrappers*) sichergestellt. Sie stellen nach dem Adaptermuster[GHJV95] sprachabhängige und –unabhängige Schnittstellen für die Konstrukte bereit.

Wenn die Überprüfung des Regelcodes keine Fehler ergeben hat, übersetzt das TycoonCompilerInterface den Parsebaum in Tycoon 2 Bytecode. Damit ist die Überprüfung des Codes abgeschlossen.

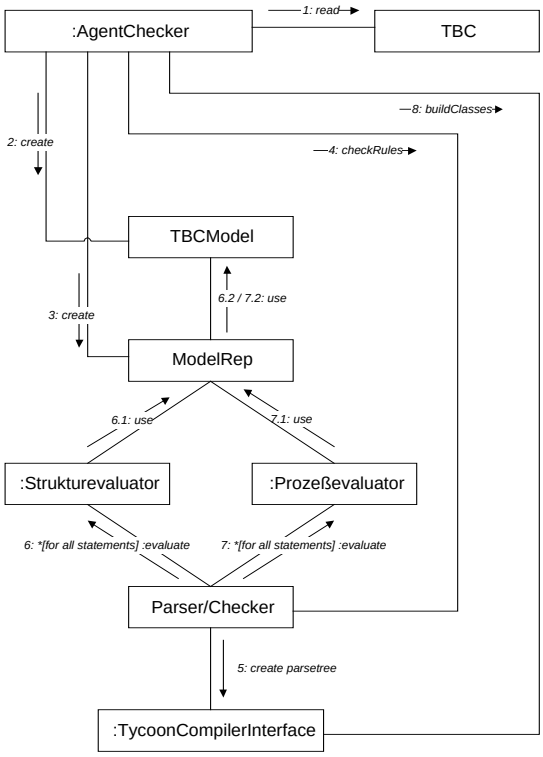


Abbildung 28: Kollaborationsdiagramm der Komponenten

4.2 Das SET-Objektmodell

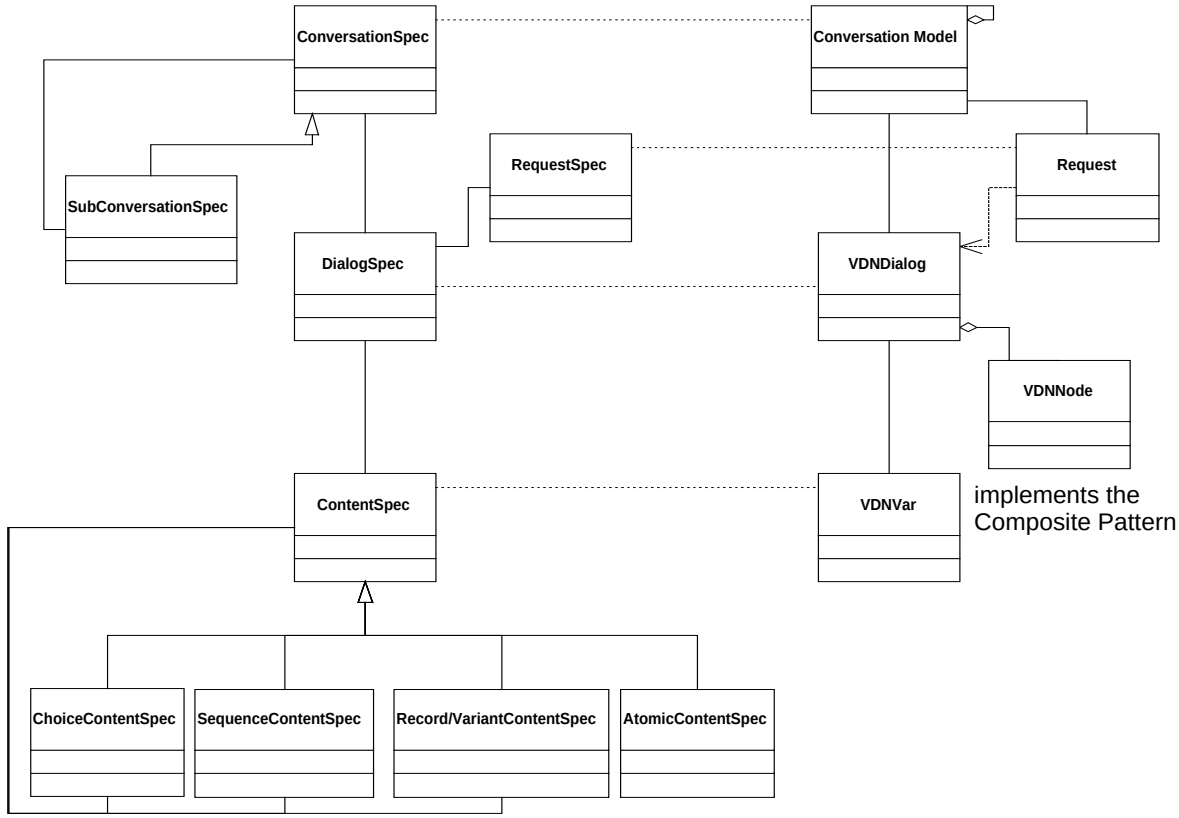


Abbildung 29: Klassendiagramm: Parallele Hierarchie

4.2.1 Die Modellrepräsentation: TBCModel

Die Modellrepräsentation definiert die SET-spezifische Abbildung der Klassen der BC–Modells. Die Klassen des TBC–Systems von Holm Wegner [Wegn98] wurden nicht verwendet, da die Konzepte von TBC und SET zu unterschiedlich sind, als daß die TBC-Klassen sinnvoll benutzt werden könnten. Zudem ist die Modellrepräsentation um mehrere Konzepte erweitert, die sich nicht im BC–Modell finden. Deshalb wurde der Ansatz gewählt, eine parallele Klassenhierarchie zu der der TBC zu bilden.

Die Teil Modellrepräsentation, der die TBC Klassen abbildet, heißt TBCModel. Die Struktur des TBCModel und die Abbildung der TBC Klassen ist in Abbildung 29 illustriert. Zu beachten ist, daß die verschiedenen ContentSpec-Klassen der TBC in einer Klasse des TBCModel, VDNVar, repräsentiert werden. Die Klasse VDNVar implementiert das Compositemuster[GHJV95] und bildet somit die Schnittstelle aller Unterklassen von ContentSpec ab.

ConversationsSpecs werden im TBCModel durch die Klasse ConversationModel repräsentiert. Die Klasse ConversationModel kann hierarchisch geschachtelt werden. Damit sind die in Kapitel 3.2.6 beschriebenen Subkonversationen, welche den Begriff der Subkonversationen des BC-Modells einfassen, in der Implementation des SET bereits enthalten, auch wenn dieses Konzept auf Seiten der TBC noch nicht benutzt wird.

4.2.2 Die Modellrepräsentation: Teil ModelRep

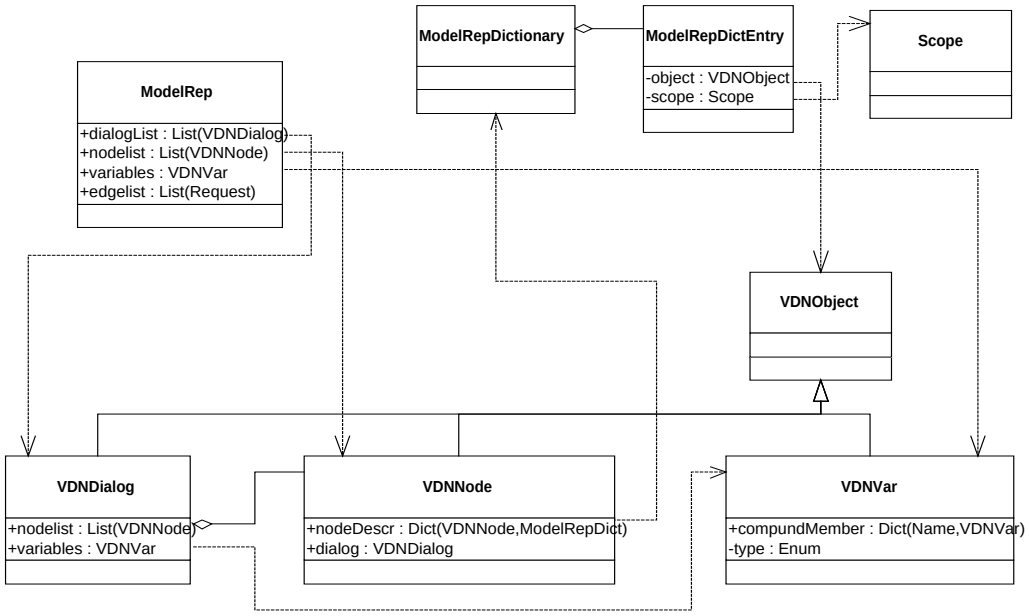


Abbildung 30: Klassendiagramm ModelRep

Der Teil ModelRep des Objektmodells setzt auf dem TBCModel auf. Er erweitert die Modellrepräsentation um das Konzept der Knoten (*nodes*), führt den Begriff des VDNObjekts und der Sichtbarkeit (*scope*) ein definiert eine passende Repräsentation des Sichtbarkeit von VDNObjekten zwischen je zwei Knoten.

Wie in Abbildung 30 zu sehen, ist die Klasse VDNObjekt die abstrakte Oberklasse der Dialoge (Klasse VDNDialog), Knoten (Klasse VDNNode) und Variablen (Klasse VDNVar). In der Implementierung ist das Konzept der Vererbung allerdings ersetzt durch typkorrekte Attribute.

Die Modellrepräsentation ist so aufgebaut, daß jede Variable in der Variablenhierarchie der Modellrepräsentation definiert ist und sie von allen Dialoge referenziert wird, in denen sie als *dialog content* definiert ist.

Die Sichtbarkeit aller VDNObjekte zwischen zwei Knoten n_i und n_j werden in einem ModelRepDictionary gespeichert. Dieses enthält für jedes VDNObjekt einen Eintrag (ModelRepDictEntry), der mit jedem VDNObjekt die zugehörige Sichtbarkeit assoziiert. Jeder Knoten n_i enthält einen Knotenbeschreiber (NodeDescriptor) in Form eines Dictionaries, das jeden Knoten n_j mit dem passenden ModelRepDictionary assoziiert.

4.2.3 Die Historienklassen

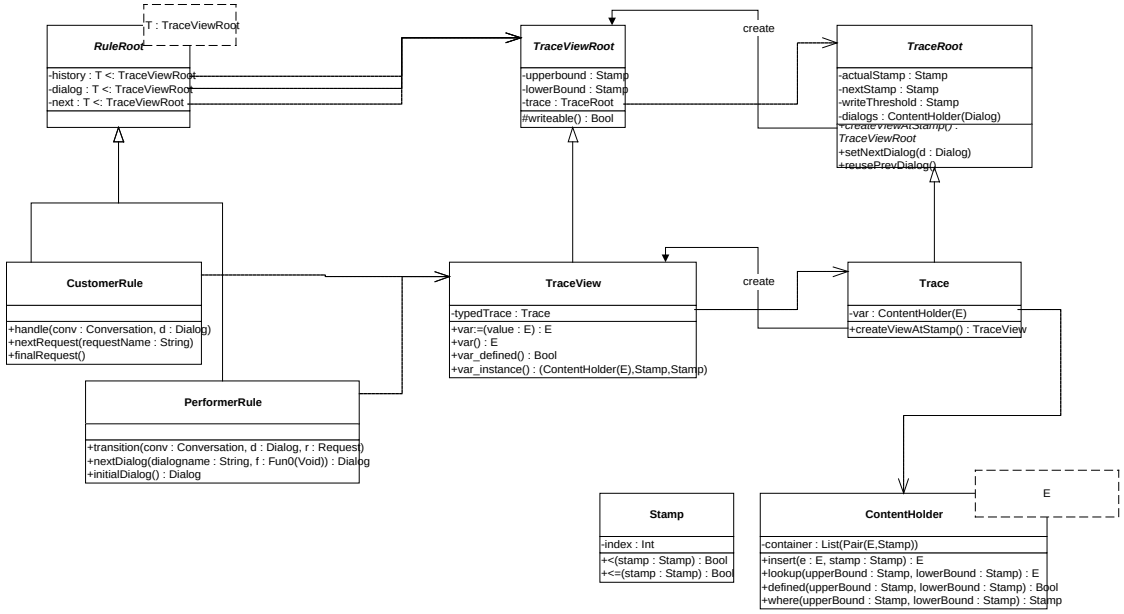


Abbildung 31: Klassendiagramm Historienklassen

4.2.3.1 Zeitstempel (*Stamp*)

Der Begriff der Phase einer Konversation wird in den Historienklassen implementiert als Zeitstempel (Klasse Stamp). Verschiedene Phasen einer Konversation werden durch verschiedene Zeitstempel referenziert. Zeitstempel können miteinander verglichen werden und der Nachfolger eines Zeitstempels (d.h. die nachfolgende Phase) ist definiert.

4.2.3.2 Behälter (*Contentholder*) und Spur (*Trace*)

Die Spur (Klasse Trace) verwaltet die Variablenbelegungen einer Konversation über den zeitlichen Ablauf.

Es bietet sich an, die Variablenbelegungen in typisierten Behältern (Klasse Contentholder) abzulegen. In einem Behälter werden die Belegungen einer einzelnen Variablen für jede Phase der Konversation abgelegt. Die Verwaltung der Variablenbelegung im Behälter kann dadurch noch zusätzlich optimiert werden, daß die Behälter nicht direkt die Belegungen verwalten, sondern über Dictionaries den Phasen Belegungen zuordnen. Somit können Phasen, in denen die Belegung nicht geändert wurde, ausgelassen werden. Dies führt zu größerer Platzersparnis bei längeren Konversationen.

Die Spur (Klasse Trace) verwaltet die Behälter der Variablen (Klasse Contentholder) aller Dialoge.

Die bereits abgelaufenen Dialoge der Konversation werden von der Spur als Liste mit entsprechenden Zugriffsmethoden abgelegt. Über die Liste der Dialoge einerseits und die Behälter andererseits verwaltet die Spur die aktuelle Phase und die nachfolgende Phase der Konversation (welche nur vom Dienstleister benötigt wird). Zur aktuellen und nachfolgenden Phase verwaltet sie eine Schreibschwelle, d.h. einen Zeitstempel, der den Schreibzugriff regelt, so daß nur die Belegungen der Variablen in Phasen, die nicht durch die Schreibschwelle gesichert sind, verändert werden können.

4.2.3.3 Spursicht (*Traceview*)

Eine Spursicht (Klasse Traceview) ist eine Sicht auf die Spur. Jede Sicht definiert einen sichtbaren Bereich der Konversationsphasen. Die Bereichsgrenzen werden über zwei Zeitstempel realisiert, upperbound und lowerbound.
Da eine konkrete Sicht typkorrekt auf die Variablen der konkreten Spur zugreifen soll, liegt die Erzeugung einer Spursicht bei der Spur, realisiert nach dem factory method-Muster[GHJV95].

4.2.3.4 Einbindung der Spursichten in Regeln des TBC-Modells

Die konkreten Spurklassen (*trace classes*) enthalten Behälter (*contentholder*) für die Variablen einer konkreten Konversation, während die konkreten Spursichtklassen (*traceview classes*) Methoden zum Zugriff auf diese Behälter bieten. Die konkreten Spur- und Spursichtklassen, welche für eine Konversation benötigt werden, werden im Verlauf des Überprüfens des Regelcodes vom Agentenprüfer (*agent checker*) erzeugt.

Das Problem ist die Einbindung der dynamisch generierten Spur- und Spursichtklassen in die vom Benutzer des SET bereitgestellten Regeln. Der Lösungsansatz ist folgender:
Die Regeln sind, nach objektorientierter Programmierung, als Algorithmen in Klassen abgelegt. In Dienstleisterregeln müssen für den Zugriff auf Variablen die Spursichten history, dialog und next existieren, in Kundenregeln die Spursichten history und dialog. Diese können zum Zeitpunkt der Erstellung der Regel nur als Assoziationen auf abstrakte Spursichten verweisen, die konkrete Implementierung kann noch nicht existieren. Es stellen sich zwei Unterprobleme:

- Zur Erzeugung der konkreten Spursichten über die Spur muß diese als Objekt aus jeder Regel erreichbar sein, soll nicht das TBC-Modell um Funktionalität erweitert werden, die dem SET eigen ist.
- Der konkrete Typ der Spursicht muß in den Regeln bekannt sein. Es reicht nicht, über das factory method-Muster[GHJV95] über die konkreten Spur konkrete Spursichten zu erzeugen und diese über die Schnittstelle der abstrakten Spursicht zuzugreifen, da so gerade nicht auf die Variablen der konkreten Konversation zugegriffen werden kann. Auf der anderen Seite werden die konkreten Spur- und Spursichtklassen automatisch vom Agentenprüfer generiert; die Namen der konkreten Klassen können also auch nicht vom Programmierer der Regeln antizipiert werden.

Das erste Problem wird gelöst, indem die Spur Attribut der Konversation wird. Die Konversation wird jeder Regel als Parameter übergeben [Wegn98]. Damit können über die konkreten Spur der Konversation passende Spursichten erzeugt werden.

Das zweite Problem wird über einen Typparameter gelöst. Die Regeln werden um einen Typparameter *T* erweitert, welcher den konkreten Typ der Spursichten bestimmt. Dieser Typparameter verweist zur Übersetzungszeit auf die abstrakte Spursichtklasse. Nach der Generierung der konkreten Spur- und Spursichtklassen wird vom Agentenprüfer vor der Übersetzung der Regelklassen der abstrakte Typparameter durch den konkreten Typparameter ersetzt. Damit haben die Spursichten der Regeln nun den konkreten Typ. Der Zugriff auf die Variablen der konkreten Konversation über die Spursichten ist somit möglich.

4.3 Der SET–Checker

Der Kern des SET ist der Checker, welcher die in Kapitel 3 (Modellierung) beschriebenen Sprachkonstrukte überprüfen kann. Vor einer Konzeption des Checkers werden zunächst die Konstrukte, die der Checker überprüfen soll, beschrieben.

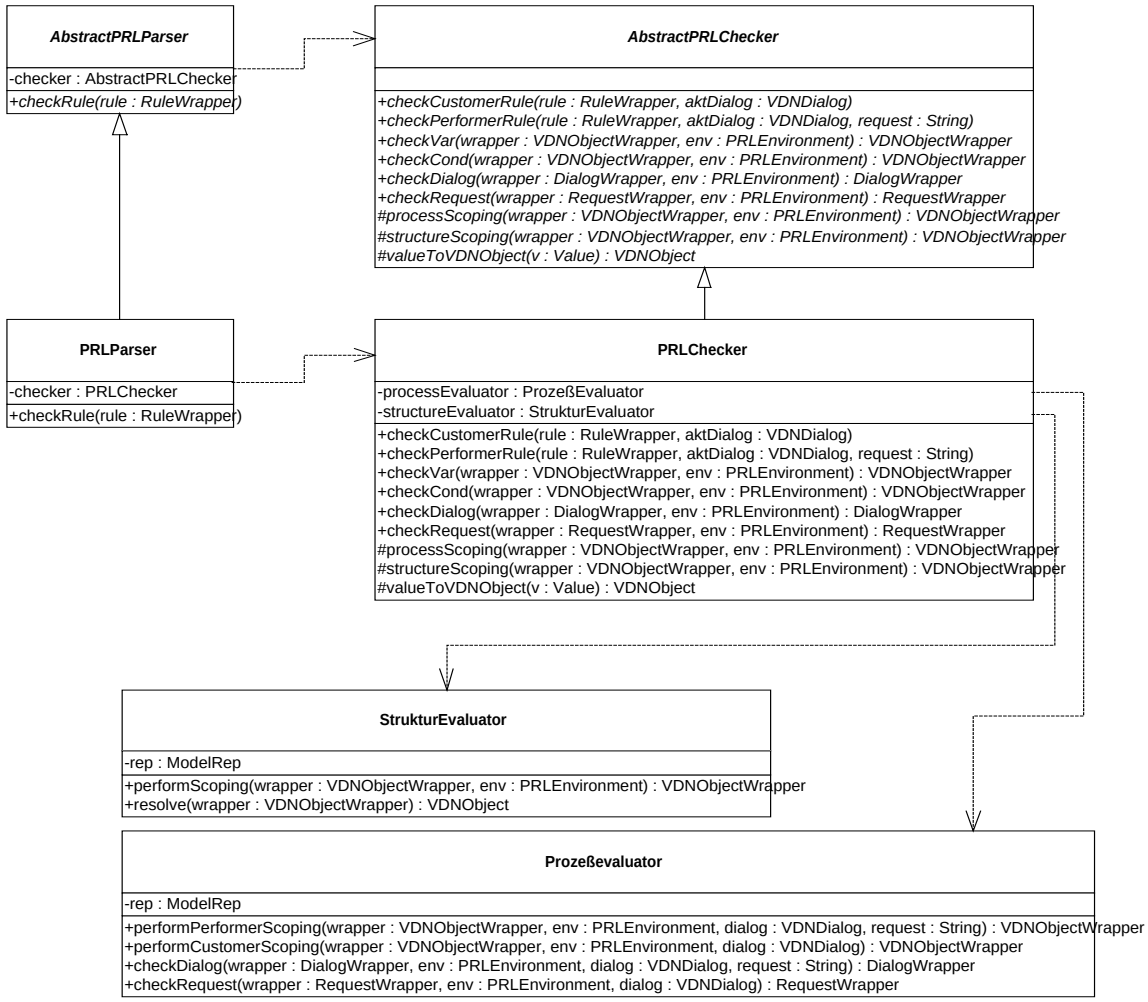


Abbildung 32: Klassendiagramm Checker/Parser

Der Checker wird vom Agentenprüfer benutzt, um entweder eine Dienstleister- oder Kundenregel zu prüfen. Die Regel reicht der Checker zunächst an den Parser weiter. Dieser sondiert den Regelcode und übergibt die zu prüfenden Programmkonstrukte an den Checker. Dieser prüft die Konstrukte nicht selbst, sondern reicht sie an die Struktur- und Prozeßevaluatoren weiter. Diese prüfen das Konstrukt und verändern es ggf.
Das überprüfte Konstrukt wird anschließend wieder an den Checker zurückgereicht. Wenn es zulässig ist, wird es an den Parser zurückgereicht, der das veränderte Programmkonstrukt in die Regelcode intergriert. Nach Prüfung der gesamten Regel wird diese vom Checker an den Agentenprüfer zurückgegeben.

4.3.1 Konstrukte

Wie bereits aus dem Kapitel Modellierung (3.2.4 Der Prozeßevaluator) bekannt, müssen mehrere Konstrukte überprüft werden:

- Variablenzugriff über Spursichten
- Erzeugung eines Nachfolgedialogs
- Erzeugung einer Anfrage (*request*)

Die Formalisierung der Konstrukte und deren Parametrisierung aus dem Kapitel Modellierung wird hier aufgegriffen.

Variablenzugriff von Dienstleister (pAccess) und Kunde (cAccess)
pAccess ::= (View,V,RW,pEnv,aktDialog,aktRequest)
cAccess ::= (View,V,RW,cEnv,aktDialog)

Erzeugung eines Nachfolgedialogs durch den Dienstleister (pDialog)
pDialog ::= (D,pEnv,aktDialog,aktRequest)

Erzeugung einer Anfrage durch den Kunden (cRequest)
cRequest ::= (R,cEnv,aktDialog)

mit

pEnv ::= (Definiertheitsumgebung, Erzeugungsumgebung)
cEnv ::= Definiertheitsumgebung

In der Implementierung ist die Unterscheidung zwischen Dienstleister- und Kundenumgebung aufgehoben. Die Umgebung besteht aus Definiertheitsumgebung und Erzeugungsumgebung, wobei letztere nur in Dienstleisterregeln verwendet wird.

Die Parameter aktDialog und aktRequest, die für alle Anweisungen einer Regel invariant sind, sind als Parameter des Checkers zum Prüfen einer Regel und nicht als Parameter der Anweisungen implementiert.

Der Checker implementiert zwei Schnittstellen:

- Die Schnittstelle zum Benutzer (*client*), in diesem Fall der Agentenprüfer, welcher den Checker benutzt, um eine Regel überprüfen zu lassen.
- Die Schnittstelle zum Parser, welcher vom Checker aufgerufen wird, den Programmcode der Regel zu interpretieren und die Konstrukte vom Checker überprüfen zu lassen. Die zweite Interaktion zwischen Parser und Checker funktioniert dabei über Callbacks.

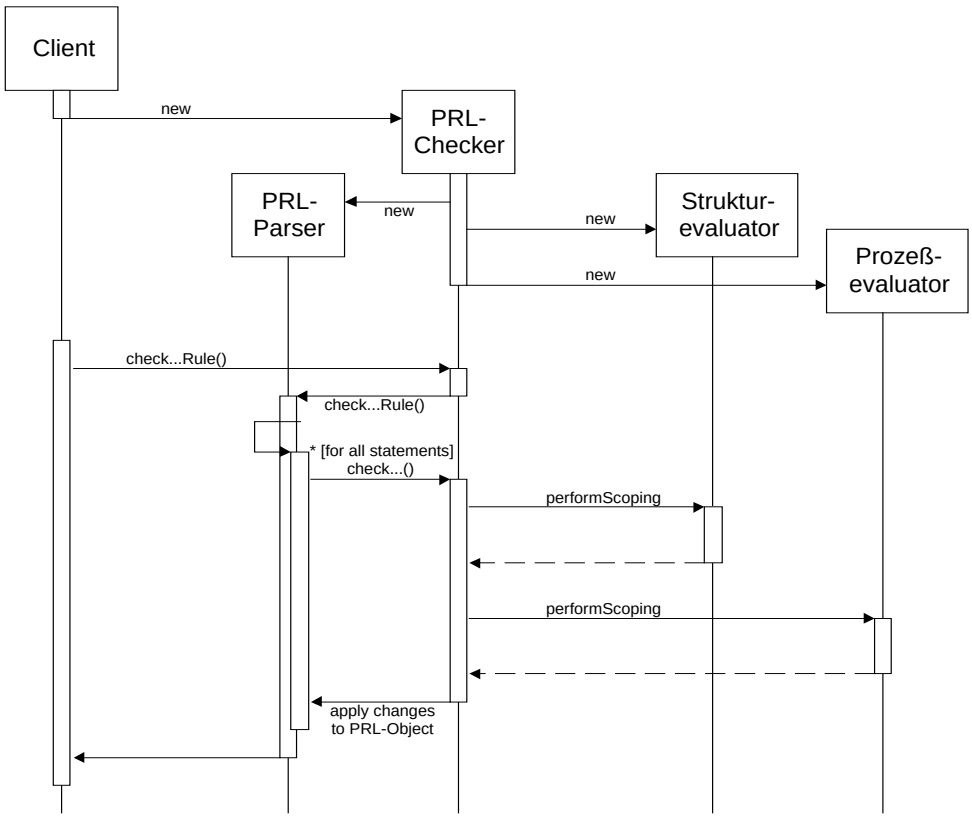


Abbildung 33: Interaktionsdiagramm Parser/Checker

Der Checker hat also folgende Schnittstellen:

```
checkPerformerRule( ruleClass, aktDialog, aktRequest )
checkCustomerRule( ruleClass, aktDialog )
```

Tabelle 3: Schnittstelle für den Benutzer

```
checkVar( View, V, RW, Env )  // Variablenzugriff
checkDialog( D, Env )         // Erzeugung eines Nachfolgedialogs
checkRequest( R, Env )        // Erzeugung eines Requests
checkCond( V, Env )           // Überprüfen einer Bedingung
```

Tabelle 4: Schnittstelle für den Parser

Die Aufteilung der Verantwortung ist folgende: Der Checker erhält vom Parser die Umgebung der zu prüfenden Anweisungen als Parameter. Diese ist zu Beginn des Prüfprozesses leer.
Der Parser kennt nicht die innere Struktur von Umgebungen, er kennt nur Konstrukte im Programmcode, welche eine Umgebung definieren. Diese Konstrukte kann er vom Checker überprüfen lassen. Dazu extrahiert der Parser die Bedingungen der Konstrukte whex und fox und übergibt sie dem Checker. Dieser prüft die Bedingung und liefert sie als Umgebung an den Parser zurück. Die bisherige Umgebung und die gelieferte Umgebung werden zu einer neuen Umgebung verknüpft. Diese neue Umgebung wird dem Checker beim Prüfen der folgenden Konstrukte als Parameter übergeben.

Die Entwurfsentscheidung, welche Klasse dafür verantwortlich ist, die Umgebungen zu verknüpfen, wie es beispielsweise beim whex-Konstrukt nötig ist, wird kurz beschrieben.
Grundsätzlich müßte der Checker die Umgebungen verknüpfen, da die Information, daß Umgebungen verknüpft werden müssen, bei ihm liegt. Dagegen spricht, daß Umgebungen komplex sein können, d.h., daß auch eine Umgebung aus nicht-elementaren Umgebungen zusammengesetzt sein kann. Der Checker soll aber die Struktur zusammengesetzter Umgebungen nicht kennen.
Aus diesem Grund wurde ein anderer Weg gewählt. Es gibt eine Klasse DNFTree, mit der Umgebungen zusammengesetzt werden können. Es ist Aufgabe des Parsers, (komplexe) Bedingungen in elementare Bedingungen zu zerlegen, und diese vom Checker prüfen zu lassen. Der Checker liefert diese als Umgebung zurück, und der Parser benutzt die Klasse DNFTree, um die Umgebungen zu verknüpfen.

Wie bereits an den Begriffen „Bedingung“ und „Umgebung“ erkennbar, arbeitet der Parser mit Programmiersprachobjekten und Bedingungen, der Checker allerdings mit Dialogen, Anfragen und Umgebungen. Dabei handelt es sich um dieselben Objekte unter verschiedenen Sichtweisen. Da diese Objekte auch mehrmals zwischen Parser und Checker ausgetauscht werden und Änderungen an den Objekten beim Checker Einfluß auf den Zustand der Objekte beim Parser haben sollen, muß hier ein Modell geschaffen werden, über das beide auf die Objekte über unterschiedliche Schnittstellen zugreifen können. Ein solches Modell wird realisiert über das Adapter Muster[GHJV95]. In der Implementierung wird dafür der Begriff des Umhüllers (Wrapper) verwendet. Die Programmiersprachobjekte des Parsers werden zunächst in Wrapper „eingehüllt“. Der Wrapper stellt Funktionalität zur Verfügung, mit Hilfe derer der Checker das mit dem Programmiersprachobjekt assoziierte VDNObjekt der Modellrepräsentation finden kann. Nach der Zuordnung kann der Checker über seine Schnittstelle des Wrappers auf das Objekt zugreifen. Änderungen des Checkers bzw. der Prozeß- und Strukturevaluatoren werden über die Schnittstelle des Wrappers für den Parser sichtbar, so daß dieser das Objekt im Parsebaum dementsprechend verändern kann.

Programmiersprache (Parser)	Modellrepräsentation (Checker)	Wrapper
Zugriff auf Variablen		
Programmvariable	Variable (<: VDNObjekt)	VDNObjektWrapper
Bedingung		
defined(Programmvariable)	Variable (<: VDNObjekt)	VDNObjektWrapper
executed(DialogName)	Dialog (<: VDNObjekt)	VDNObjektWrapper
Nachfolgedialog erzeugen		
DialogName	Dialog (<: VDNObjekt)	DialogWrapper
Anfrage erzeugen		
RequestName	Request	RequestWrapper

Die beiden Programmiersprachkonzepte „Variablenzugriff“ und „Bedingung einer Anweisung“ werden über den VDNObjektWrapper abgewickelt.

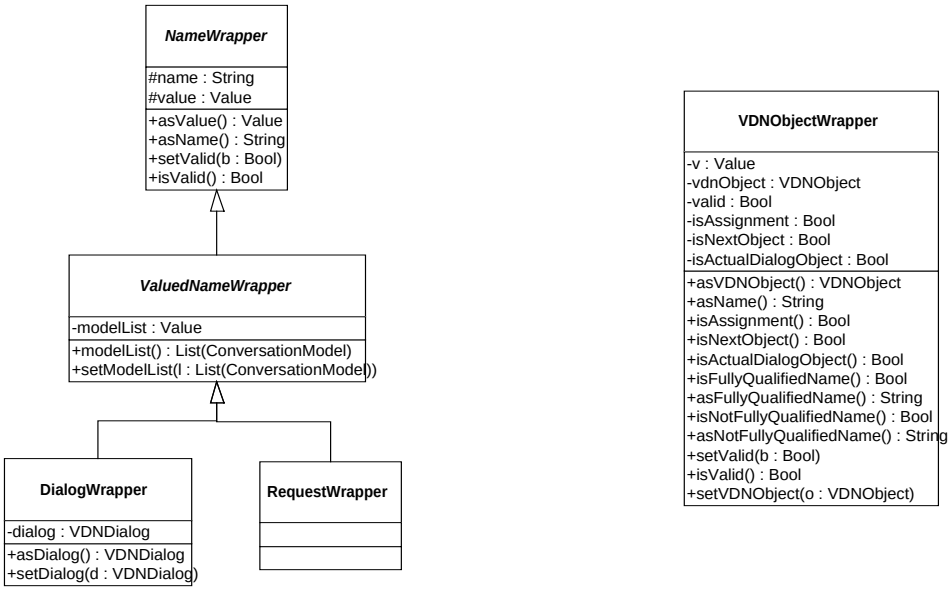


Abbildung 34: Klassendiagramm der Wrapper

Die Wrapper leisten den Zugriff auf Programmiersprachobjekte über eine sprachabhängige Schnittstelle (*asValue, asName*), die der Parser benutzt und eine sprachunabhängige Schnittstelle (*asDialog, asVDNObjekt*), die der Checker und die Evaluatoren benutzen.

4.3.2 Checker

Der Checker besteht aus drei Teilen:

- Strukturevaluator
- Prozeßevaluator
- Prozeßmodell

Die Schnittstelle des Checkers vom Parser ist folgende:

```
checkVar( View, V, RW, Env )  // Variablenzugriff
checkCond( V, Env )           // Überprüfen einer Bedingung
checkDialog( D, Env )         // Erzeugung eines Nachfolgedialogs
checkRequest( R, Env )        // Erzeugung einer Anfrage
```

Die drei Komponenten des Checkers werden nicht für alle Methoden verwendet. So wird eine Bedingung nicht vom Prozeßevaluator geprüft; ein Dialog bzw. eine Anfrage (*request*) nicht vom Strukturevaluator.

Methode	Strukturevaluation	Prozeßevaluation
checkVar	Ja	Ja
checkCond	Ja	Nein
checkDialog	Nein	Ja
checkRequest	Nein	Ja

Wie bereits im Kapitel 3 (Modellierung) beschrieben, vervollständigt der Strukturevaluator unvollständige Variablennamen unter Beachtung der Variablenhierarchie. Der Prozeßevaluator überführt die Umgebung einer Anweisung in Knotenfolgen und evaluiert über diese Folgen Aussagen über die Zulässigkeit von Anweisungen. Dazu benutzt er intern das Prozeßmodell.

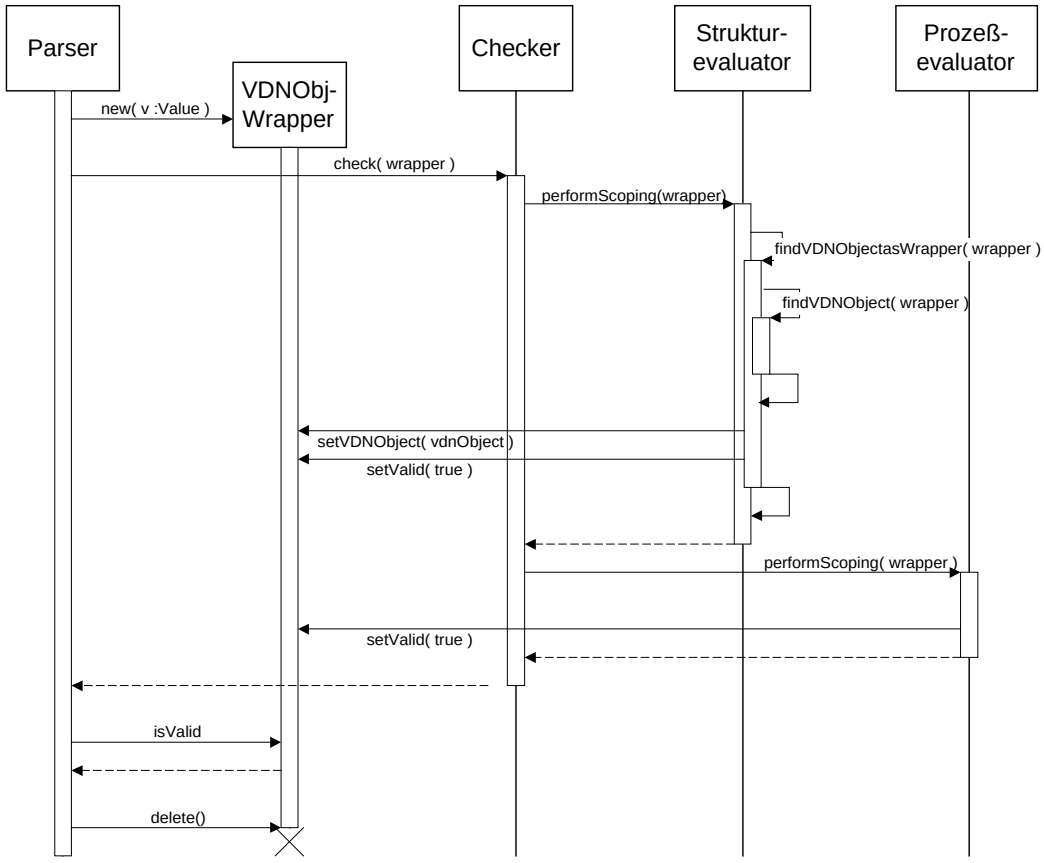


Abbildung 35: Interaktionsdiagramm VDNObjectWrapper

4.3.3 Der Strukturevaluator

Aufgabe des Strukturevaluator ist es, abstrakte, d.h. nur partiell angegebene Variablen- und Dialognamen zu vollständigen Namen zu ergänzen, und die Variablen entsprechend der Struktur der Variablenhierarchie zu referenzieren.
Variablen im Variantenstrukturen können sich bei der Verwendung gegenseitig ausschließen; der korrekte Zugriff muß vom Strukturevaluator geprüft werden. Von diesem beiden Aufgaben ist in der vorliegenden Arbeit nur der erste Teil realisiert; die theoretische Betrachtung beider Aufgaben erfolgte aber im Kapitel 3 (Modellierung).

Um eine Variable im Variablenbaum zu finden, wird dieser in Breitensuche durchlaufen. Der Algorithmus wird verfeinert durch zwei Konzepte:

- Bedingungen zur Sichtbarkeit von Teilbäumen in Varianten und
- Sichtbarkeitspunkt

4.3.3.1 Bedingungen zur Sichtbarkeit von Teilbäumen in Varianten

Varianten definieren Teilbäume des Variablenbaums der Spur, welche sich gegenseitig ausschließen. Die Bedingung des whex-Konstrukts definiert für die eingeschlossenen Anweisungen über die Umgebung die Sichtbarkeit solcher Teilbäume, d.h. die Umgebung wird über den Variablenbaum evaluiert und teilt diesen in sichtbare und nicht sichtbare Teilbäume. Die Sichtbarkeit wird im Strukturevaluator implementiert durch das Markieren von Teilbäumen als sichtbar / nicht sichtbar.

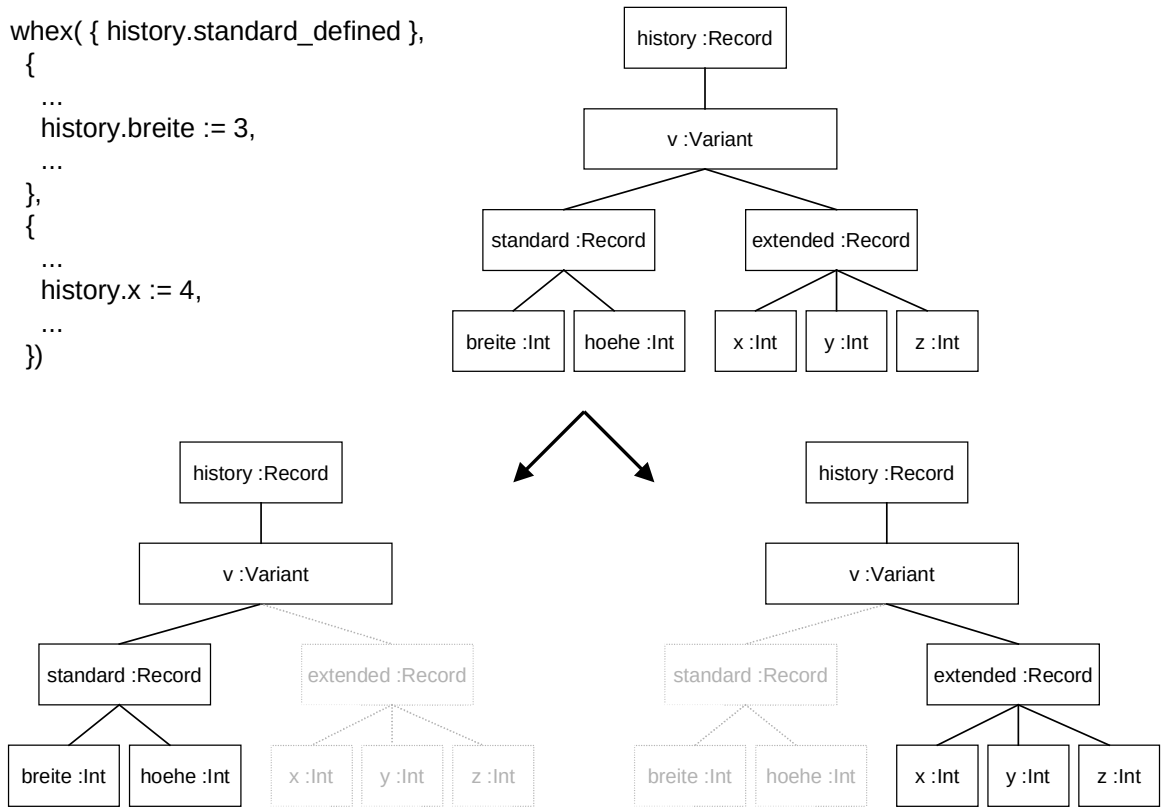


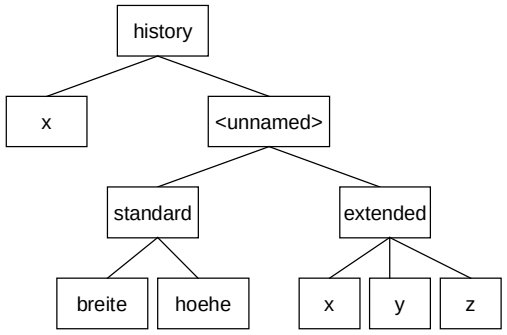
Abbildung 36: Sichtbarkeit von Teilbäumen

4.3.3.2 Sichtbarkeitspunkt

Variablen können über unvollständige Namen referenziert werden. Dabei gehören Variablen, welche in der Hierarchie nahe zusammenstehen, auch inhaltlich zusammen, d.h. sie können als semantisch zusammengehörige

Gruppe von Variablen verstanden werden. Nun kann es sein, daß die meisten Variablen dieser Gruppe durch unvollständige Namen referenziert werden können, doch eine oder wenige nicht, weil sich eine namensgleiche Variable an höherer Stelle in der Hierarchie befindet. Um dieses zu umgehen, wurde der Begriff des Sichtbarkeitspunkts (*scopepoint*) in Kapitel 3.3.1.3 eingeführt.

Der Sichtbarkeitspunkt definiert die aktuelle Suchwurzel w des Algorithmus zur Namensvervollständigung. Um eine Variable in der Hierarchie zu finden, wird die Hierarchie in Breitensuche von der aktuellen Suchwurzel durchlaufen. Findet sich unter der Suchwurzel w keine passende Variable, so wird als neue Suchwurzel w‘ die übergeordenete Variable von w (d.h. der Vaterknoten) benutzt; bereits durchlaufene Teilbäume werden zuvor markiert und nicht wieder durchlaufen. Wird eine Variable gefunden, so wird diese die der neue Sichtbarkeitspunkt.




```
whex( { history.extended_defined },
{
  history.x := 1, (* history.extended.x *)
  history.y := 3,
  history.z := 2
},
{
  ...
})
```

Abbildung 37: Sichtbarkeitspunkt

Die vom Algorithmus gefundene Variable wird an den Aufrufer des Strukturevaluators (den Checker) zurückgegeben. Im Kontext des Checkers ist der Strukturevaluator dafür verantwortlich, das vom Parser im VDNObjektWrapper gelieferte Programmiersprachobjekt mit einem VDNObjekt zu assoziieren. Dieses VDNObjekt trägt der Strukturevaluator im Wrapper ein; anschließend können Checker und Prozeßevaluator auf dem VDNObjekt arbeiten.

4.3.4 Der Prozeßevaluator

Der Prozeßevaluator überprüft sowohl Variablenzugriffe als auch die korrekte Erzeugung von Nachfolgedialogen und Anfragen. Intern arbeitet er dabei auf dem Prozeßmodell, welches Aussagen über die Sichtbarkeit von VDNObjekten zwischen zwei Knoten evaluieren kann.

Die Anfragen vom Checker müssen also entsprechend dem Verfahren, welches im Kapitel Modellierung vorgestellt wurde, in dem Prozeßmodell verständliche Aussagen umgesetzt werden.

Die Schnittstelle zum Checker ist folgende:

```
performPerformerScoping( vdnWrapper, cond, nextDialog, aktDialog, aktRequestName )
:VDNObjectWrapper

performCustomerScoping( vdnWrapper, cond, aktDialog ) :VDNObjectWrapper

checkDialog( dialogWrapper, cond, aktDialog, aktRequestName, isInitialDialog ) :DialogWrapper

checkRequest( requestWrapper, cond, aktDialog, isFinalRequest ) :RequestWrapper
```

Tabelle 5: Schnittstelle des Prozeßevaluators zum Checker

Zu sehen ist, daß die Umgebung (*environment*) env beim Aufruf in dEnv (Definiertheitsumgebung) „`cond`“ und eEnv (Erzeugungsumgebung) „`nextDialog`“ getrennt ist, wobei eEnv nur beim Checken eines

Variablenzugriffs in einer Dienstleisterregel benötigt wird. Der aktuelle Dialog wird als Parameter „`aktDialog`“ beim Prüfen von Dienstleister- und Kundenregeln vom Checker übergeben, die aktuelle Anfrage „`aktRequest`“ nur bei Dienstleitsterregeln.

Intern arbeitet ein Algorithmus, welcher die Bedingung „`cond`“ in die Menge der möglichen Knotenfolgen überführt. (Beschreibung im Kapitel 3). Die oben genannten Methoden benutzen den Algorithmus zum Generieren der Knotenfolgen und prüfen darauf unterschiedliche semantische Bedingungen. Aus diesem Grund ist der Algorithmus als Funktion höherer Ordnung (*higher order function*) implementiert.
Jede Methode übergibt zwei Funktionsblöcke:
Die erste Funktion wertet die vom Algorithmus übergebene Knotenfolge aus und liefert als Rückgabewert die Sichtbarkeit des VDNObjekts auf dieser Knotenfolge. Die zweite Funktion verknüpft diese Sichtbarkeiten der unterschiedlichen Knotenfolgen zu einem Ergebnis, der Sichtbarkeit über alle Knotenfolgen. Diese Sichtbarkeit des VDNObjekts wird an die aufrufenden Methoden zurückgegeben und von ihnen interpretiert. Durch diese Vorgehensweise ist ein Algorithmus für alle vier Methoden verwendbar und enthält nur allgemeinen Code.

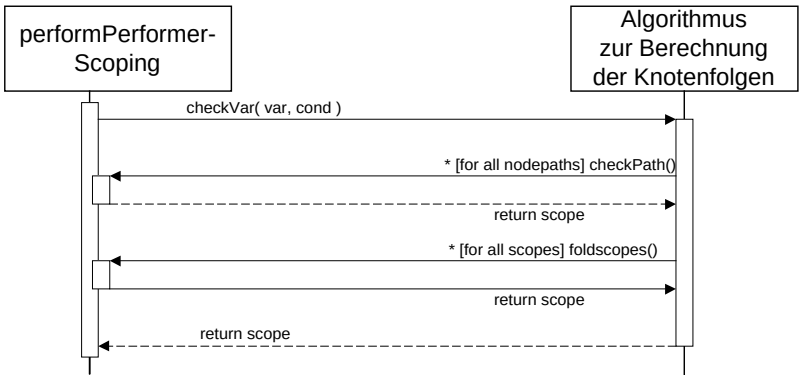


Abbildung 38: Benutzung des generischen Algorithmus zur Berechnung von Knotenfolgen im Prozeßevaluator

Anmerkung: Dieses Konzept konnte in Tycoon 2 sehr leicht implementiert werden, da Tycoon 2 im Gegensatz zu C++ und Java Funktionsabschlüsse (*function closures*) und ihre Typisierung unterstützt.

4.4 Programmiersprachen (PRL) – Konzepte

Der Parser des SET (auch genannt PRL-Parser, *PRogramming Language parser*), überprüft den Code der Dienstleister- und Kundenregeln auf Konstrukte, die zustandserweiterte Typisierung erfordern. Um dieses Vorhaben möglichst einfach zu gestalten, soll der SET-Parser nicht den Quellcode der Regeln untersuchen; vielmehr sollen Scanner und Parser des Programmiersprachencompilers den Code bereits in einen Parsebaum überführt haben, da dieser vom SET-Parser besser zu analysieren ist.
Tycoon 2 ist reflexiv und ermöglicht das Übersetzen von Quellcode zur Laufzeit. Insbesondere ist es möglich, die Codegenerierung nach dem Parsen zu unterbrechen, den Parsebaum zu analysieren, ggf. zu verändern und aus dem veränderten Parsebaum abschließend Bytecode zu generieren.

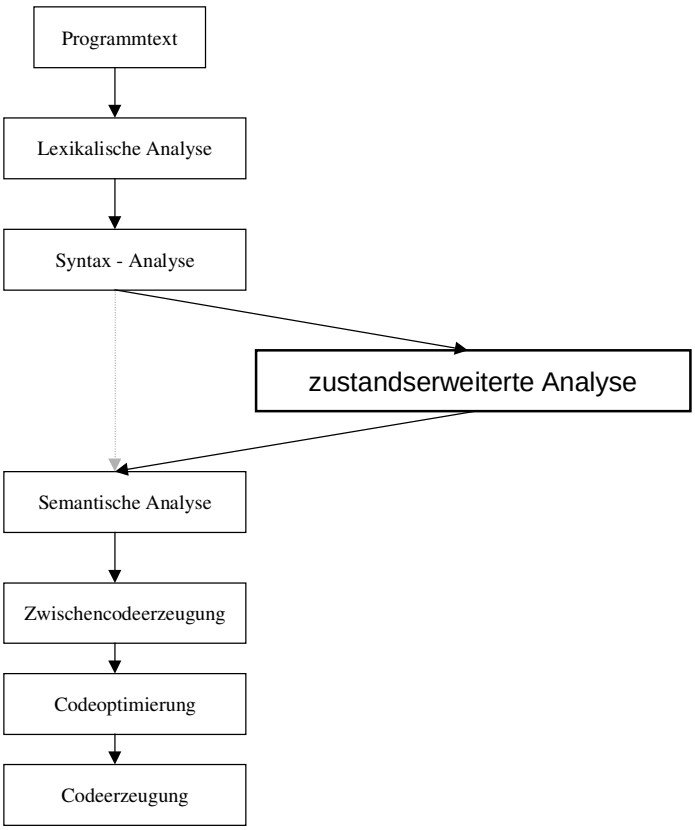


Abbildung 39: Unterbrechung des Übersetzungsprozesses

Der Code einer Regel ist in einer Methode einer Klasse implementiert. Nach dem Scannen und Parsen der Klasse liegt diese als ClassBuilder vor, in der die Methoden als Wertausdrücke (Values) abgelegt sind. Das Objektmodell der Wertausdrücke in der Programmiersprache von Tycoon 2, TL2, ist in [Wien97] ausführlich beschrieben. Um eine korrekte Analyse der SET–Konstrukte durchführen zu können, müssen diese

- in der Programmiersprache TL2 definiert werden und
- es muß analysiert werden, wie die Konstrukte im Parsebaum dargestellt sind.

Zunächst seien die Konstrukte in TL2 dargestellt:

Whex ::=	**whex** "(" BoolDefBed, *TL2Body*, *TL2Body* ")"
Fox ::=	**fox** "(" ErzBed, *TL2Body* ")"
NextDialog ::=	**nextDialog** "(" DialogName, *TL2Body* ")"
NextRequest ::=	**nextRequest** "(" RequestName, *TL2Body* ")"
VarAccess ::=	View "." *VarName*
VarReadAccess ::=	VarAccess
VarWriteAccess ::=	VarAccess „:=“
DefBed ::=	VarAccess **_defined**
BoolDefBed ::=	DefBed | ¬BoolDefBed | BoolDefBed ∧ BoolDefBed | BoolDefBed ∨ BoolDefBed

ErzBed ::=	VarAccess **_instance**
View ::=	**history** | **next** | **dialog**
Value ::=	*TL2Value* | VarReadAccess | VarWriteAccess Value | Whex | Fox | NextDialog | NextRequest
VarName ::=	*Name of a conversation variable, e.g. x or rec.y*
DialogName ::=	*Name of a conversation dialog, e.g. dialogCiao*
RequestName ::=	*Name of a conversation request, e.g. cancel*

Als nächstes ist zu zeigen, wie diese Konstrukte im TL2 Parsebaum dargestellt werden:

```
Whex( cond, plusblock, negblock ) :SendValue
    receiver        :Value          self
    selector        :Symbol         whex
    args            :List(Value)    [cond,plusblock,negblock]

Fox( cond, block )  :SendValue
    receiver        :Value          self
    selector        :Symbol         fox
    args            :List(Value)    [cond,block]

VarReadAccess       :SendValue                              history.rec.x
    receiver        :Value          enclosing value         self.history.rec
    selector        :Symbol                                 x
    args            :List(Value)    []

VarWriteAccess      :SendValue                              history.rec.x:=3
    receiver        :Value          enclosing value         self.history.rec
    selector        :Symbol                                 x:=
    args            :List(Value)    [3]

BoolDefBed                          DefBed | :NotValue | :AndValue | :OrValue <:Value

DefBed              :SendValue                              history.x_defined
    receiver        :Value          enclosing value         self.history
    selector        :Symbol                                 x_defined
    args            :List(Value)    []

:NotValue                                                   !history.x_defined
    condition       :Value                                  history.x_defined

:AndValue                           history.x_defined &&    { history.y_defined }
    condition       :Value                                  history.x_defined
    cond2           :Value                                  history.y_defined

:OrValue                            history.x_defined ||    { history.y_defined }
    condition       :Value                                  history.x_defined
    cond2           :Value                                  history.y_defined

ErzBed              :SendValue                              history.x_instance
    receiver        :Value          enclosing value         self.history
    selector        :Symbol                                 x_instance
    args            :List(Value)    []

DialogName          :LiteralValue
    value           :String                                 „dialogCiao“

RequestName         :LiteralValue
    value           :String                                 „Cancel“
```

Die Konstrukte werden im Parsebaum identifiziert und dem Checker zum Prüfen übergeben. Da der Checker und die dahinterliegenden Komponenten von den Programmiersprachobjekten abstrahieren, müssen diese vorher in Wrapper „eingepackt“ werden.

Handelt es sich bei den zu prüfenden Konstrukten um whex oder fox, so werden die Bedingungen der Konstrukte in eine Umgebung (Klasse PRLEnvironment) überführt und dieses wird mit der das whex / fox-Konstrukt umgebenden Umgebung verknüpft.
Im der Implementation sind dEnv (Definiertheitsumgebung) und eEnv (Erzeugungsumgebung) getrennt, d.h., daß Erzeugungsumgebungen, die von einem NextRequest–Konstrukt definiert werden, nicht im PRLEnvironment abgelegt, sondern separat verwaltet werden.

Beim Prüfen von Konstrukten, die im Parsebaum als Value-Objekt repräsentiert werden, wird das PRLEnvironment dem Checker als Parameter übergeben. Das zu prüfende Programmiersprachobjekt, d.h. das Value-Objekt kann durch den Checker und dahinterliegende Komponenten (insbesondere durch den Strukturevaluator) verändert werden. Abschließend wird das (veränderte) Value-Objekt nach dem Prüfen anstelle des bisherigen Value-Objekts in den Parsebaum eingehängt.

Nach der Überprüfung einer Regel wird der Parsebaum der Regel vom Parser als RuleWrapper-Objekt an den Checker zurückgereicht. Der Agentenprüfer (siehe Kapitel 5) überführt die RuleWrapperklassen abschließend in Bytecode.

5. Einbettung des SET in die TBC

Der *state enriched typechecker* SET, welcher in Tycoon 2 für die Zielsprache Tycoon 2 geschrieben wurde, sollte auf die vorhandene Implementation der *Business Conversations*, die *Tycoon 2 Business Conversations* (TBC) angewendet werden.

Die TBC-Implementierung [Wegn98] definiert Softwareagenten, die in mehreren Rollen verschiedene Konversationen führen können. Diese Agenten sollen direkt geprüft werden.

Bei der Anwendung des SET auf einen Softwareagenten wird nach folgenden Schritten verfahren:
- Analyse der Rollen des Agenten
- Extraktion der Konversationsspezifikation einer jeden Rolle
- Überführung des Contentmodells der TBC in die SET-eigene Modellrepräsentation
- Interpretation einer vorhandenen Implementation der Konversationspezifikation (d.h. einer Konversation)
- Überprüfung der darin implementierten Regeln auf Konformität mit der Spezifikation

Es sei angemerkt, daß dieses Vorgehen die bisherige (einfache) Übersetzung der Regeln ersetzt.

Diese Schritte werden von einem Agentenprüfer (*agent checker*) realisiert.
Der Agentchecker untersucht jeweils einen Agenten. Ein Agent kann in verschiedenen Konversationen verschiedene Rollen einnehmen. In jeder Rolle implementiert der Agent die Regeln (*rules*) der Konversation.

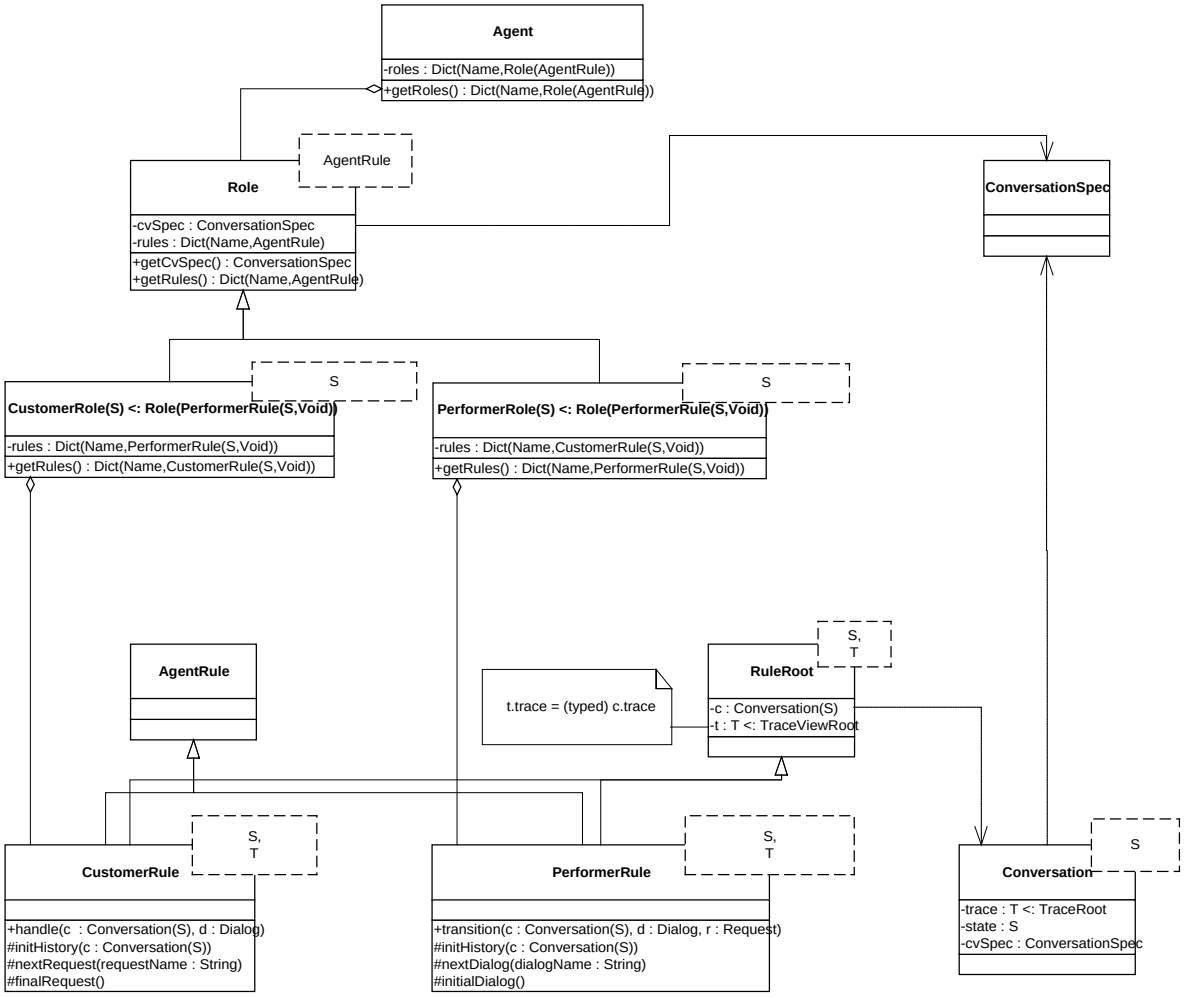


Abbildung 40: Klassendiagramm: Agent, Rollen, Regeln (*rules*) und Konversationsspezifikation

Erläuterungen zur Implementation des Agenten, der Rollen und Regeln finden sich in [Wegn98].

Vor der Beschreibung des Aufbaus des Agentenprüfers seien kurz die Probleme beim Entwurf dargestellt.

- Ein Problem der zyklischen Referenzen tritt bei der Erzeugung der Konversationsspezifikation auf: Um eine Rollen eines Agenten analysieren zu können, muß diese zunächst instanziiert und die Referenzen auf die Regeln gesetzt sein. Dazu ist es nötig, daß die Regeln als Klassen bereits übersetzt sind, d.h. als kompilierte Klassen vorliegen. Die Klassen der Regeln können aber zumeist nicht zu diesem Zeitpunkt übersetzt werden, da dieser Übersetzungsprozeß noch nicht den SET verwenden kann, die programmierten Referenzen auf Variablen der Historie aber im Code der Regeln stehen. Dieses Problem kann glücklicherweise in Tycoon 2 umgangen werden, da Tycoon 2 im Falle eines Kompilierungsfehlers dennoch (leere) Regelklassen erzeugt, auf die die Rollen verweisen können.

- Ein anderes Problem besteht darin, daß der Parsebaum des Regelcodes verändert wird, nicht aber der Quellcode. Der Klassenlader von Tycoon 2 überprüft bei jedem Compiliervorgang, ob der Quellcode dem kompilierten Bytecode entspricht. Entspricht er diesem nicht, so wird der Quellcode neu übersetzt. Da der Parsebaum der Regelklassen durch den SET verändert wird, wird damit beim Übersetzen anderer Klassen ein neuer Kompiliervorgang für den Quellcode gestartet, der fehlschlagen muß, da die Erweiterungen des SET in diesem Übersetzungsvorgang nicht vorhanden sind. Im Normalfall wäre dies kein Problem, da Übersetzungsvorgänge meist nur repetitiv vorkommen. Das Problem stellt sich dennoch, weil der Agentenprüfer meistens noch weitere Agenten überprüft (zumindest noch den Konversationspartner). Es mußte ein Verfahren gefunden werden, den automatischen Übersetzungsvorgang zu unterbinden. Tycoon 2 unterstützt dies so, daß beim Klassenlader Klassen aus der Liste der zu prüfenden Klassen ausgetragen werden können, d.h., daß diese auch bei Änderung des Quellcodes nicht neu übersetzt werden. Dieses Verfahren, die Regelklassen eines erfolgreich geprüften Agenten aus dem Kompilierprozeß herauszunehmen, heißt *freezing* (einfrieren).

- Eine dritte Aufgabe bei der Modellierung des Agentenprüfers war, ihn modular aufzubauen und damit die einzelnen Schritte des Übersetzens voneinander zu trennen.

Unter Berücksichtigung dieser Punkte wurde der Agentenprüfer (*agent checker*) folgendermaßen implementiert:

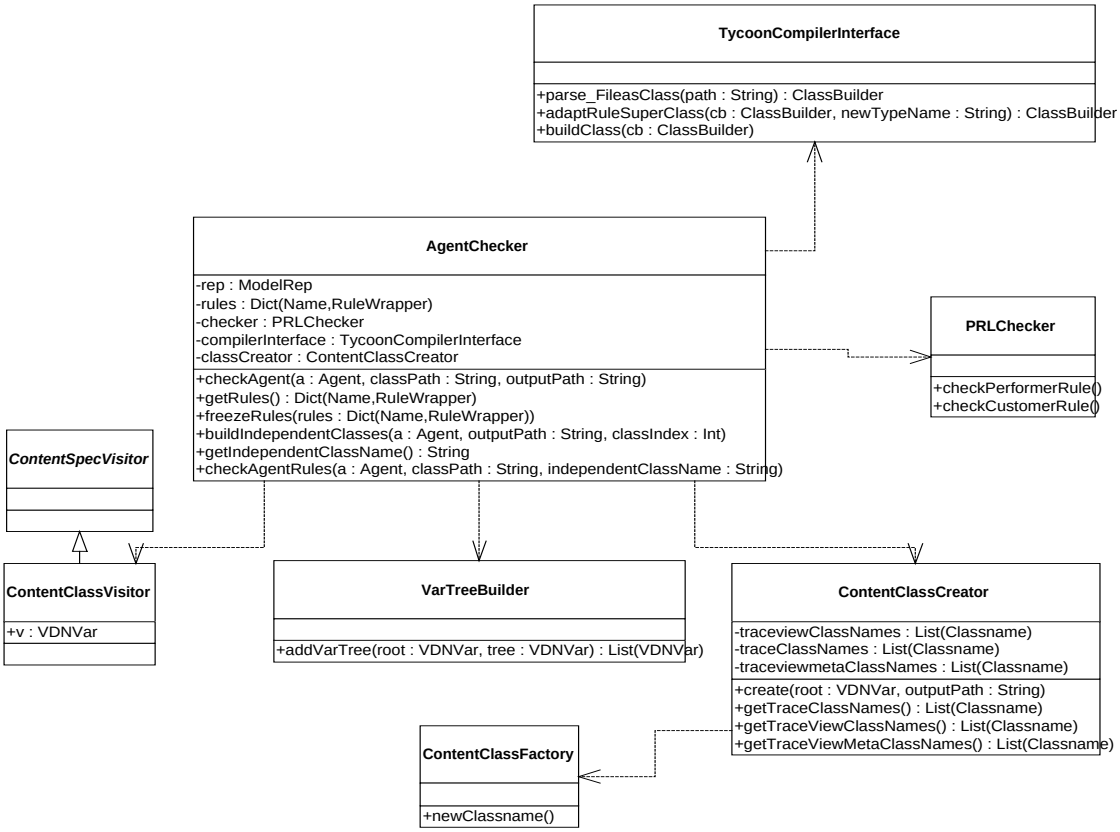


Abbildung 41: Der Agentenprüfer (agent checker) und relevante Komponenten

Der Agetenprüfer benutzt den ContentClassVisitor, VarTreeBuilder und den ContentClassCreator dazu, die Variablenstruktur der Spur (*trace*) aufzubauen und die Klassen zu generieren, und die Variablenstruktur der Modellrepräsentation zu generieren. Zunächst analysiert der Agentenprüfer die Konversationspezifikation und die darin enthaltenen Dialoge. Den Inhalt eines jeden Dialogs analysiert er über einen ContentClassVisitor. Dieser durchläuft den Variablenbaum des Dialogs und erzeugt daraus jeweils einen VDNVar-Variablenbaum. Die erzeugten Bäume werden vom VarTreeBuilder zu einem gemeinsamen Variablenbaum zusammengesetzt. Dieser Variablenbaum wird in der Modellrepräsentation gespeichert.
Durch Aufruf des ContentClassCreator werden die entsprechenden Tycoon 2-Klassen für die Laufzeitumgebung generiert. Die Namen der Klassen bildet die ContentClassFactory, welche als Singleton implementiert ist, nach der Form ContentClass<Nummer>. Die Namen beginnen bei jeder neuen Generierung bei 0.
Der Agentenprüfer übersetzt die generierten Klassen mit Hilfe des Tycoon 2-Compilers. Anschließend werden die Regeln vom SET-Checker überprüft, ihre Typisierung (*T*) an die generierten Klassen angepaßt und sie werden vom Tycoon 2-Compiler in Bytecode übersetzt. Bei einer erfolgreichen Übersetzung werden die Regeln aus der Agenda des Tycoon 2-Klassenladers entfernt und somit nicht wieder übersetzt.

Um die Regeln eines Agenten zu überprüfen, wird der Agentenprüfer mit dem Agenten, dem Klassenpfad der Regel-Quelldateien und dem Verzeichnis, in dem die generierten Klassen abgelegt werden sollen, aufgerufen:

```
checkAgent( a :Agent, classPath :String, outputPath :String )
```

Ist bei der Überprüfung kein Fehler aufgetreten, können mit

```
freezeRules( getRules )
```

die Regeln aus der Agenda des Klassenladers entfernt werden, d.h. die Quellen werden nicht durch andere Übersetzungsvorgänge, in denen der SET nicht vorhanden ist, neu übersetzt. `getRules` liefert die Liste der Regeln, die beim letzten Aufruf des Agentenprüfers überprüft wurden.

Sollen mehrere Agenten überprüfen werden, die dieselbe Konversationsspezifikation benutzen, muß der Prozeß des Klassengenerierens nur einmal durchgeführt werden. Nachdem der erste Agent mit `checkAgent()` überprüft wurde, können mit `checkAgentRules(a2,classPath,getIndependentClassName)` die anderen Agenten überprüft werden. `getIndependentClassName` liefert den Namen der Wurzel der generierten Klassen.
Es ist ebenfalls möglich, die Klassen zu einer Konversationsspezifikation zu generieren, ohne einen Agenten zu überprüfen mit:

```
buildIndependentClasses( a :Agent, outputPath :String, classIndex :Int )
```

`buildIndependentClasses` generiert die Klassen zur Konversationsspezifikation, die der Agent benutzt. Zusätzlich wird mit dem Parameter „`classIndex`“ der Basisindex für die Namen der ContentClassFactory übergeben.

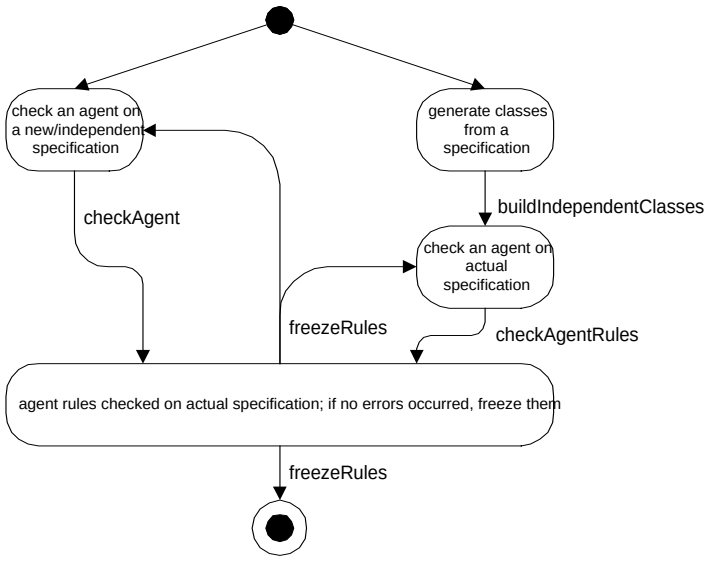


Abbildung 42: Zustandsdiagramm zur Illustration der Benutzung des Agentenprüfers (*agent checker*)

6. Vergleich von Modellierung und Implementation

Abschließend sei die Modellierung des *state enriched typecheckers* (SET) mit der Implementierung in Tycoon 2 verglichen. Dabei wird zunächst zwischen dem Modell der Business Conversation und den vom SET benötigten Anforderungen verglichen. Anschließend werden die in der Modellierung beschriebenen, aber nicht implementierten Details kurz aufgeführt. Als letztes wird nochmal kurz auf die Programmierumgebung Tycoon 2 und die Programmiersprache TL2 eingegangen.

Vergleich BC Modell und Anforderungen aus dem Modell des SET

Das Modell der Business Conversations definiert nicht, ob ein Dialog in einer Konversation mehrmals vorkommen darf. Einen Begriff, wie der durch den SET eingeführten Knoten (*node*), mit dem verschiedene Exemplare eines Dialogs unterschieden werden können, definieren die Business Conversations nicht. In der vorliegenden Implementierung der Business Conversations in Tycoon 2, den *Tycoon Business Conversations* (TBC), ist es nicht möglich, einen Dialog mehrmals innerhalb einer Konversation zu benutzen.
Das Modell des SET hat den Begriff des Knoten eingeführt. Problematisch daran ist, daß später das Modell der Business Conversations ein inhaltlich ähnliches oder gleichbedeutendes Konzept definieren könnte, ohne das Modell des SET zu berücksichtigen.

Eine weitere Anforderung an das Modell der Business Conversations von Seiten des SET ist die Möglichkeit zur Verfeinerung und Vergröberung einer Konversation. Eine Verfeinerung bzw. Vergröberung wird im Modell der Business Conversations nicht definiert; ein Konzept, welches z.T. damit vergleichbar ist, ist das Konzept der Subkonversationen.
Im SET kann bislang nicht modelliert werden, daß ein Dienstleister eine weitere, untergeordnete Konversation führt, in der er die Rolle des Kunden innehat.

Nicht implementierte Teile des SET Modells

Die Modellierung beschreibt in der strukturorientierten Evaluation mehrere Konzepte zur Auflösung von nur teilweise spezifizierten Namen. Diese Konzepte sind in der aktuellen Implementierung des SET nicht umgesetzt. Die Namensauflösung geschieht durch einfache Breitensuche im Variablenbaum und verzichtet auf Sichtbarkeitspunkt (*scopepoint*) und die Sichtbarkeit von Teilbäumen in Varianten.

Die Programmierumgebung Tycoon 2 und die Sprache TL2

Die von der Firma Higher-Order entwickelte Programmierumgebung Tycoon 2 mit der Sprache TL2 (*tycoon language*) ermöglicht dank Reflexivität und erweiterter Typisierungsmöglichkeiten ein einfaches dynamisches Einbinden von generierten Klassen zur Laufzeit. Was die Sprache von anderen reflexiven Sprachen wie z.B. Java unterscheidet, ist die Möglichkeit, in den Übersetzungsprozeß des Tycoon 2-Compilers einzugreifen, den den generierten Parsebaum zu sondieren, zu verändern und den Übersetzungsprozeß mit dem veränderten Parsebaum wieder aufzunehmen. Es existiert für Java bislang kein Compiler, mit dem dies möglich ist, d.h., daß ein SET für eine mögliche Implementierung der Business Conversations in Java einen Parser bereitstellen müßte, der Scannen und Parsen des Quellcodes selbst implementieren und aus dem veränderten Parsebaum wiederum Quellcode in Java generieren müßte. Diese zusätzlichen Aufgaben hätten die Arbeit am SET unnötig erschwert.

Anhang A

Die abstrakte SET - Sprache

VarName ::=	*Name einer Variablen (Content eines Dialogs)*
DialogName ::=	*Name eines Dialogs*
RequestName ::=	*Name eines Requests*
PerformerViewName ::=	**history** | **dialog** | **next**
CustomerViewName ::=	**history** | **dialog**

Lesender Zugriff vom Performer ::=	PerformerViewName "." VarName
Schreibender Zugriff vom Performer ::=	PerformerViewName "." VarName ":=" *Value*
Lesender Zugriff vom Customer ::=	CustomerViewName "." VarName
Schreibender Zugriff vom Customer ::=	CustomerViewName "." VarName ":=" *Value*

ViewName ::=	PerformerViewName | CustomerViewName
ViewVarName ::=	ViewName "." VarName
ViewDialogName ::=	ViewName "." DialogName
DefiniertheitEinerVariable ::=	**defined** "(" ViewVarName ")"
AustauschEinesDialogs ::=	**executed** "(" ViewDialogName ")"
BoolscherAusdruck ::=	DefiniertheitEinerVariable | AustauschEinesDialogs | ¬BoolscherAusdruck | BoolscherAusdruck ∧ BoolscherAusdruck | BoolscherAusdruck ∨ BoolscherAusdruck
Definiertheitsbedingung ::=	BoolscherAusdruck

Whex ::=	**whex** "(" Definiertheitsbedingung ")" **do** *Anweisungsblock* **else** *Anweisungsblock* **end**
Fox ::=	**fox** "(" Definiertheitsbedingung ")" **do** *Anweisungsblock* **end**
ErzeugungsBedingung ::=	**created** "(" DialogName ")"

NextDialog ::=	**nextDialog** "(" DialogName ")" **do** *Anweisungsblock* **end**
NextRequest ::=	**nextRequest** "(" RequestName ")"
Bedingung ::=	ErzeugungsBedingung | Definiertheitsbedingung | ErzeugungsBedingung ∧ Definiertheitsbedingung

Die konkrete Umsetzung der SET – Sprache in Tycoon 2

VarName ::=	*Name einer Variablen (Content eines Dialogs)*
DialogName ::=	*Name eines Dialogs*
RequestName ::=	*Name eines Requests*
PerformerViewName ::=	**history** | **dialog** | **next**
CustomerViewName ::=	**history** | **dialog**

Lesender Zugriff vom Performer ::=	PerformerViewName "." VarName
Schreibender Zugriff vom Performer ::=	PerformerViewName "." VarName ":=" *Value*
Lesender Zugriff vom Customer ::=	CustomerViewName "." VarName
Schreibender Zugriff vom Customer ::=	CustomerViewName "." VarName ":=" *Value*

ViewName ::=	PerformerViewName | CustomerViewName
ViewVarName ::=	ViewName "." VarName
ViewDialogName ::=	ViewName "." DialogName
DefiniertheitEinerVariableWhex ::=	ViewVarName **_defined**
AustauschEinesDialogsWhex ::=	ViewDialogName **_defined**
BoolscherAusdruckWhex::=	DefiniertheitEinerVariableWhex | AustauschEinesDialogsWhex | ¬BoolscherAusdruckWhex | BoolscherAusdruckWhex "&& {" BoolscherAusdruckWhex "}" | BoolscherAusdruckWhex "|| {" BoolscherAusdruckWhex "}" |
DefiniertheitsbedingungWhex ::=	BoolscherAusdruckWhex
DefiniertheitEinerVariableFox ::=	ViewVarName **_instance**
AustauschEinesDialogsFox ::=	ViewDialogName **_instance**
BoolscherAusdruckFox::=	DefiniertheitEinerVariableFox | AustauschEinesDialogsFox | ¬BoolscherAusdruckFox | BoolscherAusdruckFox "&& {" BoolscherAusdruckFox "}" | BoolscherAusdruckFox "|| {" BoolscherAusdruckFox "}" |
DefiniertheitsbedingungFox ::=	BoolscherAusdruckFox

Whex ::=	**whex** "(" DefiniertheitsbedingungWhex "," "{" *Anweisungsblock* "},{" *Anweisungsblock* "})"
Fox ::=	**fox** "(" DefiniertheitsbedingungFox "," "{" *Anweisungsblock* "})"
Definiertheitsbedingung ::=	DefiniertheitsbedingungWhex | DefiniertheitsbedingungFox | ¬Definiertheitsbedingung Definiertheitsbedingung ∧ Definiertheitsbedingung Definiertheitsbedingung ∨ Definiertheitsbedingung
ErzeugungsBedingung ::=	DialogName **_created**

NextDialog ::=	**nextDialog** "(" DialogName "," "{" *Anweisungsblock* "})"
NextRequest ::=	**nextRequest** "(" RequestName ")"
Bedingung ::=	ErzeugungsBedingung | Definiertheitsbedingung | ErzeugungsBedingung ∧ Definiertheitsbedingung

Anhang B

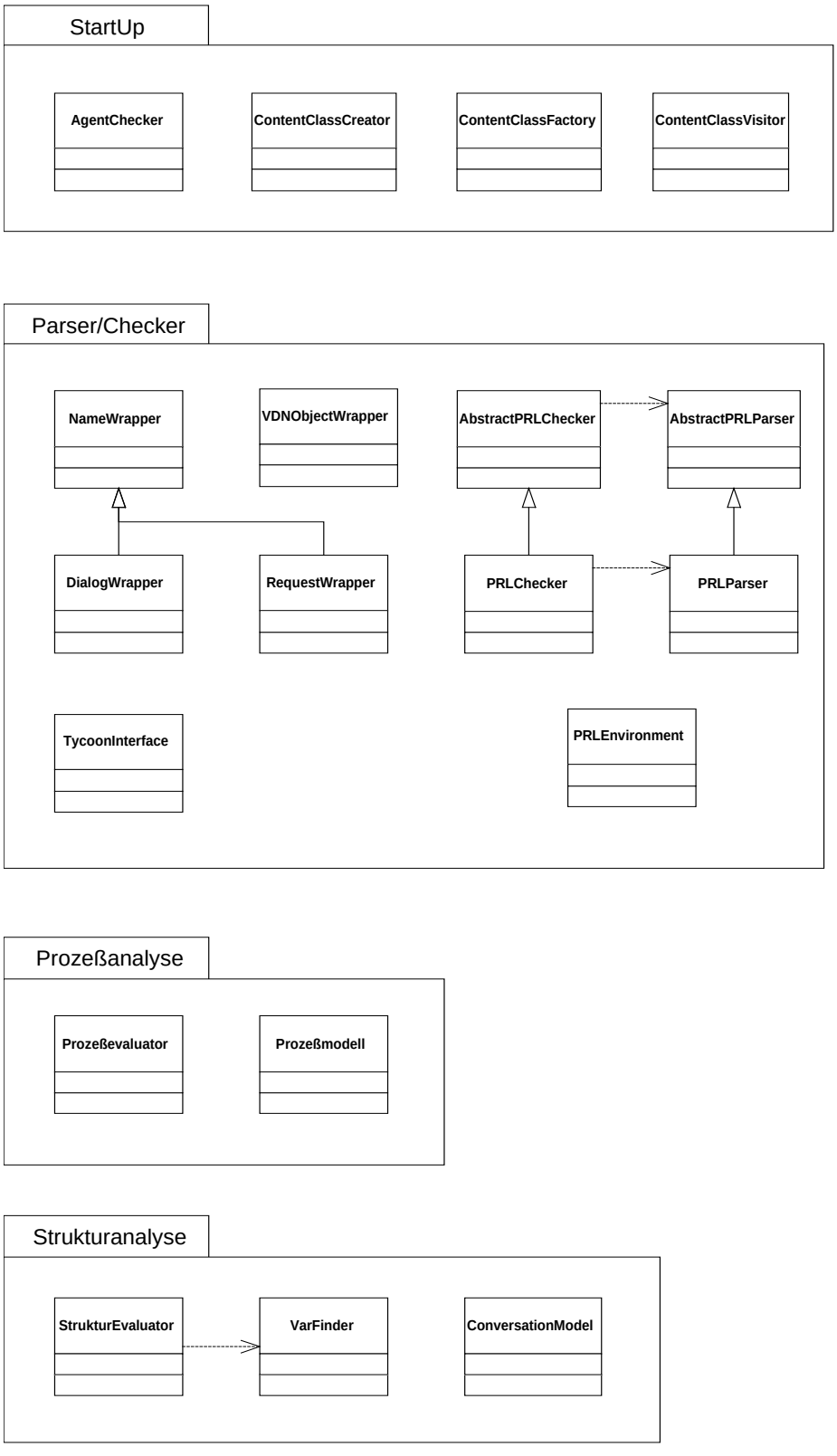
Agentenprüfer (agent checker)

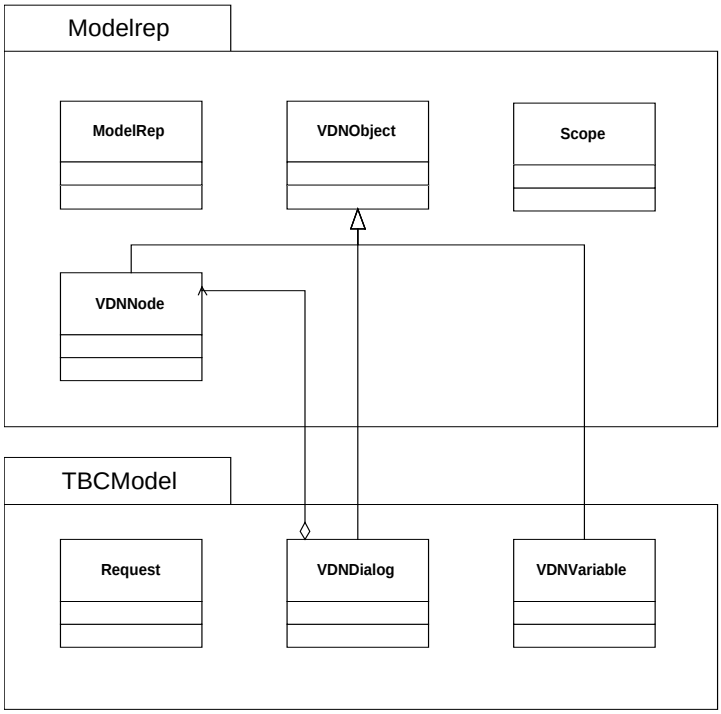
```
(* AgentChecker *)

class AgentChecker
super Object
metaclass SimpleConcreteClass( AgentChecker )


public methods

checkAgent( a :Agent, classpath :String, outpath :String ) :Void
{
  _path := classpath,
  _outpath := outpath,

  let rules :Dictionary(String,RuleWrapper) = Dictionary.new,
  ContentClassFactory.set( 0 ),

  tycoon.stdout << "[Checking agent\n",

  a.stop, (* stop agent *)
  a.getRoles.keys.do( fun( k :String )
  {
    let role :Role(AgentRule) = a.getRoles[k],

    buildClasses( role.getConversationSpec ),

    checkRole( role, rules, _classCreator.getTraceViewClassNames.head ),
    adaptRules( rules, _classCreator.getTraceViewClassNames.head )
  }),

  typeCheck,

  _rules := rules,

  tycoon.stdout << "]\n"
}


buildIndependentClasses( a :Agent, outpath :String, classIndex :Int )
:Void
{
  _outpath := outpath,

  ContentClassFactory.set( classIndex ),

  tycoon.stdout << "\n\n[Building indepent SET classes.\n",

  a.stop, (* stop agent *)
  a.getRoles.keys.do( fun( k :String )
  {
    let role :Role(AgentRule) = a.getRoles[k],

    buildClasses( role.getConversationSpec )
  }),

  tycoon.stdout << "\nBuilding independent SET classes done.]\n"
}
```

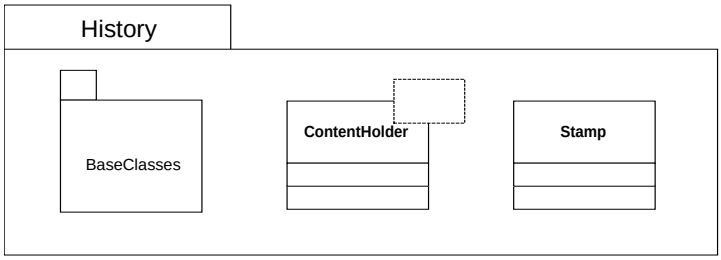
```
getIndependentClassName() :String
{
  _classCreator != nil
  ? { _classCreator.getTraceViewClassNames.head }
  : { nil }
}


checkAgentRules( a :Agent, classpath :String, independentClassName
:String ) :Void
{
  _path := classpath,

  let rules :Dictionary(String,RuleWrapper) = Dictionary.new,
  ContentClassFactory.set( 0 ),

  tycoon.stdout << "\n\n[Checking agent rules.\n",

  a.stop, (* stop agent *)
  a.getRoles.keys.do( fun( k :String )
  {
    let role :Role(AgentRule) = a.getRoles[k],

    checkRole( role, rules, independentClassName ),
    adaptRules( rules, independentClassName )
  }),

  typeCheck,

  _rules := rules,

  tycoon.stdout << "\nChecking agent rules done.]\n"
}


getRules() :Dictionary(String,RuleWrapper)
{
  _rules
}


freezeRules( rules :Dictionary(String,RuleWrapper) ) :Void
{
  unloadClasses_Dictionary( rules )
}
```

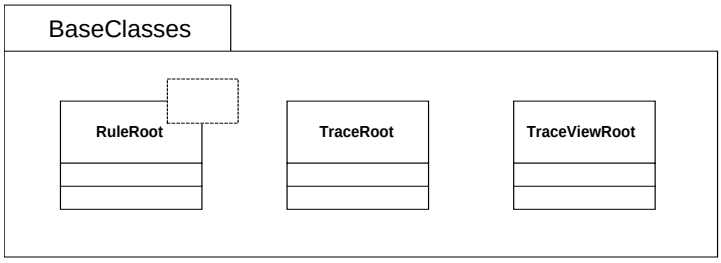
Anhang C

Paketdiagramme SET




Modelrep
ModelRep
VDNObject
Scope
VDNNode
TBCModel
Request
VDNDialog
VDNVariable


History
BaseClasses
ContentHolder
Stamp


BaseClasses
RuleRoot
TraceRoot
TraceViewRoot

Anhang D

Generierte Beispielklassen für Spur (trace) und Spursicht (traceview)

```
(* generated by Tycoon2 - state enriched typing compiler *)
(* has the following variables:

  message:String
  password:String
  subConv:ContentClass1
  name:String
*)

class ContentClass0
super TraceRoot
metaclass SimpleConcreteClass(ContentClass0)

public

message :ContentHolder(String),
password :ContentHolder(String),
subConv :ContentClass1,
name :ContentHolder(String)


methods

createViewAtStamp( name :String, upperBoundStamp :Stamp, lowerBoundStamp
:Stamp ) :TraceViewRoot
{
  ContentClass0View.new( name, self, upperBoundStamp, lowerBoundStamp )
}


private methods

_init() :Self
{
  super._init,
  message := ContentHolder(:String).new,
  password := ContentHolder(:String).new,
  subConv := ContentClass1.new,
  name := ContentHolder(:String).new,

  self
}

;
```

Codeauszug 1: Beispielklasse einer Spur (*trace*)

```
(* generated by Tycoon2 - state enriched typing compiler *)
(* yields access to the following variables:

  message :String
  password :String
  subConv :ContentClass1
  name :String
*)

class ContentClass0View
super TraceViewRoot
metaclass ContentClass0ViewClass

public

subConv :ContentClass1View

methods

"message:="( val :String ) :String
{ let result :String = nil, _writeable ? { result :=
_typedTrace.message.insert( val, _upperBoundStamp ) }, result }

message() :String
{ _typedTrace.message.lookup( _upperBoundStamp, _lowerBoundStamp ) }

message_defined() :Bool
{ _typedTrace.message.defined( _upperBoundStamp, _lowerBoundStamp ) }

message_instance() :Triple(ContentHolder(String),Stamp,Stamp)
{ Triple.new( _typedTrace.message, _upperBoundStamp, _lowerBoundStamp )
}


"password:="( val :String ) :String
{ let result :String = nil, _writeable ? { result :=
_typedTrace.password.insert( val, _upperBoundStamp ) }, result }

password() :String
{ _typedTrace.password.lookup( _upperBoundStamp, _lowerBoundStamp ) }

password_defined() :Bool
{ _typedTrace.password.defined( _upperBoundStamp, _lowerBoundStamp ) }

password_instance() :Triple(ContentHolder(String),Stamp,Stamp)
{ Triple.new( _typedTrace.password, _upperBoundStamp, _lowerBoundStamp )
}


"name:="( val :String ) :String
{ let result :String = nil, _writeable ? { result :=
_typedTrace.name.insert( val, _upperBoundStamp ) }, result }

name() :String
{ _typedTrace.name.lookup( _upperBoundStamp, _lowerBoundStamp ) }

name_defined() :Bool
{ _typedTrace.name.defined( _upperBoundStamp, _lowerBoundStamp ) }

name_instance() :Triple(ContentHolder(String),Stamp,Stamp)
{ Triple.new( _typedTrace.name, _upperBoundStamp, _lowerBoundStamp ) }
```

```
private

_typedTrace :ContentClass0

methods

_init4( name :String, trace :TraceRoot, upperBoundStamp :Stamp,
lowerBoundStamp :Stamp ) :Self
{
  super._init3( trace, upperBoundStamp, lowerBoundStamp ),
  _typedTrace := _typeCast( _trace, :ContentClass0 ),

  (* creating substructures *)
  subConv := ContentClass1View.new( name, _typedTrace.subConv,
_upperBoundStamp, lowerBoundStamp ),

  self
}

;
```

Codeauszug 2: Beispielklasse einer Spursicht (*traceview*)

```
(* generated by Tycoon2 - state enriched typing compiler *)

class ContentClass0ViewClass
super ConcreteClass( ContentClass0View )
metaclass MetaClass

public methods

new( name :String, trace :TraceRoot, upperBoundStamp :Stamp,
lowerBoundStamp :Stamp ) :ContentClass0View
{
  let instance = _new,
  instance._init4( name, trace, upperBoundStamp, lowerBoundStamp ),

  instance
}

;
```

Codeauszug 3: Metaklasse der Spursicht (*traceview*)

Literaturverzeichnis

[BS92] Model-checking for Content-free Processes, Olaf Burkart + Bernhard Steffen, RWTH Aachen, 92

[Cl87] Avoiding the State Explosion Problem in Temporal Logic Model Checking Algorithms, E.M.Clarke, Carnegie Mellon, 7/87

[Es91] Model checking of Persistent Petri Nets, Javier Esparza, Uni Hildesheim, 10/91

[Es92] Model checking using net unfoldings, Javier Esparza, Uni Hildesheim, 10/92

[GHJV95] Design Patterns, E. Gamma, R. Helm, R. Johnson, J. Vlissides, Addison Wesley, 1995

[Hi87] Design Verification of Sequential Machines Based on a Model Checking Algorithm of e-free Regular Temporal Logic, Hiromi Hiraishi, Carnegie Mellon, 9/88

[Joh97] Eine Umgebung für mobile Agenten, Nico Johannisson, Universität Hamburg, 1997

[Mat97] Business Conversations, A High-Level System Model for Agent Coordination, Florian Matthes, TU Harburg, 1997

[Richt97] Vergleich objekt- und agentenbasierter Datenbankprogrammierung am Beispiel der Kopplung autonomer Internet WebsiteProfiler, Universität Hamburg, 1997

[Ripp98] Verbesserung der Lokalität und Wiederverwendbarkeit von Geschäftsprozeßspezifikationen, Universität Hamburg, 1998

[St95] Finite Model Checking and Beyond, Bernhard Steffen, Universität Passau, Jan 1995

[Wahl98] Entwurf einer objektorientierten Sprache mit statischer Typisierung unter Beachtung kommerzieller Anforderungen, Jens Wahlen, Universität Hamburg, Juni 1998

[Wegn98] Objektorientierter Entwurf und Realisierung eines Agentensystems für kooperative Internet-Informationssysteme, Universität Hamburg, 1998

[Wien97] Bootstrap einer persistenten objektorientierten Programmierumgebung, Universität Hamburg, 1997