Fraunhofer
IESE

Komposition von Benutzungsmodellen: Spezifikation und Testtechniken

Autoren:
Thomas Bauer
Robert Eschbach
Tanvir Hussain
Johannes Kloos
Fabian Zimmermann

IESE-Report Nr. 114.09/D
Version 1.0
Januar 2009

Eine Publikation des Fraunhofer IESE

Das Fraunhofer IESE ist ein Institut der Fraunhofer-Gesellschaft.
Das Institut transferiert innovative Software-Entwicklungstechniken, -Methoden und -Werkzeuge in die industrielle Praxis. Es hilft Unternehmen, bedarfsgerechte Software-Kompetenzen aufzubauen und eine wettbewerbsfähige Marktposition zu erlangen.

Das Fraunhofer IESE steht unter der Leitung von
Prof. Dr. Dieter Rombach (geschäftsführend)
Prof. Dr. Peter Liggesmeyer
Fraunhofer-Platz 1
67663 Kaiserslautern

Komposition von Benutzungsmodellen: Spezifikation und Testtechniken

Thomas Bauer
Robert Eschbach
Tanvir Hussain
Johannes Kloos
Fabian Zimmermann

4. Februar 2009

Inhaltsverzeichnis

1 Modellierung von kompositen Modellen

In diesem Dokument soll die Beschreibung der kompositen Modelle im Vordergrund stehen; die Beschreibung der Kompositionsoperatoren findet sich in [BEH+09b].

Aufgrund der Anforderungen aus [BEH+09a] haben wir uns für einen Ansatz entschieden, der Markov-Modelle und komposite Modelle als Black Boxes behandelt. Die Beschreibung eines kompositen Modells besteht dabei aus der Angabe, welche Teilmodelle (Markov-Modell oder komposit) verwendet werden, und welche Kompositionsoperatoren in welcher Art verwendet werden sollen, um das Gesamtmodell zu ermitteln. Zusätzlich werden Interface-Bedingungen und weitere technische Informationen für die Durchführung der Komposition mitbeschrieben.

Im Folgenden soll zunächst detaillierter beschrieben werden, welche Komponenten zu einer Modellbeschreibung gehören (Kapitel 1). Dort wird zunächst an einem Beispiel demonstriert, wie ein Kompositionsoperator Teilmodelle zu einem Gesamtmodell zusammensetzt. Anschließend wird die Bedeutung von Modell-Interfaces erläutert. Aufbauend darauf werden erste Beispiele für eine konkrete Modellbeschreibung gegeben. Darauf aufbauend wird in Kapitel 2 die Modell-Beschreibungssprache im Detail spezifiziert. Die Beschreibung der Sprache zur Implementierung der Operatoren wird in Kapitel 3 und dem darauf folgenden Kapitel beschrieben.

In unserem Ansatz sollen Markov-Benutzungsmodelle aus einfacheren Benutzungsmodellen kombiniert werden. Wir haben uns entschieden, folgenden Ansatz zu wählen: Es gibt zwei Arten, ein Benutzungsmodell darzustellen. Die eine Art ist die direkte Darstellung als Markov-Kette mit Annotationen, wie sie beispielsweise vom Werkzeug JUMBL (s. [**?**]) verwendet wird. Solche Modelle nennen wir *atomare Modelle*. Andererseits gibt es sogenannte *komposite Modelle*. Diese Modelle referenzieren weitere, einfachere Modelle, und beschreiben, wie aus den einfacheren Modellen ein neues Modell zusammengesetzt wird.

Ziel des Kompositionsvorgangs ist es, ein komposites Modell als annotierte Markov-Kette darzustellen. Dazu definieren wir in diesem Dokument eine Sprache, die zur Beschreibung von kompositen Modellen angewendet wird. Diese Sprache benutzt Kompositionsoperatoren, um aus den einfacheren Modellen das komposite Modell zu konstruieren; die Spezifikation und Semantik der Kompositionsoperatoren wird in 4 detailliert beschrieben, während ein Katalog solcher Operatoren in [BEH+09b] zusammengestellt wird.

Unser Ansatz, die Komposition von Modellen mit Hilfe von Kompositionsoperatoren durchzuführen, funktioniert folgendermaßen: Wir betrachten Benutzungsmodelle im Wesentlichen als Black Boxes, die eine Folge von Stimuli ausgeben. Ein Kompositions-

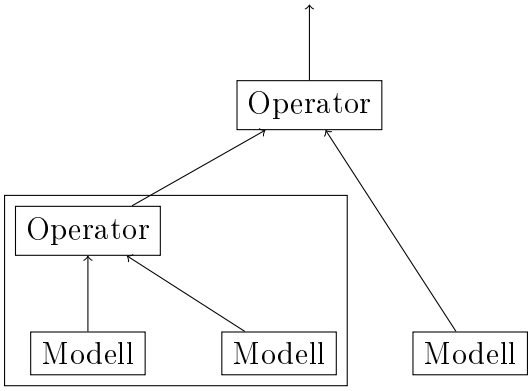

Abbildung 1.1: Konzept der Komposition. Wir betrachten Markov-Ketten als Black Boxes, die Stimuli erzeugen, und Kompositionsoperatoren als Black Boxes, die Stimuli auf einem oder mehreren Kanälen empfangen und daraus neue Stimuli erzeugen, die sie ausgeben. In diesem Bild wird die Verknüpfungsstruktur zwischen Operatoren und Modellen dargestellt; durch den Rahmen wird dargestellt, dass sich ein Kompositionsoperator zusammen mit seinen Eingabemodellen wiederum als Modell sehen lässt. Die Pfeile stellen Stimulus-Ströme dar.

operator ist nun ein Prozess, der einen oder mehrere Eingänge besitzt, auf denen er Folgen von Stimuli empfangen kann, und einen Ausgang, auf dem eine sich daraus ergebende Folge von Stimuli ausgegeben wird. Ein Kompositionsschritt wird durchgeführt, indem ein Kompositionsoperator ausgewählt und jeder seiner Eingänge über einen Kommunikationskanal mit einen Benutzungsmodell verknüpft wird. Auf diese Art und Weise erhält man ein System von Prozessen, das eine Folge von Stimuli ausgibt, mithin wieder ein Benutzungsmodell (vgl. Abbildung 1.1).

Wenn wir nun atomare Modelle und Kompositionsoperatoren als Knoten eines Graphen sowie die Kommunikationsverbindungen als gerichtete Kanten entlang des Datenflusses, so fordern wir, dass dieser Graph ein Baum ist. Auf diesem Weg können wir sicherstellen, dass ein komposites Modell durch schrittweise Transformation einzelner kompositer Modelle zu annotierten Markov-Ketten in die Form einer annotierten Markov-Kette gebracht werden kann.

Im Folgenden soll gezeigt werden, wie man komposite Modelle spezifizieren kann. Dazu führen wir zuerst eine Spezifikationssprache an Beispielen ein, geben anschließend ihre Syntax an und beschreiben abschließend ihre Semantik.

2 Ein Entwurf für die Spezifikationssprache

Wir haben uns für folgende Methode zur Modellierung von kompositen Modellen entschieden: Für jedes erzeugbare Modell wird eine Datei angelegt, in der dieses Modell beschrieben wird. In dieser Datei werden zunächst die zugrunde liegenden Benutzungsmodelle importiert. Anschließend wird die Durchführung der Komposition beschrieben.

Die Durchführung eines Kompositionsschrittes wird als Anwendung eines Kompositionsoperators auf eine Menge von Benutzungsmodellen beschrieben, wobei in dieser Beschreibung auch Zusatzinformationen mitgeführt werden können und sollen. Zusätzlich können Zwischenschritte als Gleichungen beschrieben werden, bei denen einem Bezeichner ein Kompositionsausdruck zugeordnet wird. Diese Gleichungen haben ausschließlich lokalen, abkürzenden Charakter.

Schließlich können bei den Kompositionsausdrücken Zusatzinformationen mit angegeben werden, die die Behandlung von Labels und Profilen beschreiben.

2.1 Einführung und Beispiele

Im Folgenden werden zur Illustration der Kompositionssprache einige Beispiele angegeben. Wir benutzen folgende Notationen:

- Schlüsselwörter sind `in Schreibmaschinenschrift` geschrieben.
- Kompositionsoperatoren sind *kursiv* geschrieben.

Ein einfaches Beispiel für eine Komposition könnte so aussehen:

```
model TestmodellAbstandhalter;
// Das Interface dieses Modells wird automatisch ermittelt:
// Die Menge der erzwingbaren Stimulus-Folgen ist die vom Modell erzeugte
// Sprache. Die Menge der sichtbaren Stimuli ist mit der Menge
// aller erzeugten Stimuli identisch
import VorausfahrendesFahrzeug from "Fahrzeug.tml";
import FahrerMitAssistent; // bereits erzeugtes weiteres Modell
// Der Name Fahrer soll später anderweitig verwendet werden
import Fahrer as FahrerOhneAssistent;
import UmschaltungAssistent as Umschalter;
// Fehler-Modell für Assistent: Signal wird nicht empfangen
import AssistentSignalverlust as Fehler;

// Zwischenberechnung:
// Während der Benutzung kann aufgrund von Stimuli, die von Umschalter
// erzeugt werden, zwischen beiden Fahrermodellen gewechselt werden.
Fahrer = switch(Umschalter, [
  ASSISTENT_AN =>FahrerMitAssistent,
  ASSISTENT_AUS =>FahreOhneAssistent
]);

// Endgültiges Modell:
// Die Eingaben des oben modellierten Fahreres werden
// mit einem Modell des vorausfahrenden Fahrzeugs parallel komponiert,
// wobei wir den Verlust von Messungen im Abstandshalter-Assistenten
// mit einmodellieren.
result weightedParallel([
  Fahrer =>0.8,
  dropStimuli(Fehler, VorausfahrendesFahrzeug) =>0.2
]);
```

2.1.1 Modell-Interfaces

Im Allgemeinen sollen Modelle als reine Black Boxes behandelt werden. Allerdings sind für bestimmte Kompositionsoperatoren etwas detailliertere Aussagen über das Verhalten der Modelle nötig. Als Beispiel dafür betrachten wir den Zustands-Verfeinerungs-Operator:

Eingaben: Zwei Markovmodelle, das Hauptmodell `H` und das Verfeinerungsmodell `V`, eine Trace-Sprache `T`, eine Funktion, die Eingangsfunktion `in`, und ein Default-Stimulus `out`.

Ergebnis: Es wird ein auf `H` basierendes Markovmodell erzeugt. Allerdings wird jeder

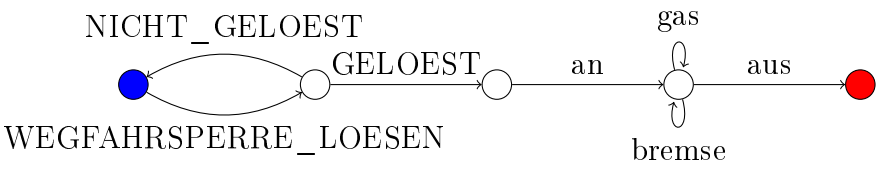

Abbildung 2.1: Benutzungsmodell: Ein (stark vereinfachter) Fahrer.

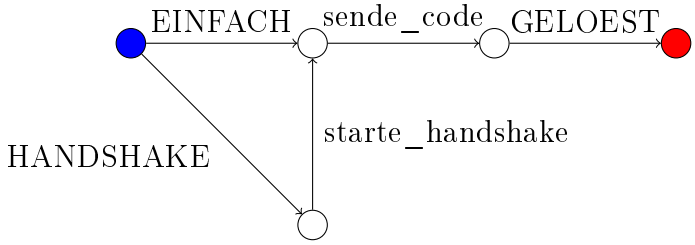

Abbildung 2.2: Benutzungsmodell: Ein Schlüssel, der mit einem Wegfahrsperren-Code-Chip ausgerüstet ist.

Zustand `z`, der durch einen Trace aus `T` erreicht werden kann, durch ein Submodell ersetzt. Dies geschieht folgendermaßen:

Es wird eine Kopie von `V` erzeugt, die wir im Folgenden mit `K` bezeichnen. Nun wird für jede eingehende Kante von `z` zunächst bestimmt, welcher Stimulus an dieser Kante steht. Dann bestimmen wir das Bild des Stimulus unter `in`und erhalten eine Folge `f` von Stimuli.

Diese Folge benutzen wir nun, um einem Pfad von Startzustand von `K` zu folgen. Der Endzustand dieses Pfades ist dann das neue Ende der betrachteten Kante.

Andererseits betrachten wir alle Kanten, die zum Endzustand von `K` führen. Für jede dieser Kanten bestimmen wir den zugehörigen Stimulus. Gibt es eine Kante von `z` mit diesem Stimulus, so ist der Zielzustand dieser Kante das neue Ende der Kante zum Endzustand. Ansonsten wird die Kante genommen, die `z` mit dem Stimulus `out` verlässt.

Wir veranschaulichen diesen Operator an einem Beispiel: Betrachte das (einfache) Benutzungsmodell in Abbildung 2.1, das einen Fahrer beschreibt. Hier findet sich eine Transition mit Stimulus WEGFAHRSPERRE_LOESEN; diese Transition kann für verschiedene Arten von Wegfahrsperre passend ersetzt werden. Vom Zustand, der mittels dieser Transition erreicht werden kann, führen zwei Kanten weg, nämlich eine mit Stimulus GELOEST und eine mit NICHT_GELOEST. Diese beiden Stimuli sind virtuelle Stimuli, die den Ausgang des Lösevorgangs beschreiben.

Nun können wir annehmen, dass es drei Arten von Möglichkeiten gibt, die Wegfahrsperre zu implementieren:

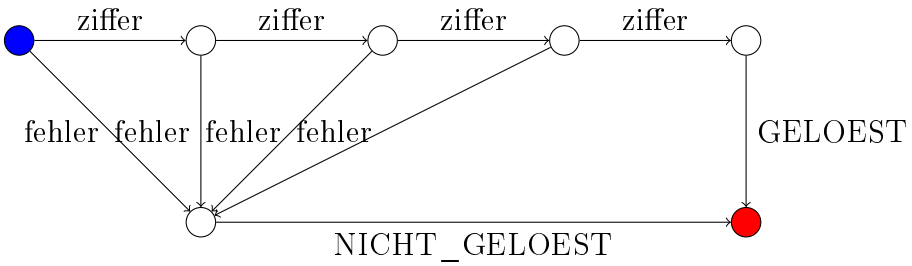

Abbildung 2.3: Benutzungsmodell: Codeeingabefeld

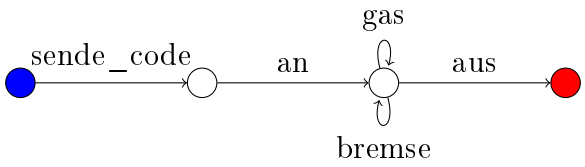

Abbildung 2.4: Benutzungsmodell: Das Ergebnis einer Komposition

1. Im Schlüssel befindet sich ein Chip, der auf ein Funksignal vom Fahrzeug einen Code ausgibt. Ist dieser Code korrekt, wird die Wegfahrsperre gelöst.

2. Im Schlüssel befindet sich wieder ein Chip. Allerdings muss das Fahrzeug einen Handshake vornehmen, damit der Schlüssel einen Code sendet.

3. Der Benutzer muss einen vierstelligen Code auf einem Tastenfeld eingeben, um das Fahrzeug starten zu können. Hier sind Varianten denkbar, bei denen ein kürzerer Code verwendet wird.

Da sich beide Schlüssel-Varianten ähnlich verhalten, modellieren wir beide in einem gemeinsamen Modell (vgl. Abbildung 2.2). Wiederum haben wir zwei virtuelle Stimuli verwendet, EINFACH und HANDSHAKE, die beschreiben, ob Variante 1 oder 2 gewählt werden soll. Wir legen hier fest, dass der zurückgegebene Code immer der richtige ist – deswegen gibt es auch nur eine Transition SENDE_CODE. Für Variante 3 findet sich das Modell in Abbildung 2.3.

Um nun Variante 1 nachzubilden, benutzen wir eine Verfeinerung wie folgt: Wir verfeinern alle Zustände im Fahrer-Modell, die durch eine Transition mit Stimulus WEGFAHRSPERRE_LOESEN erreicht werden können, wie folgt durch das Schlüssel-Modell:

- Der Stimulus WEGFAHRSPERRE_LOESEN wird auf die Stimulusfolge EINFACH abgebildet; auf diese Art steht jetzt an Stelle des ersetzten Zustands des Fahrermodells der Zustand CodeEmpfangen des Schlüssel-Modells.

- Das weitere Verhalten wird durch den Stimulus GELOEST am Ende des Schlüssel-Modells angegeben. Da es sich bei GELOEST um einen virtuellen Stimulus handeln

soll, werden die Testskripte vom Schlüsselmodell übernommen, um diesen Stimulus zu implementieren.

Das resultierende Modell findet sich in Abbildung 2.4.

Variante 2 kann analog nachgebildet werden, indem WEGFAHRSPERRE_LOESEN auf CODE abgebildet wird.

Für Variante 3 verfeinern wir analog wie bisher: Wir legen als Ausgangsfunktion die Identität fest (da die Stimuli, die vom Tastenfeld erzeugt werden, bereits passend sind, ist keine Adaption nötig), und als Eingangsfunktion die Abbildung, die WEGFAHRSPERRE_LOESEN auf die leere Sequenz abbildet. Will man stattdessen eine zwei Ziffern lange Code-Sequenz verwenden, so kann man stattdessen WEGFAHRSPERRE_LOESEN auf die Sequenz ziffer.ziffer abbilden.

Bei dieser Betrachtung des Verfeinerungs-Operators stellt man fest, dass es Situationen geben kann, bei denen nicht jede Stimulusfolge, die zur Herstellung eines Einstiegs in ein Submodell verwendet werden kann, sinnvoll ist. Aus diesem Grund sollte es möglich sein, an einem Modell zu annotieren, welche solchen Sequenzen zulässig sein sollen. Wir nennen solche Sequenzen *erzwingbar*.

Eine andere mögliche Eigenschaft eines Modells ist, welche daran annotierten Stimuli tatsächlich an das zu testende System weiterzureichen sind, und welche nur für die Synchronisation mit anderen Modellen benutzt werden. Stimuli, die wir nur für die Synchronisation mit anderen Modellen verwenden, nennen wir, wie oben bereits angedeutet, *virtuelle Stimuli*, während solche, die in einem Testfall auftauchen dürfen, *konkrete Stimuli* genannt werden.

Aus diesem Grund ist es möglich, ein Modell mit Interface-Informationen zu versehen. Das Interface umfasst Beschreibungen, welche Stimuli virtuell und welche konkret sind, sowie eine Angabe, welche Stimulus-Sequenzen erzwingbar sind.

Ein Beispiel für solche Operationen sehen wir hier dargestellt:

```
// Ein Modell eines Fahrers:
model SpeziellerFahrer
// Alle Stimuli sind konkret
 concrete all
// und nur die leere Sequenz erzwingbar
 forceable epsilon;
// Importiere einen Default-Fahrer
import Fahrer;
// und den Schluessel.
// Die erzwingbaren Sequenzen des Schlüssels sind
// CODE und EINFACH, beides virtuelle Stimuli.
import Schluessel;

// Die Instanziierung von Variante 2, wie oben beschrieben.
FahrerInstanz1 = refine(Fahrer, Schluessel,
 any*."WEGFAHRSPERRE_LOESEN",
 [ "WEGFAHRSPERRE_LOESEN" =>"CODE" ],
 [ "CODE_ANTWORT" =>"GELOEST" ]);

// Eine fehlerhafte Verfeinerung: "EINFACH"."GELOEST"
// ist nicht erzwingbar.
FahrerInstanz2 = refine(Fahrer, Schluessel,
 any*."WEGFAHRSPERRE_LOESEN",
 [ "WEGFAHRSPERRE_LOESEN" =>"EINFACH"."GELOEST" ],
 [ "CODE_ANTWORT" =>"GELOEST" ]);

result FahrerInstanz1;
```

Wir erlauben auch, Interfaces bei Zuweisungen und beim Import mit anzugeben. Eine Bedingung an das angegebene Interface ist, dass es strenger ist als das bisherige, das heißt:

1. Die neue Menge der konkreten Stimuli ist eine Teilmenge der bisherigen konkreten Stimuli und

2. die neue Menge der erzwingbaren Sequenzen ist Teilmenge der bisherigen.

Der Sinn dieser Operation ist ein doppelter:

1. Zum Einen erlauben wir, auf diese Art und Weise Interfaces einzuschränken bzw. festzulegen. Das ist insbesondere dann nützlich, wenn man eine annotierte Markov-Kette aus einer TML-Datei importieren will.

2. Zum Anderen können auf diese Art Interface-Bedingungen für Modelle dokumentiert werden: Wenn ein Modell an einer Komposition teilnehmen soll, kann

auf diese Art und Weise überprüft werden, ob es die notwendigen Schnittstellen-Bedingungen erfüllt.

2.1.2 Labels und Profile

Ein weiterer Punkt, den es zu beachten gilt, ist die Behandlung von Benutzungsprofilen und Labels. Hier haben wir uns entschieden, diese für neue Modelle wie folgt aus den Eingabemodellen zu entwickeln:

- Es kann eine Konstruktionsvorschrift angegeben werden. Dazu kann mit den Befehlen `"profile"` und `"label"` ein neues Profil bzw. Label angelegt werden; auf den Befehl folgt der Name und eine Beschreibung, wie das Profil bzw. Label aus den Eingaben erzeugt wird.
- Wurde keine Angabe gemacht, so werden die Namen der Labels und Keys in den jeweiligen Eingabemodellen bestimmt. Kommt ein Name in jedem Eingabemodell vor, so wird der entsprechende Name auch im Ergebnismodell erzeugt, und das zugehörige Profil bzw. Label wird aus den Labels oder Keys der entsprechenden Eingabemodelle erzeugt.

Ein Beispiel für diese Operationen ist folgendes:

```
import FahrerMitAssistent; // bereits erzeugtes weiteres Modell
import Fahrer as FahrerOhneAssistent;
import UmschaltungAssistent as Umschalter;

Fahrer = switch(Umschalter, [
  ASSISTENT_AN =>FahrerMitAssistent,
  ASSISTENT_AUS =>FahreOhneAssistent
])
  label MiL (FahrerMitAssistent =>"MiL", FahrerOhneAssistent =>"")
  label HiL (FahrerMitAssistent =>"HiL", FahrerOhneAssistent =>"")
  label NurFahrerMitAssistent (FahrerMitAssistent =>"MiL")
  profile Landstraße (FahrerMitAssistent =>"Landstraße",
    FahrerOhneAssistent =>"NormaleStraße")
  profile Autobahn (FahrerMitAssistent =>"Autobahn",
    FahrerOhneAssistent =>"Autobahn")
;
```

2.2 Syntax

Model ::= `"model"` IDENTIFIER Interface* `";"` Import+ Definition* `"result"` Composition `";"`

Definition ::= IDENTIFIER `"="` Composition `";"`
IDENTIFIER wird als Abkürzung für Composition definiert

Import ::= `"import"` IDENTIFIER ImportAlternatives Interface* `";"`
Importiere Modell mit angegebenem Interface

ImportAlternatives ::=
Der Import kann vorgenommen werden entweder ohne Änderungen...
| `"as"` IDENTIFIER
... oder mit Umbenennung eines vorhandenen Modells...
| `"from"` STRING
... oder durch Laden eines Markov-Modells aus einer Datei, wobei der Name dann hier festgelegt wird.

Interface ::= `"forcable"` Language `";"`
Welche Strings können als erzwungene Pfade verwendet werden?
| `"outputs"` `"("` IDENTIFIER (`","` IDENTIFIER)* `")"`
Welche Stimuli werden ausgegeben?
| `"showingThrough"` IDENTIFIER (`"yes"` | `"no"`)
Wird ein Label auch bei Reaktionen (modell-interne, virtuelle Transitionen) ausgegeben?

Composition ::= IDENTIFIER
Referenz auf bekanntes Modell
| IDENTIFIER `"("` Argument (`","` Argument)* `")"` Extras*
Operatoraufruf (die Menge der verfügbaren Operatoren wird dynamisch festgelegt, s. unten)
| Composition Interface
Interface-Verfeinerung

Argument ::= `"["` VectorElement (`","` VectorElement)* `"]"`
Vektoren werden auf die angegebene Weise definiert
| AtomicArgument

AtomicArgument ::= Composition
Verschachtelte Kompositionen sind möglich

| Rate

Übertragungsrate

| IDENTIFIER

Modell-Referenz oder explizite Stimulus-Angabe

| TraceLanguage

Beschreibung einer Menge von Traces durch eine reguläre Sprache

| MealyMachine

Eine Mealy-Maschine (nicht notwendig total oder deterministisch)

| Function

Eine Funktion, die Stimuli auf Traces abbildet

VectorElement ::= AtomicArgument

| Composition "=>" Rate

Bei Verwendung dieser Notation werden statt einem sogar zwei Vektoren erzeugt. Solche Ausdrücke sind nützlich, wenn man Paare beschreiben will, beispielsweise um gewichtete Kompositionen durchzuführen oder Fallunterscheidungen zu machen. Hier besteht ein Ausdruck aus Kompositionen und Raten, ...

| Composition "=>" IDENTIFIER

... hier aus Komposition und Stimulus-Angabe...

| IDENTIFIER "=>" Composition

... und hier umgekehrt.

Extras ::= "profile" IDENTIFIER "(" ProcessNamePairList ")"

Definition eines Profiles für das Zielmodell

| "label" IDENTIFIER "(" ProcessNamePairList ")"

Definition eines Labels für das Zielmodell

ProcessNamePairList ::= ProcessNamePair ("," ProcessNamePair)*

ProcessNamePair ::= IDENTIFIER ":" STRING

TraceLanguage ::= STRING

Regulärer Ausdruck für Traces: Ein einzelner Stimulus

| TraceLanguage "*"

Die Traces der Teilsprache dürfen beliebig oft wiederholt werden

| "(" TraceLanguage ")"

Klammerung

| TraceLanguage "|" TraceLanguage

Die Traces beider Sprachen sind gleichermaßen zulässig
| TraceLanguage TraceLanguage
Traces entstehen durch Konkatenation von Worten aus der ersten und der zweiten Sprache
| `"any"`
Irgend ein Stimulus
| `"epsilon"`
Ein leerer Trace

Rate ::= NUMBER
Für Übertragungsraten gilt das gleiche wie bei der Operatordefinition: Es gibt Kanäle mit Zeitverbrauch...
| `"imm"` NUMBER
... und ohne Zeitverbrauch...
| `"imm"`
... und den Sonderfall Übertragungsrate 1, kein Zeitverbrauch.

MealyMachine ::= `"mealyMachine"` STRING
Lade eine Mealy-Maschine aus der im String angegebenen Datei

Function ::= `"["` Mapping (`","` Mapping)* `"]"`
Eine Funktion ist eine Liste von Mappings

Mapping ::= STRING `"=>"` Trace
Mapping: Paar von Stimulus und Trace

Trace ::= STRING (`"."` STRING)*
Eine durch Punkte separierte Folge von Stimuli

2.3 Semantik

Im Folgenden beschreiben wir, wie eine Modell-Beschreibung ausgewertet wird. Wie bereits oben beschrieben, ist die wesentliche Idee hinter unserer Kompositionsmethode die Darstellung von Benutzungsmodellen als Stimulus-Quellen, während Kompositionsoperatoren mehrere Stimulus-Ströme zu einem neuen Stimulus-Strom verarbeiten. Die Anwendung eines Kompositionsoperators auf eine Menge von Benutzungsmodellen (egal ob Markov-Ketten oder komposite Modelle) liefert dann wieder ein neues Benutzungsmodell, also eine neue Stimulus-Quelle.

Die Beschreibung der Semantik wird sich daher in drei Schritte gliedern:

1. Wie werden Modelle angegeben? Hierbei werden wir insbesondere darauf eingehen, wie man Modelle mit Namen versehen kann, und was die Benutzung dieser Namen

bedeutet.

2. Wie werden Kompositionsoperatoren auf vorhandene Modelle angewendet? In diesem Abschnitt wird insbesondere gezeigt, wie man die π-Kalkül-Darstellung der Markov-Ketten und Operatoren verwendet, um eine Markov-Kette aus einem Kompositionsausdruck zu gewinnen.

3. Wie wird die Spezifikationssprache interpretiert? Dieser Abschnitt erklärt die restliche Semantik der Modell-Beschreibungssprache.

2.3.1 Modelle

Das wesentliche Objekt unserer Betrachtung sind Benutzungsmodelle. Diese spielen innerhalb der Sprache zur Beschreibung kompositer Modelle eine wesentliche Rolle: Man braucht sie einerseits als Eingabe für Kompositions-Ausdrücke sowie als Endergebnis der Modellbeschreibung, andererseits können sie mit Hilfe von Kompositionsausdrücken und Importierung zugänglich gemacht werden. Intern werden Benutzungsmodelle als π-Kalkül-Ausdrücke dargestellt, wie in Kapitel 4 beschrieben.

Zunächst gibt es zwei Möglichkeiten, wie man innerhalb eines Ausdrucks ein Modell referenziert: Entweder gibt man den Namen eines benannten Modells an oder man beschreibt das Modell mit Hilfe eines Kompositionsausdrucks. In beiden Fällen wird ein Ausdruck ermittelt, der dieses Model als Stimulus-Quelle beschreibt, und an der entsprechenden Stelle eingesetzt. Wichtig ist hierbei, dass mehrfache Referenzen auf ein benanntes Modell zu mehreren Instanzen dieses Modells führen. So w+rde der Der Ausdruck `parallel(ModellA, ModellA)` beispielsweise als Ergebnis die Parallelausführung von zwei Instanzen von ModellA beschreiben. Diese Semantik wurde aus zwei Gründen gewählt: Zum einen ist sie einfacher zu implementieren, und andererseits ist nicht ganz klar, welche Semantik der mehrfachen Verwendung einer Stimulusquelle zugrunde gelegt werden soll. Eine Folgerung aus dieser Festlegung ist, dass man die Benennung eines Modells prinzipiell als eine Art Makro-Definition für den zugrunde liegenden Kompositionsausdruck ansehen kann. Die Syntax hinter diesen Ausdrücken ist in den Produktionen zu Composition festgelegt.

Weiterhin werden, wie in der Syntax beschrieben, Composition-Ausdrücke an drei Stellen verwendet: Zur Definition von benannten Modellen (Produktionen zu Definition), innerhalb komplexer Kompositionsausdrücke und zur Festlegung des Resultats der Modell-Komposition (Teilproduktion von Model: `"result"` Composition).

Schließlich muss noch angegeben werden, wie man benannte Modelle erzeugen kann. Wie bereits erwähnt, kann das einerseits durch Definitionen geschehen – dabei wird dann für die aktuelle Modell-Beschreibung ein Name festgelegt, unter dem ein Kompositionsausdruck angesprochen werden kann (lexikalischer Scope), und andererseits durch einen Import-Ausdruck (Produktionen für import). Auch beim Import ist die Namensfestlegung lexikalisch gescopet. Es gibt zwei mögliche Quellen für Modellimporte: andere komposite Modelle (die ersten beiden Produktionen von ImportAlternatives beschreiben diesen Fall) oder Markov-Benutzungsmodelle in entsprechenden Dateien (die letzte

Produktion von ImportAlternatives). In beiden Fällen wird das Modell in eine geeignete interne Darstellung (π-Kalkül-Prozess) transformiert, um weiterverarbeitet werden zu können.

Jedem Modell sind einige weitere Informationen zugeordnet:

- Eine Interface-Beschreibung. Das Interface besteht aus drei Mengen:
  1. Die Menge der erzwingbaren Pfade,
  2. die Menge der Ausgabestimuli (also der Stimuli, die nicht rein virtuell sind) sowie
  3. die Menge der Labels, die auch bei Reaktionen in die Ausgabe-Labels mitaufgenommen werden.
- Eine Menge von Wahrscheinlichkeits-Profilen.
- Eine Menge von Labels, also Skript-Bestandteilen, die die Abbildung von abstrakten auf konkrete Testfälle erlauben.

Per Default hat jedes Modell zunächst eine allgemeine Interface-Beschreibung:

- Alle Pfade sind erzwingbar,
- alle Stimuli sind Ausgabestimuli,
- kein Label wird in Reaktionen mit aufgenommen.

Man kann allerdings in Import-Ausdrücken, in Kompositions-Ausdrücken und bei der Angabe des Ergebnisses das Interface einschränken, das heißt:

- Die Menge der erzwingbaren Pfade darf verkleinert werden,
- Stimuli dürfen als virtuell erklärt werden,
- Labels dürfen als auch bei Reaktionen sichtbar deklariert werden.

Die Syntax hinter diesen Einschränkungen findet man bei den Produktionen zu Interface.

2.3.2 Kompositionsausdrücke

Nachdem wir eben beschrieben haben, wie man auf Modelle zugreifen kann, können wir nun die Semantik der Kompositionsausdrücke beschreiben. Dazu betrachten wir die Produktionen, die zum Nichtterminal Composition gehören.

Composition ::= IDENTIFIER Mit Hilfe dieser Regel wird ein bereits benanntes Benutzungsmodell referenziert (s. oben). Das Interface des benannten Modells wird pbernommen.

Composition ::= Composition Interface Diese Regel beschreibt eine Einschränkung von Interfaces: Eine der Interface-Mengen des Kompositions-Teilausdrucks wird verändert, so wie weiter oben beschrieben wurde.

Composition ::= IDENTIFIER "(" Argument ("," Argument)* ")" Extras* Kompositions-Ausdrücke dieser Form beschreiben schließlich, wie ein Kompositionsoperator angewendet wird. Die Beschreibung der Operatorauswerten wird den Rest dieses Abschnitts füllen.

Zunächst besteht ein Ausdruck, der die Anwendung eines Kompositionsoperators beschreibt, aus zwei Teilen: Der eigentlichen Operatoranwendung und der Definition von Profilen und Labels. Die Operatoranwendung wird dabei wie ein Funktionsaufruf in einer Programmiersprache dargestellt, wobei die Menge der möglichen Argument reichhaltig ist:

- Weitere Kompositonsausdrücke
- Stimulus-Erzeugungsraten
- Stimuli, wobei mit Hilfe eines Ausdrucks der Form IDENTIFIER "." IDENTIFIER ein Stimulus, der einen Trace zu einem bestimmten Ursprungsmodell vorweisen kann (vergleiche [**?**]) beschrieben werden kann. Diese Möglichkeit der Aufhebung von Mehrdeutigkeiten ist dann relevant, wenn mehrere Modell die gleichen Stimuli erzeugen können.
- reguläre Sprachen in Form regulärer Ausdrücke über Stimuli, die Mengen von Stimulus-Sequenzen beschreiben
- Mealy-Maschinen, die zur Transformation von Stimulus-Sequenzen eingesetzt werden. Diese werden mit Hilfe von Black-Box- oder State-Box-Beschreibungen im PSE-Format (Ausgabeformat von ProtoSeq) spezifiziert.
- Funktionen, die Stimuli auf Stimulus-Sequenzen abbilden. beschreiben kann man diese Funktionen als Vektoren von Paaren der Form „Stimulus =>Stimulus-Sequenz “.
- Vektoren solcher Ausdrücke
- Vektoren von Paaren; solche Vektoren werden intern in zwei Vektoren umgerechnet, nämlich den der ersten Elemente der Paare und den der zweiten Elemente.

Vektoren werden im Allgemeinen durch Klammerung mit eckigen Klammern angezeigt.

Raten werden wie in den Operator-Definitionen angegeben: Eine Rate ohne `"imm"` beschreibt die Rate einer Zeit verbrauchenden Transition (Transitionen, die zu Stimulusausgaben führen, bilden einen Teil dieser Transitionen, aber modellinterne Transitionen können auch als Zeit verbrauchend deklariert werden; der Nutzen dieser Festlegung ist es, eine Transitions-Priorisierung vornehmen zu können), während eine Rate mit `"imm"` eine

nicht zeitverbrauchende (also modellinterne) Transition beschreibt. Wird bei `"imm"` keine Zahl angegeben, wird die Rate 1 angenommen. Zur Berechnung der Wahrscheinlichkeit einer Transition werden die Raten so normalisiert, dass die Summe der Raten ausgehender Transitionen eines Zustandes 1 beträgt; wenn in einem Zustand Transitionen möglich sind, die keine Zeit verbrauchen, werden die zeitverbrauchenden Transitionen dieses Zustandes nicht ausgeführt.

Um die Argument-Typen eines Kompositionsoperators auf die Datentypen des stochastischen π-Kalküls abbilden zu können, bemerken wir, dass die Ergebnisse von Kompositionsausdrücken, reguläre Sprachen, Mealy-Maschinen und Funktionen in der oben angegebenen Form direkt in π-Kalkül-Prozessen ausgedrückt werden können. Stimuli können als Konstanten dargestellt werden, Raten sind bereits ein elementarer Datentyp des Kalküls und Vektoren von Raten, Konstanten (Termen) und Namen ebenso.

Es bleibt also noch, zu zeigen, wie man die Anwendung eines Kompositionsoperators im π-Kalkül darstellt. Dazu treffen wir folgende Festlegung: Jedes Benutzungsmodell hat einen genau einen Ausgabekanal, auf dem Stimulus-Folgen ausgegeben werden. Dieser Ausgabekanal ist das letzte Argument der entsprechenden Prozess-Definition. Ebenso hat jeder Kompositionsoperator als letztes Argument einen solchen Ausgabekanal. Außerdem hat ein Kompositionsoperator auch für jedes Argument, bei dem eine Stimulus-Quelle angegeben wird, einen Eingabekanal. Damit genügt es, zur Verbindung von Modellen mit einem Kompositionsoperator entsprechend die richtige Anzahl von Namen (d.h. Kanälen) zu erzeugen, die Modelle passend zu parametrisieren und parallel zu komponieren. Wir führen das an einem Beispiel vor:

Angenommen, wir haben zwei Benutzungsmodelle M_1 und M_2 und einen Kompositionsoperator WeightedParallel2, der zwei Modelle so komponiert, dass sie unabhängig voneinander parallel laufen. Dabei kann die Wahrscheinlichkeit, dass das erste bzw. zweite Modell den nächsten Schritt macht, angegeben werden. Das Interface des Operators sieht also aus Sicht des kompositen Modells so aus: WeightedParallel2 : Modell $\times$ Rate $\times$ Modell $\times$ Rate $\rightarrow$ Modell. Die Definition des Operators im π-Kalkül ergibt sich daraus folgendermaßen:

```
WeightedParallel2(m1: name, r1: rate, m2: name, r2: rate, out: output)
  := new t1: r1, t2: r2. replicate
    (t1! + t2! | t1? m1(x)? out(x)! + t2? m2(x)? out(x)!)
```

Die Definitionen von M_1 und M_2 sehen im π-Kalkül ungefähr so aus:
`M1(out: output) = ....`

Die Anwendung von WeightedParallel2(M_1, 0.3, M_2, 0.7) sähe als π-Kalkül-Ausdruck dann so aus:

```
new chan1, chan2. M_1(chan1) | M_2(chan2)
                | WeightedParallel2(chan1, 0.3, chan2, 0.7, out)
```

Man kann ein komposites Modell als einen verschachtelten Ausdruck dieser Form darstellen; damit ist die Semantik der Kompositionsoperation festgelegt.

Zusätzlich werden mit Hilfe des Nichtterminals Extras noch weitere Informationen über den Kompositionsprozeß beschrieben, nämlich wie die Profile und Labels des entstehenden Modells ermittelt werden sollen. Wir beschreiben, wie die Berechnung der Labels bei der Komposition vonstatten geht; die Berechnung der Profile verläuft analog.

Mit Hilfe eines Ausdrucks der Form „`"label"` IDENTIFIER `"("` ProcessNamePairList `")"` “ wird eines der Profile des durch den Kompositionsausdruck beschriebenen Modells beschrieben, wobei ProcessNamePairList eine List von Paaren der Form „IDENTIFIER `":"` STRING“ beschreibt. Diese Liste besagt, welche Labels von welchen Eingabeprozessen verwendet werden, um das angegebene Label des Ergebnisses zu konstruieren. Die speziellen Eingabeprozess-Bezeichnungen $1, $2 und so weiter geben dabei den ersten, zweiten (u.s.w.) Eingabeprozess ohne Namen an.

Bei der Berechnung des Kompositionsausdrucks werden dazu die Labels der Eingangsprozesse wie in Abschnitt 2.1.2 beschrieben umbenannt.

2.3.3 Interpretation eines kompositen Modells

Mit dem Wissen der obigen Abschnitte können wir nun angeben, wie ein komposites Modell aus seiner Beschreibung berechnet wird. Betrachten wir dazu die Syntax der Modellbeschreibung:

`"model"` IDENTIFIER Interface* `";"` Import$^+$ Definition* `"result"` Composition `";"`

Der erste Teil (`"model"` IDENTIFIER Interface* `";"`) legt den Namen und das Interface des Ergebnismodells fest; der Name ist global sichtbar. Die darauf folgenden Import- und Definitions-Ausdrücke haben die oben beschriebene Syntax. Schließlich wird das Resultat der Komposition mit dem Ausdruck `"result"` Composition `";"` festgelegt, das Interface vom Anfang des Modells am Modell annotiert und schließlich unter dem am Anfang angegebenen Namen zum Import verfügbar gemacht.

Als Abschluss können wir nun schließlich angeben, wie ein komposites Modell in eine Markov-Kette übersetzt wird: Mit Hilfe einer Tiefensuche durch den von Import-Anweisungen induzierten Modell-Definitionsbaum können wir nach und nach einen π-Kalkül-Ausdruck aufbauen, der alle relevanten Teilprozesse (die Markovketten sowie angewendete Kompositionsoperatoren beschreiben) vereinigt und ihre Verbindung mittels geeigneter Kanaldefinitionen beschreibt. Dieser Ausdruck kann dann auf Basis der Semantik der π-Kalkül-Prozesse (vgl. 3) in eine Markov-Kette übersetzt werden.

Gleichzeitig können wir beim Durchlaufen dieses Baumes bestimmen, welche Profile und Labels der ursprünglichen Markov-Modelle auf welche Weise in die Profile und Labels des Ergebnis-Modells eingehen.

3 Der stochastische π-Kalkül

In diesem Kapitel wird eine Spezifikationssprache für Kompositionsoperatoren von Markov-Modellen entwickelt. Diese Sprache wird auf einer Erweiterung des π-Kalküls (siehe [Mil99]) aufbauen, dem stochastischen π-Kalkül. Eine Variante des stochastischen π-Kalküls, auf der wir unsere Sprache aufbauen werden, ist in [KLN07] beschrieben.

3.1 Einführung

Beim π-Kalkül handelt es sich um eine formale Sprache zur Beschreibung der Kommunikation nebenläufiger Systeme. Ein System wird durch die Parallelkomposition mehrerer Prozesse beschrieben.

Ein Prozess im π-Kalkül ist immer entweder ein Null-Prozess (keine Operation) oder kann auf eine oder mehrere Weisen mit anderen Prozessen synchron kommunizieren. Bei der Kommunikation können Namen vom sendenden an den empfangenden Prozess übergeben werden. Außerdem ist es möglich, neue Namen zu erzeugen, die dann wiederum zur Kommunikation eingesetzt werden können.

3.1.1 Ein Beispiel für den π-Kalkül

Zur Erläuterung dieses Kalküls betrachten wir ein Beispiel aus [Mil99]. Dabei handelt es sich um die Beschreibung eines Mobiltelefonie-Systems, bei dem einzelne mobile Endgeräte über Funkzellen miteinander kommunizieren. Die Struktur des Systems ist in Abbildung 3.1 dargestellt.

Genau wie Milner vereinfachen wir die Struktur des betrachteten Systems, indem wir ein Endgerät und zwei Funkzellen betrachten. Allerdings wählen wir bereits eine

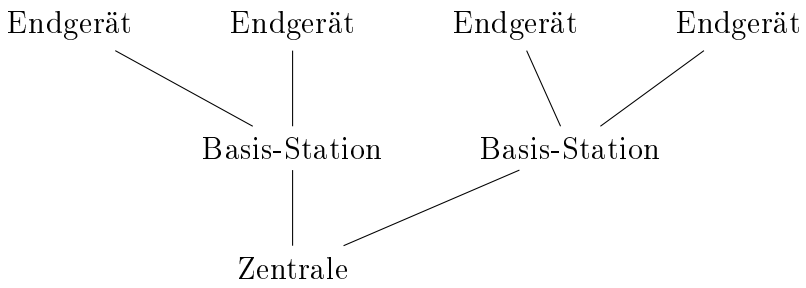

Abbildung 3.1: Struktur des Beispielsystems

Modellierung mit einem erweiterten π-Kalkül, der Muster zur Kommunikation benutzt. Die Kommunikationsstruktur des resultierenden Systems ist in Abbildung 3.1 dargestellt.

Eine Basisstation kann entweder mit dem Endgerät kommunizieren oder nicht; beide Fälle werden mit Hilfe zweier π-Kalkül-Prozesse beschrieben. Wir definieren sie mit Hilfe der folgenden Gleichungen:

```
BasisAktiv(funkstrecke, leitungZentrale) :=
  funkstrecke(SPRECHEN)? BasisAktiv(funkstrecke, leitungZentrale)
 +leitungZentrale(ABMELDEN(neuerKanel))?
    funkstrecke(MIGRIERE(neuerKanel))! BasisInaktiv(leitungZentrale);

BasisInaktiv(leitungZentrale) :=
  leitungZentrale(ANMELDEN(neuerKanel))?
    BasisAktiv(neuerKanel, leitungZentrale);
```

Die erste Gleichung bedeutet, dass eine Basisstation mit zwei Namen als Parametern instanziiert werden kann, nämlich mit der Kommunikationsverbindung zum Endgerät (`funkstrecke`) und mit dem Kanal zur Zentrale (`leitungZentrale`). Sie kann sich auf zwei Arten verhalten:

1. Sie kann vom Endgerät eine Nachricht SPRECHEN empfangen. Konkret tritt dieser Fall ein, wenn auf dem Kanal `funkstrecke` das Muster `SPRECHEN` empfangen wird. Konkret bedeutet ein Ausdruck der Form „kanal(muster)?“, dass auf dem Kanal das angegebene Muster empfangen werden soll.

   Theoretisch müsste hier jetzt eine weitere Behandlung der Daten erfolgen; wir betrachten nur das weitere Verhalten der Basisstation, die aktiv bleibt. Ihre Kanäle bleiben dabei wie bisher.

2. Sie kann von der Zentrale eine Nachricht `ABMELDEN(neuerKanal)` empfangen. Dabei ist `neuerKanal` eine Variable, die nach Empfang dieser Nachricht an den von der Zentrale übermittelten Wert gebunden wird.

   Wenn das geschieht, wird die Nachricht `MIGRIERE(neuerKanal)` an das Endgerät geschrieben. Jeder Ausdruck der Form „kanal(term)!“ beschreibt das Senden eines Terms auf einem Kanal. Die Kommunikation ist synchron, das heißt, es wird nur gesendet, wenn ein Empfänger vorhanden ist, und in diesem Fall finden das Absenden und das Empfangen der Nachricht gleichzeitig statt.

   Anschließend wird der Prozess `BasisInaktiv` mit dem Kanal `leitungZentrale` als Argument aufgerufen.

Der Operator `+` beschreibt dabei eine nichtdeterministische Auswahl. Die Definition von `basisInaktiv` verläuft in ähnlicher Weise.

Man erkennt, dass die Zentrale, wenn sie das Endgerät von `BasisAktiv` abmeldet, dem Endgerät einen neuen Namen zur weiteren Kommunikation mit einem anderen Endgerät übermittelt. Dieser Name wird dann auch der anderen Basisstation zum Zwecke der

Kommunikation übermittelt. In diesem einfachen Beispiel gibt es zwei mögliche Kanäle, die übermittelt werden:

```
Zentrale1 := basis1(ABMELDEN(kanal2))!
  basis2(ANMELDEN(kanal2))! Zentrale2;
Zentrale2 := basis2(ABMELDEN(kanal1))!
  basis1(ANMELDEN(kanal1))! Zentrale1;
```

Das Endgerät kann schließlich entweder Daten übertragen oder den Auftrag erhalten, auf einen neuen Kanal zu wechseln:

```
Endgeraet(kanal) :=
  kanal(SPRECHEN)! Endgeraet(kanal)
 +kanal(MIGRIERE(neuerKanal))? Endgeraet(neuerKanal);
```

Schließlich können wir alle diese Teilprozesse zu einem Gesamtprozess `System` zusammensetzen. Dabei ist `new` ein Operator, der an die ihm folgenden Namen neue Kanäle bindet, und `|` der Parallelkompositions-Operator.

```
System := new kanal1, kanal2, basis1, basis2.
  (Endgeraet(kanal1) | BasisAktiv(kanal1, basis1)
  |BasisInaktiv(basis2) | Zentrale1);
```

3.1.2 Eine stochastische Erweiterung des π-Kalküls

Der stochastische π-Kalkül ordnet nun jedem Namen eine Übertragungsrate (s. [KLN07]) bzw. eine Übertragungswahrscheinlichkeit zu. In [KLN07] werden auch weitere Konstrukte zur leichteren Formulierung von komplexen System sowie sogenannte immediate-Transitionen eingeführt. Wir benutzen von diesen zusätzlichen Möglichkeiten nur die Übertragungswahrscheinlichkeit und die immediate-Transitionen. Im Folgenden werden wir nun unseren erweiterten stochastischen π-Kalkül vorstellen.

3.2 Syntax des erweiterten stochastischen π-Kalküls

Im Folgenden werden wir eine reichere Version des π-Kalküls definieren, die uns die Definition der Kompositionsoperatoren erleichtern wird. Wir legen dazu zunächst eine erweiterte Definition der Syntax fest und beschreiben anschließend ihre Semantik. Die Notation orientiert sich an [KLN07], aber ist mit Elementen aus [Mil99] und eigenen Erweiterungen angereichert.

Für unsere Version des stochastischen π-Kalküls (im folgenden kurz π-Kalkül) können wir die Syntax mit Hilfe einer erweiterten BNF angeben. Wir benutzen folgende Konventionen:

- GROSS GESCHRIEBEN: Terminalsymbol

- Gemischt geschrieben: Nicht-Terminal
- `"Schreibmaschine"`: Schlüsselwort

Zunächst beschreiben wir den Aufbau eines Prozesses:

Proc ::= `"0"`
Der Null-Prozess; keine Operation
| `"("` Proc `")"`
| ChoiceProc
Alternative Verhalten, werden durch Aktions-Präfixe (Guards) unterschieden
| Proc `"|"` Proc
Parallelkomposition von einzelnen Prozessen
| `"parallel" "("` Eachexpr `")" ":"` Proc
Parallelkomposition über einen Vektor-Ausdruck
| `"new"` ChanExprs `"."` Proc
Erzeugung neuer Namensbindungen
| `"newvec"` IDENTIFIER `":"` RateVecExpr `"."` Proc
Erzeugung neuer Namensvektor-Bindungen
| Instantiation
| `"replicate"` Proc
Replizierter Prozess: Es können beliebig viele parallel Kopien des Prozesses erzeugt werden.

ChoiceProc ::= Action `"."` Proc
Ein Aktions-Präfix, gefolgt von einem Prozess, ist ein Choice-Prozess
| Action
Syntax-Zucker: Ein Aktions-Prozess, gefolgt vom Null-Prozess (der implizit ist)
| ChoiceProc `"+"` ChoiceProc
Nicht-deterministische Auswahl zwischen alternativen Verhalten
| `"sum" "("` Eachexpr `")" ":"` Action `"."` Proc
Nicht-deterministische Auswahl zu wischen alternativen Verhalten – Vektorausdruck

Action ::= IDENTIFIER `"("` TermList `")" "!"` ActionParameters
Sende eine Term-Liste auf dem angegebenen Kanal
| IDENTIFIER `"!"` ActionParameters
Sende eine leere Nachricht auf dem angegebenen Kanal
| IDENTIFIER `"("` PatternList `")" "?"`

Empfange eine Nachricht, die auf die angegebene Muster-Liste passt, auf dem Kanal

| IDENTIFIER `"?"`

Empfange eine leere Nachricht auf dem Kanal

| `"barrier"` IDENTIFIER `"."`

Empfange auf allen Kanälen des Vektors, der durch den Bezeichner angegeben ist, nacheinander eine leere Nachricht.

Instantiation ::= IDENTIFIER `"("` TermList `")"`

Instanziiere einen Prozess mit Parametern

| IDENTIFIER

Instanziiere einen Prozess ohne Parameter

Prozesse sind die grundlegenden Einheiten des stochastischen π-Kalküls. Wie wir bereits am Begriff der Instanziierung erahnen können, kann man einen Prozess auch mit einem Namen versehen und ihn dann unter diesem Namen ansprechen. Die Syntax solcher Definitionen wird wie folgt beschrieben:

Def ::= IDENTIFIER `"("` PatternList `")"` `":="` Proc `";"`

Format einer Definition: Das Interface ist ein Namen und ein Muster, die Implementierung ein Prozess.

Ein Bezeichner ist in dieser Sprache leicht zu definieren – er fängt mit einem Kleinbuchstaben an und wird von beliebig vielen Buchstaben, Ziffern und Unterstrichen gefolgt; formal sieht die Definition so aus:

$$\begin{aligned} \text{IDENTIFIER} &= \text{NONCAPITAL} \cdot \text{IDLETTER}^* \\ \text{LETTER} &= \{\texttt{"A"}, \ldots, \texttt{"Z"}, \texttt{"a"}, \ldots \texttt{"z"}\} \\ \text{IDLETTER} &= \text{LETTER} \cup \{\texttt{"0"}, \ldots, \texttt{"9"}, \texttt{"\_"}\} \end{aligned}$$

Bisher kam der stochastische Aspekt des Kalküls noch nicht zum tragen. Dies wird jetzt nachgeholt, indem wir zeigen, wie neue Namen definiert werden. Dabei kann jedem Namen eine Übertragungsrate zugeordnet werden. Zusätzlich kann bei den Übertragungsraten vermerkt werden, ob es sich um eine Zeit verbrauchende oder eine keine Zeit verbrauchende („immediate“) handelt. Solche Ausdrücke haben die folgende Form:

ChanExpr ::= IDENTIFIER

Ein Kanal kann ohne Rate definiert werden (dann wird als Übertragungsrate 1, Zeit verbrauchend, festgelegt)...

| IDENTIFIER `":"` RateExpr

... oder mit Rate

ChanExprs ::= ChanExpr ("," ChanExpr)*

RateExpr ::= RATE
Zeitverbrauchend, Rate wie im Argument
| "imm"
Kein Zeitverbrauch, Rate 1
| "imm" "(" RATE ")"
Kein Zeitverbrauch, Rate wie im Argument

Als nächstes betrachten wir die Vektor-Ausdrücke. Zur Motivation betrachten wir, wie Vektorausdrücke uns erlauben, Kompositionsoperatoren mit beliebiger Stelligkeit anzugeben. Dazu geben wir an, wie ein Operator, der eine gewichtete Parallelkomposition beschreibt, implementiert werden kann:

```
WeightedParallel(chans: input vector, rates: rate vector, out: output)
:= newvec triggers: rates. replicate (
    sum(trigger: triggers) trigger!
   |sum(trigger: triggers, channel: chans) trigger?
       channel(data)? out(data)!
  );
```

Dieser Operator erlaubt es, beliebig viele Markov-Modelle parallel zu komponieren. Eine Implementierung ohne Vektoren würde die Zahl der erlaubten Argumente hingegen a priori festlegen.

RateVecExpr ::=
Ein Vektor von Übertragungshäufigkeiten. Mit Hilfe solcher Vektoren kann man vermöge des "newvec"-Operators neue Vektoren von Namen erzeugen. Er kann angegeben werden
"[" RateExpr ("," RateExpr)* "]"
... durch eine explizite Liste...
| IDENTIFIER
... oder als Referenz auf einen vorhandenen Vektor

EachExpr ::= EachComponent ("," EachComponent)*
Beschreibung, welche Vektoren simulation durchlaufen werden sollen

EachComponent ::= IDENTIFIER ":" IDENTIFIER

Jedem Vektor (zweites Label) wird ein Label zugeordnet, an das nacheinander die Werte des Vektors gebunden werden

---

Für die Übertragung von Daten sowie die Instanziierung von Prozessen benutzen wir Listen von Mustern und Termen; ihre Syntax wird wie folgt angegeben:

TermList ::= Term ("," Term)*

Termliste: Folge von Termen, durch Kommas separiert

Term ::= IDENTIFIER

Ein Term kann ein Name sein...

| IDENTIFIER "(" ")"

... oder eine konstante Funktion...

| IDENTIFIER "(" TermList ")"

... oder eine Funktion mit Argumenten...

| STRING

... oder eine Konstante in Form eines Stimulus...

| "[" TermList "]"

... oder ein Vektor von Termen

PatternList ::= Pattern ("," Pattern)*

Pattern ::= IDENTIFIER ":" TYPE

Ein typisiertes Muster kann ein Bezeichner sein, der später gebunden wird; der gebundene Wert muss den richtigen Typ haben...

| IDENTIFIER

... oder ein Bezeichner ohne Typangabe; dann ist jeder Typ zulässig...

| "*" IDENTIFIER

... oder ein „Aufsammel-Muster“, das alle Ausdrücke matcht, für die kein genauerer Matching-Ausdruck vorhanden ist...

| STRING

... oder ein Stimulus...

| IDENTIFIER "(" ")"

| IDENTIFIER "(" TypedPatternList ")"

... oder ein Funktionsausdruck...

| "$" IDENTIFIER

... oder ein Wert, der momentan an den angegebenen Bezeichner gebunden ist (Quelle egal)...

| IDENTIFIER "$" IDENTIFIER

... oder ein Wert, der momentan an den zweiten Bezeichner gebunden ist und von dem Eingabekanal stammt, der an den ersten Bezeichner gebunden ist...
... oder ein Bezeichner, der jeden Wert des angegebenen Typs annehmen kann, außer denen, für die ein spezifischeres Muster vorliegt...
| `"_"`
... oder ein Ausdruck, der beliebige Terme matcht.

Die Semantik dieser Muster sowie die relevanten Typisierungsbedingungen werden später erklärt. Man beachte, dass es kein Vektor-Muster gibt; das ist so beabsichtigt, da wir bei Vektoren im Gegensatz zu Listen keine Längenannahmen treffen. Man kann einen solchen Vektor später durch entsprechende Vektor-Operationen weiter verarbeiten.

Als mögliche Typen legen wir fest: Es sollen Namen und Raten übertragen angegeben können, ebenso Vektoren von Namen und Raten. Zusätzlich sollen manche Namen besonders ausgezeichnet werden: Zum einen soll ein Kompositionsoperator einen oder mehrere Eingabekanäle besitzen können, und weiterhin benötigt er genau einen Ausgabekanal. Offenbar kann ein Vektor von Eingabekanälen sinnvoll sein, ein Vektor von Ausgabekanälen ist es nicht. Somit können wir die Syntax der Typangaben wie folgt festlegen:

Type ::= VectorizableType
| `"output"`
| VectorizableType `"vector"`

VectorizableType ::= `"name"`
| `"rate"`
| `"input"`

Schließlich müssen wir noch beschreiben, wie die Rückverfolgbarkeit implementiert wird. Dazu wird die Trace-Beschreibung von oben benutzt. Wir werden später noch andere Parameter für Ausgaben kennen lernen, weswegen eine komplexere Formulierung als nötig verwendet wird.

ActionParameters ::= ActionParameter*
ActionParameter ::= Trace

Trace ::= `"<"` IDENTIFIER (`","` IDENTIFIER)* `">"`
Eine Menge von Eingabekanälen wird als Auslöser benannt.

3.2.1 Typisierungs- und Bindungsregeln

Wie bereits oben angedeutet, können Bezeichner typisiert sein, und es können Objekte an Bezeichner gebunden werden. Im Folgenden werden die Bedingungen angegeben, unter welchen Voraussetzungen was an welchen Bezeichner gebunden wird; daraus ergeben sich danach die Regeln, wann ein Bezeichner gebunden ist.

Zuerst legen wir die möglichen Typen fest:

- Name, mit den Untertypen
  1. Eingabe-Kanal
  2. Ausgabe-Kanal
  3. Verbundener Kanal, zusammen mit einer Trace-Berechnungsfunktion.
  4. Anderer Name
- Vektor von Namen, zusammen mit seiner Länge
- Vektor von Eingabe-Kanälen, zusammen mit seiner Länge
- Rate
- Vektor von Raten, zusammen mit seiner Länge
- Term
- Prozess
- keine Bindung

Zusätzlich wird bei allen Typen außer „keine Bindung“ noch ein Trace mitgeführt; dieser Trace gibt an, von welchem Input-Kanal eine Bindung empfangen wurde, oder hat den speziellen Wert `"local"`, wenn eine Bindung nicht von einer Eingabe-Operation auf einem Input-Kanal stammt. Dabei ist `"local"` eine Konstante vom Typ Eingabe-Kanal.

Ein Bezeichner hat genau dann einen bestimmten Typ (außer „keine Bindung“), wenn er an ein Objekt des entsprechenden Typs gebunden ist, sonst „keine Bindung“. Wenn ein Bezeichner an ein Objekt gebunden ist, nennen wir ihn auch kurz *gebunden*.

Wir können nun die Syntax-Regeln, in denen Bezeichner auftauchen, durchgehen, und die Typisierungsbedingungen angeben:

Proc ::= ...
| `"newvec"` IDENTIFIER `":"` RateVecExpr `"."` Proc

Die neue Bindung des Bezeichners ist vom Typ „Vektor von Namen“mit der gleichen Länge wie der RateVecExpr.

Action ::= IDENTIFIER `"("` TermList `")"` `"!"` Trace

Der Bezeichner muss an einen Namen gebunden sein; dieser darf kein Eingabe-Kanal sein.

| IDENTIFIER "!" Trace

Der Bezeichner muss an einen Namen gebunden sein; dieser darf kein Eingabe-Kanal sein.

| IDENTIFIER "(" PatternList ")" "?"

Der Bezeichner muss an einen Namen gebunden sein; dieser darf kein Ausgabe-Kanal sein. Zu den neuen Namenbindungen nach Ausführung s. PatternList

| IDENTIFIER "?"

Der Bezeichner muss an einen Namen gebunden sein; dieser darf kein Ausgabe-Kanal sein.

| "barrier" IDENTIFIER

Der Bezeichner muss an einen Vektor von Namen gebunden sein; die Namen dürfen keine Ausgabe-Kanäle sein.

Instantiation ::= IDENTIFIER "(" TermList ")"

Der Bezeichner muss an einen Prozess gebunden sein. Außerdem muss die TermList zur Prozessdefinition passen (vgl. Semantik)

| IDENTIFIER

Der Bezeichner muss an einen Prozess gebunden sein.

Def ::= IDENTIFIER "(" PatternList ")" ":=" Proc ";"

Der neue Bindung des Bezeichners ist vom Typ Prozess.

ChanExpr ::= IDENTIFIER

Die neue Bindung des Bezeichners ist vom Typ Name (anderer Name)

| IDENTIFIER ":" RateExpr

ditto

RateVecExpr ::= ...

| IDENTIFIER

Der Bezeichner muss ein Vektor sein

EachComponent ::= IDENTIFIER ":" IDENTIFIER

Der zweite Bezeichner muss vom Typ Vektor von Namen/Eingabe-Kanälen sein; die Bindungen des ersten Bezeichners werden alle Namen bzw. Eingabe-Kanäle sein.

Term ::= IDENTIFIER

Der Bezeichner muss nur gebunden sein.

| IDENTIFIER "(" ")"

Funktionsnamen haben ihren eigenen Namensraum, sie brauchen keine Bindung

| IDENTIFIER "(" TermList ")"

ditto

| ...

Pattern ::= IDENTIFIER ":" TYPE

Eine Bindung an den angegebenen Bezeichner ist vom angegebenen Typ (bzw. einem seiner Subtypen)

| IDENTIFIER

Eine Bindung an den Bezeichner kann von einem beliebigen Typ sein.

| "*" IDENTIFIER

Eine Bindung an den Bezeichner kann von einem beliebigen Typ sein.

| "$" IDENTIFIER

Da hier keine neue Bindung stattfindet, muss der Bezeichner nur eine Bindung besitzen.

| IDENTIFIER "$" IDENTIFIER

Da hier keine neue Bindung stattfindet, muss der zweite Bezeichner nur eine Bindung besitzen, und der erste Identifier muss ein Name sein.

| IDENTIFIER "(" ")"

| IDENTIFIER "(" TypedPatternList ")"

Funktionsnamen haben ihren eigenen Namensraum.

| ...

Trace ::= ...

| "<" IDENTIFIER ">"

Hier muss ein Eingabe-Kanal angegeben werden.

3.2.2 Beispiele zur Nutzung der Sprachmittel

Wir betrachten im Folgenden beispielhaft ein Benutzungsmodell, das einen (stark vereinfachten) Autofahrer beschreibt. Das Verhalten des Fahrers kann wie folgt dargelegt werden:

1. Bevor andere Aktionen durchgeführt werden, schaltet der Fahrer das Auto an.
2. Die letzte Aktion, die der Fahrer ausführt, ist das Abschalten des Autos.
3. Dazwischen kann er Gas geben, bremsen, den Blinker betätigen, die Lautstärke des Radios einstellen und einen von vier Programmknöpfen am Radio betätigen.

4. Nach dem Betätigen des Programmknopfes wird immer die Lautstärke eingestellt.

5. Wurde als letztes die Bremse benutzt, so wird das Gas in 60% der Fälle und die Bremse in 40% der Fälle als nächstes betätigt. Wurde das Gas betätigt, ist das Verhältnis umgekehrt.

6. Zehn Prozent der Bedienaktionen sind Radiobenutzung, das Auto wird in weiteren fünf Prozent der Fälle ausgeschaltet, und Gas bzw. Bremse werden im Rest der Fälle benutzt.

7. Der Sender wird mit 30 Prozent Wahrscheinlichkeit gewechselt, die restlichen Radiobedienungen sind Lautstärkekontrolle.

8. Die Radio-Bedienungen sollen mittels der Terme SENDER1 bis SENDER4 und LAUTSTAERKE auf dem Kanal radio übertragen werden. Gas und Bremse werden über jeweils eigene Kanäle abgewickelt, und An- und Ausschalten wird durch die Übermittlung der Terme AN und AUS über Zündung übertragen.

9. Nach dem Anlassen wird zuerst das Gaspedal bedient, die Bremse immer erst später.

Wir modellieren dieses Verhalten mit Hilfe einiger π-Kalkül-Prozesse. Zunächst beschreiben wir die Radiobedienung.

```
Radiobedienung(ereignis) :=
  new radioFertig: imm, senderwechsel: 0.3, lautstaerke: 0.7.
    replicate radioFertig? ereignis? (senderwechsel! + lautstaerke!)
  | replicate senderwechsel?
      (  radio(SENDER1)!
        +radio(SENDER2)!
        +radio(SENDER3)!
        +radio(SENDER4)!
      ) lautstaerke!
  | replicate lautstaerke? radio(LAUTSTAERKE)! radioFertig!
  | radioFertig!;
```

Die Radio-Bedienung wird durch vier parallele Prozesse dargestellt. Der Erste beschreibt die Bedingungen, wann welche Operation durchgeführt wird. Konkret muss die letzte Radio-Operation abgeschlossen sein (`radioFertig?` und ein Fahrer-Ereignis anstehen (`ereignis?`), woraufhin ein Senderwechsel oder eine Lautstärkeanpassung ausgelöst wird. Die Wahrscheinlichkeiten werden mit Hilfe der Kanalwahrscheinlichkeiten von oben modelliert.

Der zweite Prozess beschreibt einen Senderwechsel. Mittels nicht-deterministischer Auswahl wird einer der vier Sender gewählt (hier liegt dann eine Gleichverteilung vor) und anschließend die Lautstärke-Anpassung angestoßen. Der dritte Prozess beschreibt schließlich die Lautstärkeanpassung, nach der eine weitere Radio-Bedienung möglich ist. Schließlich stößt der vierte Prozess die Radio-Benutzung an.

Der Operator "replicate" leitet einen replizierten Prozess ein. Von diesem Prozess können beliebig viele Kopien innerhalb der Parallelkomposition erzeugt werden. Die ersten drei Prozesse sind repliziert, da jeder dieser Prozesse beliebig oft ausgeführt werden kann. Eine alternative Darstellung werden wir weiter unten kennen lernen.

Alle vier Prozesse werden mit Hilfe einer Definition `Radiobedienung(ereignis)` zusammengefasst. Diese Definition gibt dem Gesamtprozess einen Namen und erlaubt, den Namen `ereignis` als formalen Parameter zu benutzen.

Als nächstes modellieren wir die Benutzung von Gas- und Bremspedal.

```
GasZuletzt := wkeit40? gas! GasZuletzt + wkeit60? bremse! BremseZuletzt;
BremseZuletzt :=
  wkeit40? Bremse! BremseZuletzt + wkeit60? gas! GasZuletzt;
GasBremse(ereignis) := ereignis? gas! new wkeit40: 0.4, wkeit60: 0.6. (
    GasZuletzt
  | replicate ereignis? (wkeit40! + wkeit60!);
```

Hier benutzen wir die Instanziierung von Prozessen, um GasZuletzt und BremseZuletzt ineinander überzuleiten. Diese beiden Prozesse beschreiben einen Zustandsautomaten, der das wechselnde Verhalten von Gas und Bremse modelliert. Der Gesamtprozess GasBremse benutzt dann Instanzen dieser Teilprozesse, um die Zufallsereignisse mit Wahrscheinlichkeit 40% und 60% in die richtigen Signale zu übersetzen.

Als letztes zeigen wir die Zusammensetzung aller dieser Teilprozesse zum gesamten Fahrermodell:

```
Schritt :=
  aktion? (aktionRadio! + aktionAus! + aktionRest!) Schritt + ende?;
Fahrer :=  zuendung(AN)!
    new aktionRadio: 0.1, aktionAus: 0.05, aktionRest: 0.85,
  ende: imm, aktion: 1.
          replicate aktion!
        | Radiobedienung(aktionRadio)
        | GasBremse(aktionRest)
        | aktionRest? zuendung(AUS)! ende!
        | Schritt
```

Hier gibt es zwei Dinge zu bemerken. Zum Einen werden Prozesse mit Parametern instanziiert. Dadurch werden die `ereignis`-Namen an die Kanäle aktionRadio und aktionRest gebunden. Zum anderen benutzen wir eine sogenannte immediate-Transition, um dem Testende-Signal eine höhere Priorität als die aktion-Signal zukommen zu lassen. Hierzu gibt es eine einfache Regel: Wenn immediate-Transitionen möglich sind, so werden keine normalen Transitionen ausgeführt.

Man könnte diese Prozesse auch etwas eleganter formulieren:

```
Schritt(schritt: channel, ende: channel, aktionen: vektor) :=
  (schritt? sum(a: aktionen): a! Schritt(schritt, ende, aktionen)
  + ende?;
```

```
Fahrer :=  zuendung(AN)!
    new aktionRadio: 0.1, aktionAus: 0.05, aktionRest: 0.85,
  ende: imm, aktion: 1.
          replicate aktion!
        | Radiobedienung(aktionRadio)
        | GasBremse(aktionRest)
        | aktionRest? ende!
        | Schritt(aktion, ende, [aktionRadio, aktionAus, aktionRest]);
```

Hier wird ein parametrisierter Prozess `Schritt` verwendet, der einen Vektor ausführbarer Aktionen annimmt. Solche Vektoren können mit den Operatoren `parallel` und `sum` bearbeitet werden. Sie verarbeiten die einzelnen Komponenten des Vektors und fügen die so entstandenen Prozesse mittels Parallel-Komposition oder Auswahl zusammen.

Der Ausdruck `[aktionRadio, aktionAus, aktionRest]` bezeichnet den Vektor, der aus den drei angegebenen Signalen besteht.

Zu guter Letzt noch ein Beispiel, bei dem die übrigen Feinheiten der Vektor-Operationen beschrieben werden:

```
WeightedCopy(in: vector, rates: rateVector, out: output) :=
  newvec selector: rates.
    sum(selection: selector): selection!
  | sum(selection: selector, channel: in):
      selection? in(data)? out(data)!.
```

4 Definition von Kompositionsoperatoren

Bevor wir nun die formale Semantik der π-Kalkül-Ausdrücke angeben, soll abschließend erklärt werden, wie man π-Kalkül-Prozesse als Kompositionsoperatoren auszeichnet, und welche Bedingungen an solche Prozesse gestellt werden.

Offenbar müssen, damit unsere Konzeption der Kompositionsoperatoren als Quelle und Senke von Stimulus-Strömen funktioniert, ein Ausgabe-Strom sowie mindestens ein Eingabe-Strom existieren. Wir legen also fest, dass ein Prozess, der einen Kompositionsoperator implementiert, mindestens ein `"input"`-Argument sowie genau ein `"output"`-Argument besitzen soll.

Weiterhin wollen wir gewisse Kompatibilitätsbedingungen von den Argumenten des Kompositionsoperators fordern können; beispielsweise soll man bei einem Refinement-Operator erreichen können, dass alle virtuellen Stimuli des zu verfeinernden Modells auch tatsächlich durch das Refinement ersetzt werden. Schließlich kann es auch sinnvoll sein, die Datentypen der Parameter noch genauer zu spezifizieren. Aus diesem Grund führen wir eine besondere Deklaration ein, die einen Prozess als Kompositionsoperator auszeichnet und die entsprechenden Annotationen zulässt:

Operator ::= `"operator"` IDENTIFIER Opparameters? Opprocessname? Opalias* Opconstraint*

Opparameters ::= `"("` Opparam (`","` Opparam)* `")"`

Opparam ::= IDENTIFIER `":"` OPTYPE

Die Parameter-Beschreibung des Operators

Opprocessname ::= `"process"` IDENTIFIER

Falls der Operator anders heißen soll als der implementierende Prozess, so kann man mit dieser Direktive den richtigen Prozess-Namen angeben.

Opalias ::= `"alias"` IDENTIFIER

Alternativer Name des Operators

Die Menge der OPTYPEs ist dabei gegeben durch

$$OPTYPE = \{\texttt{input}, \texttt{output}, \texttt{stimulus}, \texttt{rate}, \texttt{input vector}, \texttt{stimulus vector}, \texttt{rate vector}, \texttt{function}, \texttt{automaton}, \texttt{regex}, \texttt{mealy}, \texttt{trace}\}.$$

Die Semantik dieses Ausdrucks wurde im Wesentlichen schon oben beschrieben. Hier können wir nun angeben, welches Format eine Datei mit Kompositionsoperator-Definitionen hat:

Definitionfile ::= statement*
statement ::= operator
| Def

Als letztes Syntax-Element bleibt uns die Angabe der Interface-Constraints. Die wesentlichen Größen, über die in den Interface-Constraints Aussagen getroffen werden, sind Mengen von Stimuli sowie Mengen von Traces. Aus diesem Grund verwenden wir hier mengenbasierte Ausdrücke, um Constraints anzugeben. Diese haben die folgende Syntax:

Opconstraint ::= StimSet CsetRelation StimSet
Ein Constraint vergleicht zwei Stimulusmengen...
| TraceSet CsetRelation TraceSet
... oder zwei Tracemengen.
| `"each"` IDENTIFIER `":"` StimSet `"."` Opconstraint
Iteriere über alle Stimuli in der Menge; der Bezeichner wird immer an eine einelementige Stimulus-Menge gebunden
| `"each"` IDENTIFIER `":"` TraceSet `"."` Opconstraint
Iteriere über alle Traces in der Menge; der Bezeichner wird immer an eine einelementige Trace-Menge gebunden

CsetRelation ::= `"="`
Gleichheit
| `"!="`
Ungleichheit
| `"<="`
Teilmenge
| `"<"`
Echte Teilmenge
| `">="`
Obermenge
| `">"`
Echte Obermenge

| `"disjoint"`

Leerer Schnitt

| `"nondisjoint"`

Nicht-leerer Schnitt

StimSet ::= IDENTIFIER

Ein Stimulus-Set kann aus einer formalen Variablen der Operatordefinition ermittelt werden, die ein Stimulus oder Stimulus-Vektor ist, oder aus einer passenden Iterationvariablen...

| `"stimuli"` `"leaving"`? TraceSet `"in"` IDENTIFIER

..., aus den Stimuli, die die durch das TraceSet beschriebenen Zuständen des angegebenen Markov-Modells verlassen Transitionen erzeugt werden,...

| `"stimuli"` `"entering"` TraceSet `"in"` IDENTIFIER

..., aus den Stimuli, die die durch das TraceSet beschriebenen Zuständen des angegebenen Markov-Modells betretenden Transitionen erzeugt werden,...

| `"domain"` `"("` IDENTIFIER `")"`

..., aus dem Definitionsbereich der angegebenen Funktion,...

| `"visible"` `"("` IDENTIFIER `")"`

... oder aus der Menge der sichtbaren Stimuli des angegebenen Eingabe-Kanals.

TraceSet ::= IDENTIFIER

Ein Trace-Set kann aus einer formalen Variablen der Operatordefinition ermittelt werden, die ein Trace, ein regulärer Ausdruck oder ein endlicher Automat ist, oder aus einer passenden Iterationvariablen...

| `"forceable"` `"("` IDENTIFIER `")"`

... oder aus der Menge der erzwingbaren Traces des angegebenen Eingabe-Modells ...

| IDENTIFIER `"("` TraceSet `")"`

... oder aus der Bildmenge der angegebenen Funktion...

| `"sinks"` `"("` IDENTIFIER `")"`

... oder aus der Menge der Endzustände des angegebenen Eingabe-Modells...

| `"extend"` `"("` TraceSet `")"`

... oder durch Erweiterung der Traces in der angegebenen Menge um einen Schritt...

| `"retract"` `"("` TraceSet `")"`

... oder durch Verkürzung der Traces in der angegebenen Menge um einen Schritt...

5 Markov-Ketten-Semantik

Bei der Definition der Semantik orientieren wir uns weiter an [KLN07] und [Mil99]. Ein wichtiger Punkt ist allerdings, dass wir eine Semantik aufbauend auf zeitdiskreten Markov-Ketten verwenden und Ausgaben an die Umgebung zulassen. Dazu werden an einigen Stellen Anpassungen notwendig sein.

Die Festlegung der Semantik wird in mehreren Schritten geschehen:

1. Pattern Matching. Im Gegensatz zu [Mil99] übertragen wir nicht Vektoren von Namen, sondern Ausdrücke, die sich aus Konstanten und Namen zusammensetzen. Die Struktur und Bedeutung dieser Ausdrücke wird erklärt, und ein Pattern-Matching-Algorithmus angegeben.

2. Erste strukturelle Kongruenzen und Festlegung der Mengen freier und gebundener Namen. In diesem Abschnitt werden einfache Kongruenzrelationen angegeben, die uns erlauben, bestimmte Operatoren zu eliminieren und einige Prozessaudrücke als äquivalent anzusehen. Insbesondere reduzieren wir auf diesem Weg Vektorausdrücke zu Ausdrücken ohne Vektoren. Davon ausgehend definieren wir die Mengen freier und gebundener Namen eines Ausdruckes.

3. Festlegung der restlichen strukturellen Kongruenzen. Hier werden wir die oben begonnene Kongruenzrelation vervollständigen.

   Die Kongruenzen beschreiben dabei äquivalente Möglichkeiten, einen Prozess aufzubauen. Beispielsweise findet man hier, dass bei der Parallelkomposition die Reihenfolge irrelevant ist, oder wie man `"newvec"` mit Hilfe von `"new"` nachbildet.

   Hier wird auch eine Normalform für Prozesse angegeben, die die nachfolgenden Schritte erleichtert.

4. Reduktionsregeln für die Definition des Reaktions-Verhalten. Hier wird für die übrigen Operatoren eine Menge von Reduktionsregeln angegeben, mit deren Hilfe man den `"new"`-Operator behandeln und die möglichen Kommunikationen bestimmen kann. Hier werden auch die ersten Berechnungsschritte für die Übergangswahrscheinlichkeiten durchgeführt.

   Ergebnis dieses Schrittes ist ein Reduktionssystem, das mögliche Ausgangspunkte für das sichtbare Systemverhalten beschreibt.

5. Reduktionsregeln für die Definition des sichtbaren Verhaltens und Konstruktion einer Markov-Kette. In diesem Schritt werden schließlich Reduktionsregeln für das nach außen sichtbare Verhalten angegeben. Mit Hilfe einiger weiterer Regeln kann

man dann eine zeitdiskrete Markov-Kette konstruieren, deren Zustände Normalformen von Prozessen und deren Übergänge bestimmten Reduktionen entsprechen.

Ein wichtiger Punkt bei der folgenden Semantik ist, dass sie noch nicht vollständig an Markov-Benutzungsmodelle angepasst ist; diese Anpassung wird in Kapitel 7 geschehen. Allerdings ist diese Semantik-Erweiterung sehr einfach zu bewerkstelligen, so dass wir aus Gründen der Klarheit bei dieser ersten Darstellung darauf verzichten.

5.1 Pattern Matching

Aus [KLN07] übernehmen wir den Ansatz, bei einer Kommunikation Terme zu übertragen. Dazu wird bei Sende-Ausdrücken ein Term angegeben, in den zum Sendezeitpunkt die aktuellen Werte der an diesen Term übergebenen Namen eingesetzt werden. Auf Empfängerseite wird ein Muster angegeben. Die Kommunikation kommt zustande, wenn der Term der Senderseite auf das Muster passt. Wenn die Kommunikation zustande kommt, bedeutet das, dass es eine Substitution der Namen im Empfängerterm durchgeführt wird; diese Substitution wird so gewählt, dass das Muster nach Einsetzung dem Term entspricht.

Im Folgenden werden wir Terme und Muster der Form `"x"` als konstante Funktionen behandelt; semantisch besteht zwischen beiden Konstrukten kein Unterschied.

Eine Substitution ist eine Funktion, die gewisse Namen durch Terme ersetzt; wir benutzen die folgenden Schreibweisen: Eine Substitution wird mit griechischen Buchstaben benannt, vorzugsweise σ. Eine Anwendung einer Substitution auf ein Objekt x schreiben wir als $\sigma(x)$. Wollen wir eine Substitution explizit angeben, so verwenden wir die Schreibweise $\sigma = \{a \mapsto x, b \mapsto y, c \mapsto z\}$; in diesem Beispiel wird a durch x ersetzt, b durch y und c durch z.

Im Folgenden wird ein Algorithmus angegeben, der bestimmt, ob ein Term auf ein Muster passt. Wenn ja, wird zusätzlich eine entsprechende Substitution angegeben. Da wir mit Hilfe der Substitution die gebundenen Werte von Namen ändern, werden auch die aktuellen Bindungsrahmen beider Prozesse mit berücksichtigt.

Sei also t ein Term und p ein Muster (d.h., t entspricht der Term-Produktion und p der Pattern-Produktion). Weiterhin seien σ und ρ Funktionen, die den Namenmengen des Senders bzw. Empfängers Werte zuweisen; es handelt sich hierbei um die Bindungsrahmen.

Wir brauchen zur Durchführung des Matchings zunächst einige Hilfsfunktionen. Zunächst brauchen wir eine Funktion, die die Bindungen von in einem Term erscheinenden Namen in diesen Term einsetzt. Das kann einfach durch rekursive Ersetzung geschehen; wir bezeichnen diese Funktion kurz mit $t\sigma$.

Als nächstes müssen wir überprüfen, ob eine Menge von Substitutionen widerspruchsfrei simultan angewendet werden kann, und wenn ja, welche Substitution der simultanen Anwendung entspräche. Wir definieren dazu eine Funktion $\text{merge}(s_1, \ldots, s_n)$, die eine Folge von Substitutionen zusammenführt oder einen Fehler meldet, wenn das nicht möglich ist:

- Bestimme zunächst die Menge V aller Namen, die durch eine der Substitutionen $s_1, \ldots, s_n$ ersetzt werden.
- Für jede Variable $v \in V$, prüfe, ob es zwei Substitutionen s_i, s_j gibt, so dass v von s_i und s_j ersetzt wird, aber $s_i(v) \neq s_j(v)$. Wenn es solche Substitutionen gibt, melde einen Fehler.
- Definiere eine Substitution s. Sie ersetzt jede Variable $v \in V$ durch $s_i(v)$ für ein beliebiges s_i, das v ersetzt.

Der Matching-Algorithmus funktioniert nun, indem die Syntaxbäume von Term und Muster simultan durchlaufen werden:

- Wenn $p = \text{x}$ für einen Bezeichner x ist, so ist das Matching möglich. Die entstehende Substitution ersetzt x durch $t\sigma$.
- Wenn $p = *\text{x}$ für einen Bezeichner x ist, so ist das Matching möglich. Eine Ausnahme liegt vor, wenn kein genaueres Muster vorliegt; dieser Fall wird allerdings weiter unten abgehandelt. Die entstehende Substitution ersetzt x durch $t\sigma$.
- Wenn $p = \$x$, so ist die Bedingung, dass das Matching möglich ist, $t\sigma = x\tau$. Die entstehende Substitution ist die leere Substitution.
- Wenn $p = n\$x$, so ist die Bedingung, dass das Matching möglich ist, $t\sigma = x\tau$. Außerdem muss der Trace von t dann n sein. Die entstehende Substitution ist die leere Substitution.
- Wenn $p = \_$, so ist das Matching immer möglich. Die leere Substitution ist das Ergebnis.
- Wenn $p = f(p_1, \ldots, p_n)$, dann gibt es mehrere Bedingungen für das Matching:
  1. Es muss $f = f(t_1, \ldots, t_n)$ sein.
  2. Für alle $i = 1, \ldots, n$ muss das Matching von p_i und t_i möglich sein. Die entstehenden Substitutionen seien mit s_i bezeichnet.
  3. $\text{merge}(s_1, \ldots, s_n)$ muss erfolgreich sein. Das Ergebnis von merge ist dann die vom Matching berechnete Substitution.

Wie oben erwähnt, brauchen wir noch eine Relation, die beschreibt, wann ein Muster genauer als ein anderes ist. Dazu benutzen wir eine rekursive Definition. Sei im Folgenden p und q Muster. Wir sagen, dass p mindestens so genau wie q ist (kurz $p \succeq q$), wenn eine der folgenden Bedingungen erfüllt ist:

- p geht durch Variablen-Umbenennung (also eine injektive Substitution der Variablennamen) aus q hervor.
- $p \neq *\text{x}$ für alle x, $p \neq \_$ und $q = *\text{y}$.

- $p = f(p_1, \ldots, p_n)$, $q = f(p_1, \ldots, q_n)$ und $p_i \succeq q_i$ für alle $i = 1, \ldots, n$.

Damit wird eine partielle Ordnung auf der Menge der Muster beschrieben.

Als Beispiel für diese Ordnung betrachten wir vier Muster, nämlich $p_1 = f(x, \texttt{"abc"}, g(y))$, $p_2 = f(x, *z, g(y))$, $p_3 = f(*x, \texttt{"abc"}, g(y))$ und $p_4 = f(x, *y, *z)$. Dann gilt $p_1 \succeq p_2, p_3$ und $p_2 \succeq p_4$. Bei einem Ausdruck der Form $x(p_1)?Q_1 + x(p_2)?Q_2 + x(p_3)?Q_3 + x(p_4)?Q_4$ fände dann folgende Auswertung statt: Zuerst wird getestet, ob p_1 auf den gesendeten Term passt. Ist das nicht der Fall, werden p_2 und p_3 getestet. Sollte p_2 nicht passen, wird zusätzlich p_4 getestet. Somit können folgende Empfangskombinationen auftreten:

- p_1,
- p_2,
- p_3,
- p_2, p_3,
- p_4,
- p_3, p_4.

Eine mögliche Anwendung eines solchen Matchings wäre ein Operator, der für bestimmte Stimuli eine Sonderbehandlung darstellt. Wir beschreiben dies mit einem speziellen Beispiel: Der Stimulus `"foo"` soll unterdrückt werden, während alle anderen weitergereicht werden. Der entsprechende Operator sähe wie folgt aus:

```
DropFoo(in: input, out: output) :=
  replicate (in(\String{foo})? + in(*x)? out(x)!);
```

5.2 Elementare strukturelle Kongruenzen

Bei der Definition von π-Kalkülen verkleinert man üblicherweise die Menge der möglichen Prozessausdrücke mit Hilfe von Kongruenzen, erklärt also bestimmte Ausdrücke für äquivalent und damit durcheinander ersetzbar. Im Folgenden werden wir zuerst eine Menge einfacher Kongruenzen konstruieren; die restlichen Kongruenzen benötigen die Begriffe freie Variable, gebundene Variable und α-Äquivalenz, die im nächsten Abschnitt definiert werden.

Als erstes legen wir fest, dass sich die Operatoren `"+"` und `"|"` kommutativ und assoziativ verhalten:

$$(P_1|P_2)|P_3 \equiv P_1|(P_2|P_3)$$
$$(P_1 + P_2) + P_3 \equiv P_1 + (P_2 + P_3)$$
$$P_1|P_2 \equiv P_2|P_1$$
$$P_1 + P_2 \equiv P_2 + P_1$$

Danach definieren wir die Semantik des "replicate"-Operators: replicate P entspricht einer unendlichen Parallel-Komposition von Kopien von P, also P | P | P | \dots. Außerdem können zwei identische "replicate"-Ausdrücke zu einem zusammengefasst werden.

$$\texttt{replicate}\, P \equiv P | \texttt{replicate}\, P$$
$$\texttt{replicate}\, P | \texttt{replicate}\, P \equiv \texttt{replicate}\, P$$

Als nächstes betrachten wir einige Eigenschaften von Sende- und Empfangsoperatoren. Bei Sendeoperatoren sind sowohl der zu sendende Term, der Trace als auch der nachfolgende Prozess optional – um eine einfachere Definition zuzulassen, beschreiben wir hier, wie man eine Normalform herstellen kann. Dabei sei τ ein Trace, P ein Prozess und NONE ein reservierter Bezeichner.

$$\begin{aligned} x! &\equiv x(NONE)!0 \\ x!P &\equiv X(NONE)!P \\ x(t)! &\equiv x(t)!P \\ x!\tau &\equiv x(NONE)!\tau 0 \\ x!\tau P &\equiv X(NONE)!\tau P \\ x(t)!\tau &\equiv x(t)!\tau P \end{aligned}$$

Analog können wir eine Normalform von Empfangsoperatoren definieren:

$$\begin{aligned} x? &\equiv x(NONE)?0 \\ x?P &\equiv x(NONE)?P \\ x(p)? &\equiv x(p)?0 \end{aligned}$$

Der Barrier-Operator ist wie folgt definiert:

$$\texttt{"barrier"}v\texttt{"."}P \equiv v_1(NONE)? \cdots v_n(NONE)?P, \text{ wenn } v = [v_1, \ldots, v_n]$$

Die folgenden Kongruenzen beschreiben Eigenschaften des "new"-Operators: Eine Folge von Bindungen ist äquivalent zu einer Folge von "new"-Schritten, und die Reihenfolge der Aufrufe ist irrelevant, solange die auftretenden Namen paarweise verschieden sind.

$$\begin{aligned} \texttt{new}\, x_1 : \rho_1\, \texttt{new}\, x_2 : \rho_2 . P &\equiv \texttt{new}\, x_2 : \rho_2\, \texttt{new}\, x_1 : \rho_1 . P \text{ wenn } x_1 \neq x_2 \\ \texttt{new}\, x_1 : \rho_1, \ldots, x_n : \rho_n . P &\equiv \texttt{new}\, x_1 : \rho_1 . \ldots \texttt{new}\, x_n : \rho_n . P \end{aligned}$$

Weiterhin benutzt der Nullprozeß keine Namen, also kann ein "new" vor einem Nullprozess entfallen, und ein Nullprozess hat kein sichtbares Verhalten, kann also aus einer Parallelkomposition gestrichen werden.

$$\begin{aligned} \texttt{new}\, x : \rho . 0 &\equiv 0 \\ P | 0 &\equiv P \end{aligned}$$

Term, Termliste t	$v_f(t)$	$v_g(t)$
x (Bezeichner)	$\{x\}$	$\varnothing$
$f()$	$\varnothing$	$\varnothing$
$f(t)$ (t Termliste)	$v_f(t)$	$v_g(t)$
$[t]$ (t Termliste)	$v_f(t)$	$v_g(t)$
$t_1, \ldots, t_n$	$\bigcup_{i=1}^{n} v_g(t_i)$	$\bigcup_{i=1}^{n} v_g(t_i)$

Tabelle 5.1: Bestimmung freier und gebundener Namen: Terme und Termlisten

Muster, Musterliste p	$v_f(p)$	$v_g(p)$
x (Bezeichner)	$\varnothing$	$\{x\}$
$*x$	$\varnothing$	$\{x\}$
$\$x$	$\{x\}$	$\varnothing$
$x\$y$	$\{x, y\}$	$\varnothing$
$f()$	$\varnothing$	$\varnothing$
$f(p)$ (p Musterliste)	$v_f(p)$	$v_g(p)$
$[p]$ (p Musterliste)	$v_f(p)$	$v_g(p)$
$p_1, \ldots, p_n$	$\bigcup_{i=1}^{n} v_f(p_i)$	$\bigcup_{i=1}^{n} v_g(p_i)$

Tabelle 5.2: Bestimmung freier und gebundener Namen: Muster und Musterlisten

Übrig bleibt noch eine Regel, die Instanziierungen ohne Argument beschreibt; wir halten uns an die Vorgehensweise der Aktionen:

$$A \equiv A(NONE)$$

Als nächstes definieren wir die Mengen der *freien* und *gebundenen Namen*. Für einen Prozess P beschreibt dabei $v_f(P)$ die freien und $v_g(P)$ die gebundenen Namen von P; analoges gilt für Muster und Terme.

5.3 Freie und gebundene Namen

Für die weitere Definition der Semantik benötigen wir die Begriffe *freie* und *gebundene* Variable. Anschaulich ist eine Variable dann frei, wenn sie außerhalb des Wirkungsbereichs eines Musters, eines `"newvec"`-Operators oder `"new"`-Operators und nicht auf der linken Seite eines Vektorausdrucks auftaucht. Sie ist gebunden, wenn sie innerhalb eines solchen Wirkungsbereichs auftritt. Für einen Prozess P bezeichnen wir die Menge der freien Namen mit $v_f(p)$ und die Menge der gebundenen Namen mit $v_g(P)$; diese Mengen sind nicht notwendigerweise disjunkt. Die detaillierten Berechnungsvorschriften finden sich in Tabellen 5.1, 5.2 und 5.3.

Ausgehend von dieser Definition kann man den Begriff der *Umbenennung gebundener Namen* definieren. Wir führen sie an einem Beispiel ein, indem wir den folgenden

Prozess P_0	$v_f(P_0)$	$v_g(P_0)$
0	$\varnothing$	$\varnothing$
(P)	$v_f(P)$	$v_g(P)$
$x(p)?P$	$v_f(P)\backslash v_g(p) \cup x \cup v_f(p)$	$v_g(P) \cup v_g(p)$
$x(t)!P$	$v_f(P) \cup x \cup v_f(t)$	$v_g(P)$
$x(t)!\tau P$	$v_f(P) \cup x \cup v_f(t)$	$v_g(P)$
$P_1 \mid P_2$	$v_f(P_1) \cup v_f(P_2)$	$v_g(P_1) \cup v_g(P_2)$
$P_1 + P_2$	$v_f(P_1) \cup v_f(P_2)$	$v_g(P_1) \cup v_g(P_2)$
$\texttt{parallel}\, a_1 : v_1, \ldots, a_n : v_n.P$	$v_f(P) \cup \vec{v}\backslash\vec{a}$	$v_g(P) \cup \vec{a}$
$\texttt{sum}\, a_1 : v_1, \ldots, a_n : v_n.P$	$v_f(P) \cup \vec{v}\backslash\vec{a}$	$v_g(P) \cup \vec{a}$
$\texttt{new}\, a_1 : \rho_1, \ldots, a_n : \rho_n.P$	$v_f(P)\backslash\vec{a}$	$v_g(P) \cup \vec{a}$
$A(t)$	$v_f(t)$	$\varnothing$
$\texttt{replicate}\, P$	$v_f(P)$	$v_g(P)$

Tabelle 5.3: Bestimmung freier und gebundener Namen: Prozesse. Wir vereinbaren zwei Kurzschreibweisen: $\vec{x}$ steht für $\{x_1, \ldots, x_n\}$, und $X \cup a$ für $X \cup \{a\}$, wenn a keine Menge ist.

Prozess betrachten: `x? | new x. (x! | x?)`. Die Variable `x` ist hier offenbar gebunden; wir können sie jetzt beispielsweise in `y` umbenennen, indem wir in dem Teilprozess `new x. (x! | x?)` konsequenz `x` durch `y` ersetzen und so `new y. (y! | y?)` erhalten. Der Gesamtprozess hat dann die Form `x? | new y. (y! | y?)`. Man erkennt, dass man mit Hilfe solcher Umbenennungen die Menge der freien und der gebundenen Namen disjunkt machen kann, und mehr noch: man kann auch die Namen, die in "`new`"-Ausdrücken auftreten, paarweise verschieden machen.

Formal definieren wir den Umbenennungsprozess durch die sogenannte *α-Kongruenz*, die beschreibt, welche Ausdrücke durch Umbenennung gebundener Variablen ineinander hervorgehen können. Dazu definieren wir eine Äquivalenzrelation $\equiv_\alpha$ wie folgt:

1. $\texttt{new}\, x_1 : \rho_1, \ldots, x_n : \rho_n.P \equiv_\alpha \texttt{new}\, y_1 : \rho_1, \ldots, y_n : \rho_n.P\,\{x_1 \mapsto y_1, \ldots, x_n \mapsto y_n\}$, wobei $y_i = y_j$ genau dann, wenn $x_i = x_j$ gilt.

2. $\texttt{newvec}\, x : \rho.P \equiv_\alpha \texttt{newvec}\, y : \rho.P\,\{x \mapsto y\}$.

3. $x(p)?P \equiv_\alpha x(p\,\{y \mapsto y'\})?P\,\{y \mapsto y'\}$, falls $y \in v_g(p)$.

4. $A(p) := P \equiv_\alpha A(p\,\{y \mapsto y'\}) := P\,\{y \mapsto y'\}$, falls $y \in v_g(p)$.

5. $\texttt{sum}(a_1 : v_1, \ldots, a_n : v_n)P \equiv_\alpha \texttt{sum}(b_1 : v_1, \ldots, b_n : v_n)P\,\{a_1 \mapsto b_1, \ldots, a_n \mapsto b_n\}$.

6. $\texttt{parallel}(a_1 : v_1, \ldots, a_n : v_n)P \equiv_\alpha \texttt{parallel}(b_1 : v_1, \ldots, b_n : v_n)P\,\{a_i \mapsto b_i, 1 \leq i \leq n\}$.

7. $\equiv_\alpha$ ist eine Kongruenz bezüglich Prozesskonstruktion. Das bedeutet: Betrachte den Syntaxbaum T eines Prozesses P. Für einen Teilbaum U von T, der zu einem Prozess Q gehört, gilt: Wenn $Q \equiv_\alpha Q'$, und man ersetzt U durch den Syntaxbaum,

der zu Q' gehört, so erhält man einen Syntaxbaum, der zu einem Prozess P' gehört. Dann gilt $P \equiv_\alpha P'$.

Eine genauere Erklärung findet sich in [Mil99] unter dem Begriff „Process congruence“; sie muss allerdings an unsere Syntax entsprechend angepasst werden.

Offenbar kann man die ersten sechs Punkte auch als Ersetzungsregeln formulieren. Wenn man für die Ersetzungen auf der rechten Seite immer neue Namen benutzt, erhält man auf diese Weise einen Algorithmus, der die freien von den gebundenen Namen trennt und sicherstellt, dass jede Variable höchstens einmal gebunden wird.

5.4 Strukturelle Kongruenz

Nachdem wir nun einfache Kongruenzregeln und die Mengen der freien und gebundenen Namen definiert haben und wissen, worum es sich bei der α-Kongruenz handelt, können wir die restlichen Kongruenzregeln angeben. Diese Regeln benutzen die Mengen der freien und gebundenen Namen entweder direkt oder indirekt dadurch, dass Substitutionen angegeben sind. Wir fordern für alle Substitutionen im Folgenden, dass sie *zulässig* sind, das heißt: Sei σ eine Substitution und P ein Prozess. Dann ist $\sigma(P)$ zulässig, wenn für jede Variable x, die von σ ersetzt wird gilt: $x \notin v_g(P)$.

Zunächst legen wir, um immer die Zulässigkeit von Ersetzungen zu garantieren, fest, dass die α-Kongruenz auch eine strukturelle Kongruenz ist:

$$P_1 \equiv_\alpha P_2 \implies P_1 \equiv P_2$$

Als nächstes definieren wir die Semantik der Vektor-Operatoren, indem wir sie auf einfache Operatoren zurückführen. Hierbei und im Folgenden bedeute $\rho(x)$ die Rate des Kanals x.

$$\texttt{newvec}\, x : y. \equiv \texttt{newvec}\, x : [\rho(y_1), \ldots, \rho(y_n)].P, \text{ wenn } y \text{ } n \text{ Elemente hat}$$

$$\texttt{newvec}\, x : [\rho_1, \ldots, \rho_n].P \equiv \texttt{new}\, x_1 : \rho_1, \ldots, x_n : \rho_n.P\,\{x \mapsto [x_1, \ldots, x_n]\}$$

$$\texttt{sum}\, x_1 : [v_{1,1}, \ldots, v_{1,m}], \ldots, x_n : [v_{n,1}, \ldots, v_{n,m}].P \equiv \sum_{i=1}^{m} P\,\{x_1 \mapsto v_{1,i}, \ldots, x_n \mapsto v_{n,i}\}$$

$$\texttt{parallel}\, x_1 : [v_{1,1}, \ldots, v_{1,m}], \ldots, x_n : [v_{n,1}, \ldots, v_{n,m}].P \equiv \prod_{i=1}^{m} P\,\{x_1 \mapsto v_{1,i}, \ldots, x_n \mapsto v_{n,i}\}$$

Die letzte Regel beschreibt die folgende Situation: Wenn zwei Prozesse parallel laufen, und die Variable x kommt in einem von beiden nicht frei vor, so wird dieser Prozess nicht von Bindungen dieser Variable beeinflusst. Die Regel sieht wie folgt aus:

$$\texttt{new}\, x : \rho.(P_1 | P_2) \equiv P_1 | \texttt{new}\, x : \rho.P_2, \text{ wenn } x \notin v_f(P_1)$$

Proc ::=	`"new"` Channels `"."` ProcBound | ProcBound | `"0"`
ProcBound ::=	ProcParallel `"|"` ProcBound | ProcParallel
ProcParallel ::=	ProcReplicate | `"replicate"` ProcReplicate
ProcReplicate ::=	ProcSum `"+"` ProcReplicate |ProcSum
ProcSum ::=	Action Proc
Channels ::=	Channel `","` Channels | Channel
Channel ::=	ID `":"` RateExpr

Tabelle 5.4: Vereinfachte Grammatik für Prozesse in Normalform

5.5 Elimination von Instanziierungen und Prozess-Normalformen

Um die Komplexität der Semantik weiter zu verringern, benutzen wir eine Technik, die in [Mil99] beschrieben wird, um Instanziierungen zu entfernen. Wir führen die Technik zunächst an einem einfachen Beispiel ein.

Sei die Definition `A(x,y) := x(data)? y(data)!` gegeben. Dann definieren wir einen Prozess `replicate def_A(x,y)? x(data)? y(data)!`, den wir kurz mit Defs bezeichnen. Wenn wir nun eine Instanziierung von $A(x, y)$ haben, sagen wir `foo(bar)? A(foo,bar)`, so können wir diese wie folgt umschreiben:

```
foo(bar)? def_A(foo,bar)! | replicate def_A(x,y)? x(data)? y(data)!.
```

Diese Definition ist semantisch mit einer „natürlichen“Einsetzung äquivalent; das ergibt sich aus der unten beschriebenen Reduktions-Semantik.

Sei also $\{A_1(p_1) := P_1, \ldots, A_n(p_n) := P_n\}$ die Menge aller Definitionen. Dann definieren wir einen Prozess Defs = `replicate` $\mathrm{def}_1(p_1)?.P_1 | \ldots |$ `replicate` $\mathrm{def}_n(p_n)?.P_n$. Dann können wir jede Instanziierung $A_i(t)$ durch $\mathrm{def}_i(t)!0$ ersetzen, wenn wir annehmen, dass alle betrachteten Prozesse von der Form $P|$Defs sind.

Somit können wir nun eine Normalform für Prozesse definieren, die uns die Definition des Prozessverhaltens später stark erleichtern wird.

Definition 1 Ein Prozess P heißt *in Normalform*, wenn $v_f(P) \cap v_g(P) = \varnothing$, $P = P'|$Defs, P' der Grammatik in Tabelle 5.4 mit Startsymbol Proc genügt und in P' kein Ausdruck der Form $\mathrm{def}_i(p)?$ vorkommt.

Genügt ein P' sogar der Grammatik, wenn ProcBound das Startsymbol ist, so ist P *in freier Normalform.*

Lemma 1 *Jeder Prozess ist äquivalent zu einem Prozess in Normalform, und Defs ist in Normalform.*

Beweis Analog Proposition 9.13 in [Mil99] durch geeignete Anwendung der Kongruenzen.

5.6 Verhalten der Prozesse

Im Folgenden zeigen wir, welche Semantik die Kommunikation in unserem Kalkül besitzt, wie man mit dem `"new"`-Operator umgeht und schließlich, wie man aus einem π-Kalkül-Prozess eine Markov-Kette ableitet.

Dazu müssen wir zwei wichtige Begriffe einführen, nämlich Aktion und Reaktion.

Aktion Eine Aktion beschreibt die Ausführung eines Sende-Operators in einem π-Kalkül-Prozess, so dass ein Term an die Umgebung ausgegeben wird. Wir benutzen Aktionen, um Stimuli aus unserem Modell auszugeben.

Reaktion Eine Reaktion beschreibt einen Kommunikationsschritt innerhalb eines π-Kalkül-Prozesses. Dabei handelt es sich um die gleichzeitige Ausführung eines Sende- und eines Empfangs-Operators, wobei der Empfangs-Operator den vom Sender-Operator gesendeten Term entgegen nimmt. Auf diese Art wird eine synchrone Kommunikation zwischen verschiedenen Teilprozessen beschrieben.

5.6.1 Reaktionen

Ab hier orientieren wir uns vornehmlich an [KLN07]. Wir werden im Folgenden mehrere Arten von Reduktionsrelationen benutzen:

1. $\xrightarrow{p}$: Reaktion. Diese Reduktionen werden grundsätzlich mindestens mit einer Wahrscheinlichkeit annotiert, z.B. $\xrightarrow{\rho}$. Manchmal werden zusätzliche Indizes annotiert, z.B. $\underset{i,j}{\xrightarrow{\rho(x)}}$.

2. $\xrightarrow{p}\!\!\!\!\rightarrow$: Mehrschritt-Reaktion. Diese Reduktionen werden genau so annotiert wie die Reaktionen.

3. $\underset{x}{\overset{p}{\longmapsto}}$: Sichtbares Verhalten. Diese Reduktionen werden mit Wahrscheinlichkeit und Ausgabe annotiert.

4. $\underset{x}{\overset{p}{\Rightarrow}}$: Kombination aus beidem. Diese Reduktionen entsprechen Zustandsübergängen in der Markov-Kette und werden mit Wahrscheinlichkeit und Ausgabe annotiert.

Wir legen zuerst die Reduktionsregeln für Reaktionen fest.

Diese Reduktionsregeln geben an, wie eine interne Kommunikation abläuft, das bedeutet, wie Stimuli, die in einen Kompositionsoperator eingegeben werden, von diesem aufgebraucht werden können. Sie beschreiben, dass immediate-Transitionen vor allen anderen abgewickelt werden. Die Annotation oben am Pfeil, also das p bei $\xrightarrow{p}$, gibt dabei die Wahrscheinlichkeit einer Transition an. Die folgenden Regeln beschreiben das Kommunikationsverhalten an sich:

COMTRACE: Diese Regel und die nächste die komplexesten. Sie beschreiben, wann, wie und mit welcher Wahrscheinlichkeit eine Kommunikation zwischen zwei Prozessen stattfinden kann, und welche Vorgeschichte einzelnen Termen zugeschrieben wird. Zunächst betrachten wir den Fall, dass ein Trace vorgeschrieben wurde:

$$\frac{\begin{array}{c} C_{i_1}^{j_1} = x(p)?.\,\texttt{new}\,\vec{v_1} : \vec{\rho_1}.Q_1 \\ C_{i_2}^{j_2} = x(t)!\tau.\,\texttt{new}\,\vec{v_2} : \vec{\rho_2}.Q_2 \\ Q_1, Q_2 \text{ in freier Normalform}, i_1 \neq i_2 \\ \text{p hat ein Matching mit t, und die zugehörige Substitution ist } \sigma \end{array}}{\prod_{i=1}^{n} \sum_{i=1}^{m_i} C_i^j | \text{Defs} \xrightarrow[i_1,j_1,i_2,j_2]{\rho(x)} \texttt{new}\,\vec{v_1} : \vec{\rho_1}, \vec{v_2} : \vec{\rho_2}.\left(Q_1\sigma | Q_2 | \prod_{i=1, i\neq i_1, i_2}^{n} \sum_{j=1}^{m_i} C_i^j\right) | \text{Defs}}$$

Die Voraussetzung, damit Kommunikation stattfindet, ist die Existenz zweier unterschiedlicher Prozesse, von denen einer auf einem Kanal x einen Term t sendet, während der andere auf dem gleichen Kanal ein dazu passendes Muster p erwartet.

Wenn die Kommunikation tatsächlich stattfindet, werden die Teilprozesse der Parallelkomposition durch die auf die Kommunikationsoperatoren folgenden Prozesse ersetzt. Die Wahrscheinlichkeit für die Kommunikation errechnet sich aus der dem Kanal zugeordneten Wahrscheinlichkeit.

Als Trace wird für jede von σ substituierte Variable τ festgelegt.

COMNOTRACE: Wird kein Trace vorgeschrieben, so muss einer berechnet werden; ansonsten ist diese Reduktionsregel identisch mit der in COMTRACE. Für die Trace-Berechnung gilt es, zwei Fälle zu unterscheiden:

1. Der Kanal x aus der Reduktionsregel ist ein verbundener Kanal. In diesem Fall wird die Traceberechnungsfunktion des Kanals benutzt.
2. Der Kanal ist kein verbundener Kanal. In diesem Fall werden die Traces aus dem sendenden Prozess übernommen.

NEW: Hier wird beschrieben, dass die Einführung von `"new"`-Bindungen mit der Reduktion verträglich ist:

$$\frac{P \xrightarrow[w]{s} Q, \rho(x) = p}{\texttt{new}\,x : p.P \xrightarrow[w]{s} \texttt{new}\,x : p.Q}$$

EQU: Die Aussage dieser Regel ist, dass äquivalente Ausgangssituationen die gleichen Reduktionen besitzen. Wichtig ist, dass in dieser Regel *nicht* steht, dass äquivalente Zielsituationen eingesetzt werden können. Diese Einschränkung wird für die korrekte Berechnung der Übergangswahrscheinlichkeiten benötigt (ein Beispiel wird weiter unten angegeben):

$$\frac{P \equiv P', P \xrightarrow[w]{p} Q}{P \xrightarrow[w]{p} Q}$$

FIXIMM: Diese Regel legt das Übergangsgewicht von `"imm"`-Transitionen fest:

$$\frac{P \overset{imm}{\underset{w}{\to}} Q}{P \overset{imm(1)}{\underset{w}{\to}} Q}$$

SUMDEL: Da mehrere Reaktionen zu äquivalenten Zielzuständen führen können, wird für die eigentliche Ermittlung der Wahrscheinlichkeit eines Übergangs von P nach Q hier die Summe der Wahrscheinlichkeiten aller Transitionen berechnet, die den gleichen Übergang implementieren. Wichtig ist, dass verzögerte Transitionen überhaupt nur ausgeführt werden, wenn keine immediate-Transitionen mehr möglich sind. Wir definieren dafür zuerst zwei Hilfsfunktionen: Die Funktion $r_{del}(P, Q) = \sum_{p \in \mathbb{R}^+} p \cdot \# \left\{ w \in \mathbb{N}^4 \mid P \overset{p}{\underset{w}{\to}} Q \right\}$ beschreibt dabei die Summe aller Raten, die für Reduktionen der Form $P \overset{p}{\underset{w}{\to}} Q$ mit beliebigem w und festen P, Q, p auftreten, wobei p eine zeitverbrauchende Rate ist. Die Funktion $r_{imm}(P, Q) = \sum_{p \in (0,1]} p \cdot \# \left\{ w \in \mathbb{N}^4 \mid P \overset{imm(p)}{\underset{w}{\to}} Q \right\}$ beschreibt in analoger Weise die gleiche Größe für Raten ohne Zeitverbrauch.

Daraus ergibt sich folgende Reduktionsregel:

$$\frac{\begin{array}{c} \sum_{Q''} r_{imm}(P, Q'') = 0 \\ n = \sum_{Q' \equiv Q} r_{del}(P, Q') > 0 \\ m = \sum_{Q'} r_{del}(P, Q') \quad (\text{es gilt } m \geq n) \end{array}}{P \overset{n/m}{\to} Q}$$

SUMIMM: Diese Regel ist analog zu SUMDEL für immediate-Transitionen und sieht wie folgt aus:

$$\frac{\begin{array}{c} n = \sum_{Q' \equiv Q} r_{imm}(P, Q') > 0 \\ m = \sum_{Q''} r_{del}(P, Q'') \quad (\text{es gilt } m \geq n) \end{array}}{P \overset{imm(n/m)}{\to} Q}$$

Zur Erklärung von EQU und SUMDEL betrachten wir den folgenden Beispiel-Prozess, den wir kurz P nennen: `new x:1, y:1. x? | x! | x! | y? | y!`. Aufgrund von COMNOTRACE erhalten wir drei mögliche Reduktionen: $P \underset{2,1,1,1}{\overset{1}{\to}} 0|0|x!|y?|y! = Q_1$, $P \underset{3,1,1,1}{\overset{1}{\to}} 0|x!|0|y?|y! = Q_2$ sowie $P \underset{5,1,4,1}{\overset{1}{\to}} x?|x!|x!|0|0 = Q_3$. Offensichtlich gilt $Q_1 \equiv Q_2 \equiv x!|y?|y! =: R_1$ sowie $Q_3 \equiv x?|x!|x! =: R_2$.

Vermöge EQU erhalten wir damit $P \underset{2,1,1,1}{\overset{1}{\to}} R_1$, $P \underset{3,1,1,1}{\overset{1}{\to}} R_2$ und $P \underset{5,1,4,1}{\overset{1}{\to}} R_2$. Diese Reduktionen werden mittels SUMDEL zu den zwei Transitionen umgewandelt, nämlich $P \overset{2/3}{\to} R_1$ und $P \overset{1/3}{\to} R_2$.

Wäre EQU formuliert worden als $P \equiv P', Q \equiv Q', P' \xrightarrow[w]{p} Q' \implies P \xrightarrow[w]{p} Q$, so wären statt der drei oben angegebenen Transitionen unendlich viele verschiedene mit Ergebnis Q_1, Q_2 bzw. Q_3 entstanden, so dass SUMDEL nicht mehr hätte angewendet werden können.

5.6.2 Aktionen

Um die Brücke zurück zur Testfallgenerierung mittels Markov-Ketten zu schlagen, legen wir nun fest, dass bestimmte Synchronisationen auch „nach außen“ gehen können. Sendende Synchronisationen mit Namen, die Ausgabe-Kanäle sind, sind immer gestattet, was durch folgenden Reduktionsregeln beschrieben wird:

EXTTRACE: Diese Transition beschreibt in Analogie zu COMTRACE, wie eine Aktion mit Trace-Beschreibung durchgeführt wird:

$$\frac{\begin{array}{c} C_k^\ell = x(t)!\tau.Q, Q \text{ in Normalform} \\ x \text{ Ausgabe-Kanal} \end{array}}{\prod_{i=1}^{n} \sum_{i=1}^{m_i} C_i^j \overset{\rho(x)}{\underset{t,\tau}{\mapsto}} Q | \prod_{i=1, i\neq k}^{n} \sum_{j=1}^{m_i} C_i^j}$$

EXTNOTRACE: Ganz analog zu COMNOTRACE wird hier ein Trace berechnet; wir folgen den gleichen Regeln wie dort, und nennen den Ergebnis-Trace τ'. Damit sieht die zugehörige Reduktionsregel wie folgt aus:

$$\frac{\begin{array}{c} C_k^\ell = x(t)!\tau.Q, Q \text{ in Normalform} \\ x \text{ Ausgabe-Kanal} \end{array}}{\prod_{i=1}^{n} \sum_{i=1}^{m_i} C_i^j \overset{\rho(x)}{\underset{t,\tau'}{\mapsto}} Q | \prod_{i=1, i\neq k}^{n} \sum_{j=1}^{m_i} C_i^j}$$

EEQU: Diese Regel ist das Analogon zu EQU:

$$\frac{P \equiv P', P' \overset{p}{\underset{w}{\mapsto}} Q}{P \overset{p}{\underset{w}{\mapsto}} Q}$$

ENEW: Diese Regel ist das Analogon zu NEW:

$$\frac{P \overset{s}{\underset{w}{\mapsto}} Q, \rho(x) = p}{\texttt{new}\, x : p.P \overset{s}{\underset{w}{\mapsto}} \texttt{new}\, x : p.Q}$$

Schließlich müssen wir noch Aktionen und Reaktionen zusammenbringen. Hierzu fassen wir eine Folge von Reaktionen zusammen zu einer Mehrschritt-Reaktion, und verbinden danach Folgen von Mehrschritt-Reaktionen mit einer Aktion zu einem Berechnungsschritt. Dazu geben wir, analog zu den Eliminationsregeln in [KLN07], Zusammenfassungsregeln an. Wir benutzen dabei die Notation $P \xrightarrow{imm}$, um auszusagen, dass es eine Reduktion der Form $P \xrightarrow{imm(p)} Q$ gibt.

MERGEIMM: Füge eine Folge von keine Zeit verbrauchenden Reaktionen zu einer Mehrschritt-Reaktion zusammen:

$$\frac{P \overset{imm(p_1)}{\rightarrow} \cdots \overset{imm(p_n)}{\rightarrow} Q \overset{imm}{\not\rightarrow}}{P \overset{p_1 \cdots p_n}{\twoheadrightarrow} Q}$$

MERGEMULT: Eine Reaktion, gefolgt von einer Mehrschritt-Reaktion, ist wieder eine Mehrschritt-Reaktion:

$$\frac{P \overset{p}{\rightarrow} Q \overset{q}{\twoheadrightarrow} R, p \in [0,1]}{P \overset{p \cdot q}{\twoheadrightarrow} R}$$

Wir können hier die zusätzliche Bedingung $P \overset{imm}{\not\rightarrow}$ fordern, was allerdings keinen Unterschied machen würde: Wenn $P \overset{imm}{\rightarrow}$, dann gibt keine Transition $P \overset{p}{\rightarrow} Q$ für irgend ein $p \in [0,1]$ und Q. Das liegt daran, dass solche Transitionen nur von SUMDEL erzeugt werden können, und diese Regel ist wegen $P \overset{imm}{\rightarrow}$ nicht anwendbar.

MERGEEXT: Die letzte Regel vereinigt Aktionen und Mehrschritt-Reaktionen zu Transitionen, die das tatsächliche Systemverhalten beschreiben:

$$\frac{P \overset{p_1}{\twoheadrightarrow} \cdots \overset{p_n}{\twoheadrightarrow} Q \underset{x}{\overset{q}{\mapsto}} R \quad Q \overset{imm}{\not\rightarrow}, n \geq 0}{P \underset{x}{\overset{p_1 \cdots p_n \cdot q}{\Rightarrow}} R}$$

Diese Regel erklärt sich dadurch, dass zwischen zwei Aktionen eine beliebig lange (aber endliche) Folge von Reaktionen auftreten kann. Durch die Verwendung der Mehrschritt-Reaktionen haben wir deren Semantik bereits komplett erfasst. Um nun eine Folge von Aktionen mit dazwischenliegenden Reaktionen darzustellen, kombinieren wir eine Aktion mit den vorhergehenden Reaktionen zu einer Transition.

Die Wahl der vorhergehenden Transitionen ergibt sich daraus, dass wir zum Erreichen des Startzustandes eventuell einige zusätzliche Reaktionen benötigen.

5.6.3 Transformation in eine Markov-Kette

Mit Hilfe der oben angegebenen Reduktionsregeln können wir eine zeitdiskrete Markov-Kette definieren: Sei V die Menge der Äquivalenzklassen von π-Kalkül-Ausdrücken bezüglich $\equiv$, $E = \{(P, x, Q) \mid P \underset{x}{\overset{p}{\Rightarrow}} Q\}$ die Menge der Übergänge und $\pi((P, x, Q)) = p$, wenn $P \underset{x}{\overset{p}{\Rightarrow}} Q$.

Diese Markov-Kette ist allerdings noch nicht in der Form, die wir benötigen: Zum einen gibt es keinen ausgezeichneten Start- und Endzustand, und zum anderen hat sie einen unendlichen Zustandsraum.

Diese Probleme kann man glücklicherweise lösen:

- Als Endzustand können wir jeden Zustand betrachten, in dem keine Aktion oder Reaktion mehr stattfinden kann.
- Der Anfangszustand kann explizit festgelegt werden, indem man den Prozess, der initial konstruiert wird, als diesen Zustand betrachtet.
- Auch wenn der Zustandsraum unendlich ist, kann man im Allgemeinen dafür sorgen, dass nur endliche viele Zustände vom Startzustand aus erreichbar sind. Durch Entfernen aller unerreichbaren Zustände hat man dann wiederum eine endliche Markovkette erzeugt.

6 Darstellung der Eingaben im π-Kalkül

Es stellt sich die Frage, wie man Systemeingaben wie Markov-Ketten, Homomorphismen oder Mealy-Maschinen in einer Weise darstellen kann, dass sie für Kompositionsoperatoren, die als π-Kalkül-Ausdrücke implementiert sind, nutzbar werden. Glücklicherweise werden wir in diesem Kapitel zeigen können, dass jedes dieser Objekte ohne großen Aufwand als π-Kalkül-Prozess dargestellt werden kann. Bei dieser Gelegenheit werden wir zusätzlich noch einige Abkürzungen definieren, die uns das Angeben von Operatoren erleichtern werden.

6.1 Funktionen und Homomorphismen

Für Funktionen schlagen wir das folgende Aufrufprotokoll vor:

1. Der Aufrufer sendet die Nachricht (c, x) an den Prozess, der die Funktion implementiert.

2. Der Funktionsprozess sendet den Funktionswert $f(x)$ über den Kanal c zurück.

Soll ein endliches Wort über c übertragen werden, schlagen wir folgendes Protokoll vor: Zuerst wird START gesendet, darauf die Buchstaben und endlich END.

Ein Homomorphismus kann dann folgendem vereinfachten Protokoll genügen:

1. Der Aufrufer sendet die Nachricht $START(i, o)$ an den Prozess, der den Homomorphismus implementiert.

2. Danach werden die Eingaben an den Homomorphismus auf i übertragen, und die Ausgaben auf o.

3. Sobald auf i END übertragen wird, so wird auch auf o END übertragen. Der Aufruf des Homomorphismus ist damit beendet.

Ein Homomorphismus $f : \Sigma^* \to \Delta^*$ mit $f(\sigma_i) = \delta_{i,1} \cdots \delta_{i,n_i}$, $\Sigma = \{\sigma_1, \ldots, \sigma_m\}$, kann dann als folgender Prozess dargestellt werden:

```
Pf := new run: imm. (
     ( replicate f(START(in,out))? run(in,out)!
     | replicate run(in, out)? (
         in(σ_1)? out(δ_{1,1})! ...out(δ_{1,n_1})! run(in, out)!
         + in(σ_2)? out(δ_{2,1})! ...out(δ_{2,n_2})! run(in, out)!
         ⋮
         + in(σ_m)? out(δ_{m,1})! ...out(δ_{m,n_m})! run(in, out)!
         + in(END)? out(END)!)
    )
)
```

6.2 Markovketten und Mealy-Automaten

Die Idee bei der Übersetzung dieser Konstrukte wird es sein, einen Prozess für jeden Zustand des Automaten bzw. der Markov-Kette zu erzeugen und die Zustandsübergänge dann als Menge von Kommunikationen zu implementieren. Wir fangen mit einem Mealy-Automaten M an: Angenommen, M hat n Zustände $s_1, \ldots, s_n$ und m Stimuli $\sigma_1, \ldots, \sigma_m$. Es sei weiterhin $s_M = s_1$.

Wir definieren zunächst ein Prozessfragment $S(s, \sigma)$ wie folgt:

$$S(i, \sigma) = \texttt{s(}i\texttt{)? in(}\sigma\texttt{)?} \sum_{(s_i, \sigma, \gamma, s_j) \in \delta_M} \texttt{out(}\gamma\texttt{)! s(}j\texttt{)!}$$

Hierbei ist die leere Summe der Null-Prozess.

Wir können dann einen Mealy-Automaten mit Hilfe des folgenden Prozesses darstellen:

$$\texttt{PM(in,out) := new s1: imm, ..., sn: imm. (s1! | replicate } \sum_{i=1}^{n} \sum_{\sigma \in \Sigma} \texttt{S(i,}\sigma\texttt{))}$$

Für Markov-Modelle können wir ganz ähnliche Ansätze verwenden. Wir nehmen wieder an, dass die Zustande von P mit $v_1, \ldots, v_n$ bezeichnet sind und die Stimuli mit $\sigma_1, \ldots, \sigma_m$. Sei nun v ein Zustand des Markov-Modells P und $e_1, \ldots, e_r$ die Menge der Kanten, die von v ausgehen. Es sei $e_i = (v, \sigma_i, v_{n_i})$ und $p_i = \pi_P(e_i)$. Wir definieren wieder ein Prozessfragment $S(v)$ wie folgt:

```
S(v_i) = vi? new c1: p_1,..., cr: p_r.
     ( c1! + ...+ cr!
     | c1? out(σ_1)! sn_1! + cr? out(σ_r)! vn_r!
)
```

Damit können wir wiederum einen Prozess angeben, der die Markov-Kette darstellt. Wir nehmen an, dass $s_P = v_1$ und $f_P = v_n$ ist.

PP(out) := new v1: imm, ..., vn: imm. (v1! | replicate ($\sum_{i=1}^{n-1} S(v_i)$ + vn? out(END)!)

7 Behandlung allgemeiner Markov-Benutzungsmodelle

7.1 Behandlung von Benutzungsprofilen und Labels

Die oben angegebene Semantik beschreibt nur, wie man Markovketten verarbeiten kann. Um allgemeine Markov-Benutzungmodelle zu verarbeiten, müssen wir die Semantik entsprechend anpassen.

Hierbei machen wir uns zu Nutze, dass ein Markov-Benutzungsmodell angegeben werden kann als ein Transitionssystem, bei dem jeder Transition eine Familie von Übergangswahrscheinlichkeiten sowie eine Menge von Labels (Paare von Namen und String, die zur Erzeugung der konkreten Testfälle verwendet werden) zugeordnet ist, wobei die Indexmengen dieser Familien für alle Transitionen gleich sind.

Nun haben die Labels für die Semantik der Benutzungsmodelle nur insofern Bedeutung, als dass sie die tatsächliche Umsetzung der Stimuli beschreiben; auf die Komposition haben sie keinen Einfluss. Deshalb genügt es später, die Labels einfach als Zusatzinformation an den Reduktionen mitzuführen und geeignet zu verrechnen; die Reduktionsregeln ändern sich prinzipiell nicht.

Für die Übergangswahrscheinlichkeiten gilt ähnliches: Sie gehen nur in die Berechnung der Reduktionswahrscheinlichkeiten ein, sind aber für die Bestimmung der prinzipiell möglichen Reduktionen irrelevant. Somit speichern wir später statt einer einzelnen Reduktionswahrscheinlichkeit eine Familie von Wahrscheinlichkeiten.

Trotzdem stellen sich noch Fragen, wie diese Daten weiter zu verarbeiten sind; wir können nicht annehmen, dass in allen Eingabemodellen die gleichen Benutzungsprofilnamen und Labels verwendet werden, ebenso wenig wie wir annehmen können, dass immer die gleichen Profile und Labels verschmolzen werden sollen. Im Folgenden werden für diese Fragen Lösungsmöglichkeiten vorgeschlagen.

Für Benutzungsprofile ist das Vorgehen relativ einfach: wir fordern vom Benutzer, dass er beim Aufruf eines Kompositionsoperators angibt, welche Benutzungsprofile im Ergebnismodell erzeugt werden und aus welchen Profilen in den Quellmodellen sie errechnet werden. Die Wahrscheinlichkeitsberechnungen werden für alle solchen Kombinationen durchgeführt. Die Beschreibung der Profil-Kombination findet sich dementsprechend bei der Beschreibung der Kompositionssprache.

Bei den Labels gilt zunächst die gleiche Überlegung wie bei den Benutzungsprofilen: Der Benutzer soll eine Liste von Label-Berechnungen angeben. Im Unterschied zu den Benutzungsprofilen, bei denen man für die Durchführung der Komposition von jedem Teilmodell genau ein Profil für die Berechnung der Endwahrscheinlichkeiten braucht,

muss hier unterschieden werden, ob die Labels eines Eingabemodells können nur bei Aktionen oder auch bei Reaktionen mit ausgegeben werden. Die Reihenfolge, in der die Labels mit ausgegeben werden, ergeben sich dabei durch die Reihenfolge der Reaktionen bei der Zusammenfassung zu Mehrschritt-Reaktionen. Die Beschreibung der Label-Kombination findet sich wiederum bei der Beschreibung der Kompositionssprache.

Wir beschreiben dieses Vorgehen an einem Beispiel: Angenommen, wir haben zwei Benutzungsmodelle, sagen wir `Fahrer` und `Abstandssensor`, und wollen eine Parallelkomposition von beiden berechnen. In `Fahrer` gibt es drei Profile namens `stadt`, `autobahn` und `landstraße`; der Abstandssensor hat zwei Profile, `geringeEigengeschwindigkeit` (unter 30km/h) und `hoheEigengeschwindigkeit`. Wir nehmen an, dass es vier für uns interessante Profile gibt: In der Stadt kann man langsam oder schnell sein, also soll es `stadtLangsam` und `stadtSchnell` geben, und auf anderen Strecken ist man schnell, also soll es `autobahn` und `landstraße` geben.

Somit haben wir folgende Profilbeschreibungen:

Gemeinsames Profil	`Fahrer`	`Abstandssensor`
`stadtLangsam`	`stadt`	`geringeEigengeschwindigkeit`
`stadtSchnell`	`stadt`	`hoheEigengeschwindigkeit`
`autobahn`	`autobahn`	`hoheEigengeschwindigkeit`
`landstraße`	`landstraße`	`hoheEigengeschwindigkeit`

Bei der Berechnung der Markov-Kette, die sich aus der Komposition ergibt, gehen wir dann wie folgt vor: Wir annotieren unsere Eingabe-Modelle für die Berechnung mit neuen Profilen, nämlich den vom Benutzer geforderten (`stadtLangsam, stadtSchnell, autobahn, landstraße`) und übernehmen die Übergangswahrscheinlichkeiten der entsprechenden Ursprungsprofile. Als Semantik können wir dann die Markov-Kette, die durch die Komposition entsteht, für jedes Profil einzeln berechnen, oder, was äquivalent ist, uns zu Nutze machen, dass die resultierenden Markov-Ketten alle die gleiche Struktur haben und die benutzten Aktions- und Reaktionsmengen identisch sind; somit müssen wir nur statt einer Wahrscheinlichkeitsberechnung eine pro Profil durchführen, was im nächsten Abschnitt beschrieben wird.

Für Labels wählen wir ein ähnliches Vorgehen, nur dass wir in der Tabelle noch zusätzlich vermerken, ob die Labels nur bei Aktionen oder auch bei Reaktionen ausgegeben werden sollen. Die Annotation der Labels an die resultierende Markov-Kette wird im folgenden Abschnitt besprochen.

7.2 Integration in die Syntax und Semantik

Als letzten Schritt zur Beschreibung, wie man Markov-Benutzungsmodelle aus entsprechenden π-Kalkül-Ausdrücken generiert, geben wir im Folgenden zwei kleine Syntax-Erweiterungen an (zum einen werden Raten-Ausdrücke zu Profil-Ausdrücken erweitert, und zum anderen erhalten Terme die Möglichkeit, Labels mit ihnen zu assoziieren), und beschreiben die erweiterte Semantik. Glücklicherweise müssen wir dazu nur sehr wenige

Änderungen vornehmen; insbesondere gelten die Definition von freien und gebundenen Variable unverändert weiter, bei den strukturellen Kongruenzen müssen wir nur eine kleine Anpassung vornehmen, und die Anpassungen der Reduktionsregeln sind ohne großen Aufwand durchführbar.

Wir betrachten zuerst die Syntax-Erweiterungen:

RateExpr ::= RATE
Zeitverbrauchend, Rate wie im Argument
| `"imm"`
Kein Zeitverbrauch, Rate 1
| `"imm" "(" RATE ")"`
Kein Zeitverbrauch, Rate wie im Argument
| `"(" STRING "=>" RATE ")"`
lege Wahrscheinlichkeiten für verschiedene Profile fest

BasicRateExpr ::= RATE

ActionParameter ::= Trace
| `"label" STRING ":" STRING`
Lege ein Label für diese Aktion fest

Um nun die neue Syntax von Sende-Ausdrücken anzugeben, bezeichne (τ, λ) im Folgenden die Zusammenfassung ActionParameters-Ausdruck, wobei τ ein Trace und λ eine Menge von Labels ist. Für den Fall, dass τ nicht angegeben ist, schreiben wir $\tau = \bot$; wenn kein λ angegeben wird, können wir $\lambda = \varnothing$ schreiben. Ansonsten enthalten τ den angegebenen Trace, und λ die angegebenen Labels.

Dann erhalten wir folgende Kongruenzen für den Sende-Operator:

$$\begin{aligned} x(t)!(\tau,\lambda) &\equiv x(t)!(\tau,\lambda)0 \\ x!(\tau,\lambda)P &\equiv x(NONE)!(\tau,\lambda)P \\ x!(\tau,\lambda) &\equiv x(NONE)!(\tau,\lambda)0 \end{aligned}$$

Abschließend geben wir die neuen Reduktionsregeln an, und beschreiben, wie sie sich von den bisherigen unterscheiden. Wir behalten dabei die Regelnamen bei, um einen besseren Vergleich zu ermöglichen. Die nicht angegebenen Regeln sind identisch mit den Original-Regeln.

COMTRACE: Hier werden diejenigen Labels, die auch bei Reaktionen sichtbar sind, an der Reaktions-Reduktion annotiert. Außerdem definieren wir ρ wie folgt um: Hat x Benutzungsprofile, so liefert $\rho(x)$ diese. Wenn nicht, so wird eine Menge von Benutzungsprofilen für x angelegt, deren Namen denen des aktuellen Profils

entsprechen und deren Raten alle mit der Rate von x identisch sind (für zeitverbrauchende Transitionen), oder ein Ausdruck $\mathrm{imm}(x)$ (für nicht zeitverbrauchende Transitionen).

Eine Menge von Benutzungsprofilen kann als Funktion dargestellt werden, und zwar von einer Menge K (keys) von Profilnamen nach $\mathbb{R}^+$.

$$\frac{\begin{array}{c} C_{i_1}^{j_1} = x(p)?.\,\texttt{new}\,\vec{v_1} : \vec{\rho_1}.Q_1 \\ C_{i_2}^{j_2} = x(t)!(\tau,\lambda).\,\texttt{new}\,\vec{v_2} : \vec{\rho_2}.Q_2 \\ Q_1, Q_2 \text{ in freier Normalform}, i_1 \neq i_2, \tau \neq \bot \\ \text{p hat ein Matching mit t, und die zugehörige Substitution ist } \sigma \\ \lambda' \subseteq \lambda \text{ so, dass alle Labels in } \lambda' \text{ bei Reaktionen sichtbar sind.} \end{array}}{\prod_{i=1}^{n} \sum_{i=1}^{m_i} C_i^j | \mathrm{Defs} \overset{\rho(x),\lambda'}{\underset{i_1,j_1,i_2,j_2}{\longrightarrow}} \texttt{new}\,\vec{v_1} : \vec{\rho_1}, \vec{v_2} : \vec{\rho_2}.\left(Q_1\sigma | Q_2 | \prod_{i=1, i\neq i_1, i_2}^{n} \sum_{j=1}^{m_i} C_i^j\right) | \mathrm{Defs}}$$

COMNOTRACE: Wie oben beschrieben (neues COMTRACE + Trace-Berechnung).

SUMDEL: Bei SUMDEL und der folgenden Transition, SUMIMM, müssen wir zwei Dinge beachten. Zum einen haben wir jetzt Ratenvektoren statt einzelner Raten, und zum anderen können wir Transitionen mit verschiedenen Labeln nicht miteinander identifizieren.

Die Funktion r_{del} erweitern wir wie folgt zu einem Funktor:

$$r_{del}(P, Q, \lambda)(k) = \sum_{r \in \mathbb{R}^+} r \cdot \#\left\{w \in \mathbb{N}^4 \mid P \overset{p,\lambda}{\underset{w}{\rightarrow}} Q, p(k) = r\right\}$$

Man beachte, dass an r_{imm} keine Änderung nötig ist. Außerdem bezeichnen wir mit 0 die konstante 0-Funktion. Für zwei Funktion $f, g : K \to \mathbb{R}+$ gelte $f \geq g$ genau dann, wenn für alle $k \in K$ gilt: $f(k) \geq g(k)$; $f > g$ ist äquivalent zu $f \geq g$ und $f \neq g$. Schließlich sind die arithmetischen Operationen $(+, -, \cdot, /)$ auf der Menge dieser Funktionen punktweise definiert (d.h., $(f/g)(k) = f(k)/g(k)$), und wir legen fest, dass $0/0 = 0$ gelten soll.

Daraus ergibt sich folgende Reduktionsregel:

$$\frac{\begin{array}{c} \sum_{Q'',\lambda''} r_{imm}(P, Q'', \lambda'') = 0 \\ n = \sum_{Q' \equiv Q} r_{del}(P, Q', \lambda) > 0 \\ m = \sum_{Q',\lambda'} r_{del}(P, Q', \lambda') \quad (\text{es gilt } m \geq n) \end{array}}{P \overset{n/m}{\underset{\lambda}{\rightarrow}} Q}$$

SUMIMM: Diese Regel ist analog zu SUMDEL für immediate-Transitionen und sieht wie folgt aus:

$$\frac{\begin{array}{c} n = \sum_{Q' \equiv Q} r_{imm}(P, Q', \lambda) > 0 \\ m = \sum_{Q'',\lambda''} r_{imm}(P, Q''\lambda'') \quad (\text{es gilt } m \geq n) \end{array}}{P \overset{imm(n/m)}{\underset{\lambda}{\rightarrow}} Q}$$

EXTTRACE: Hier ergänzen wir wieder die Labels:

$$\frac{C_k^\ell = x(t)!(\tau,\lambda).Q, Q \text{ in Normalform} \quad x \text{ Ausgabe-Kanal}}{\prod_{i=1}^n \sum_{i=1}^{m_i} C_i^j \overset{\rho(x)}{\underset{t,\tau,\lambda}{\mapsto}} Q | \prod_{i=1,i\neq k}^n \sum_{j=1}^{m_i} C_i^j}$$

EXTNOTRACE: Hier gelten die gleichen Bemerkungen zur Trace-Berechnung wie bei der Original-Definition; der berechnete Trace heißt τ'. Wir ergänzen wieder die Labels.

$$\frac{C_k^\ell = x(t)!(\bot,\lambda).Q, Q \text{ in Normalform} \quad x \text{ Ausgabe-Kanal}}{\prod_{i=1}^n \sum_{i=1}^{m_i} C_i^j \overset{\rho(x)}{\underset{t,\tau',\lambda}{\mapsto}} Q | \prod_{i=1,i\neq k}^n \sum_{j=1}^{m_i} C_i^j}$$

MERGEIMM: Hier müssen zwei Dinge beachtet werden: Transitionen, bei denen jedes Profil die Wahrscheinlichkeit 0 hat, brauchen nicht beachtet zu werden, und wir müssen uns um die Labels kümmern. Dazu stellen wir Label-Mengen auch als Funktion dar, nämlich $\lambda : L \to \mathcal{S}$, wobei L die Menge der Label-Namen und $\mathcal{S}$ die Menge der Strings über einem nicht näher angegebenen Alphabet ist. Gibt es zu einer Transition ein bestimmtes Label ℓ nicht, so sei $\lambda(\ell) = \varepsilon$, der leere String. Wir definieren dann für zwei solcher Funktionen λ_1, λ_2 die Konkatenations-Operation $\lambda_1 \cdot \lambda_2$ durch $(\lambda_1 \cdot \lambda_2)(\ell) = \lambda_1(\ell) \cdot \lambda_2(\ell)$.

$$\frac{P \overset{imm(p_1)}{\underset{\lambda_1}{\to}} \cdots \overset{imm(p_n)}{\underset{\lambda_n}{\to}} Q \overset{imm}{\not\mapsto} \quad p_1 \cdots p_n \neq 0}{P \overset{\text{imm}(p_1 \cdots p_n)}{\underset{\lambda_1 \cdots \lambda_n}{\twoheadrightarrow}} Q}$$

MERGEMULT: Hier gilt das gleiche wie bei MERGEMULT. Außerdem definieren wir für ein Profil p und eine Immediate-Übergangswahrscheinlichkeit $\text{imm}(q)$ den Ausdruck $p \cdot \text{imm}(q)$ als $(p \cdot \text{imm}(q))(k) = p(k) \cdot q$.

$$\frac{P \overset{p}{\underset{\lambda_1}{\to}} Q \overset{q}{\underset{\lambda_2}{\twoheadrightarrow}} R, p \text{ Profil} \quad p \cdot q \neq 0}{P \overset{p \cdot q}{\underset{\lambda_1 \cdot \lambda_2}{\twoheadrightarrow}} R}$$

MERGEEXT: Auch hier gilt das gleiche wie bei MERGEMULT:

$$\frac{P \overset{p_1}{\underset{\lambda_1}{\twoheadrightarrow}} \cdots \overset{p_n}{\underset{\lambda_n}{\twoheadrightarrow}} Q \overset{q}{\underset{x,\lambda}{\mapsto}} R \quad Q \overset{imm}{\not\mapsto}, n \geq 0, p_1 \cdots p_n \cdot q \neq 0}{P \overset{p_1 \cdots p_n \cdot q}{\underset{x,\lambda_1 \cdots \lambda_n \cdot \lambda}{\Rightarrow}} R}$$

Die Konstruktion des Markov-Benutzungsmodells erfolgt dann in völlig analoger Weise zur Konstruktion der Markov-Kette.

8 Modelleigenschaften

8.1 Einleitung

8.2 Definitionen

8.2.1 Terminierende und nicht-terminierende Markov-Modelle

Im Folgenden definieren wir die Begriffe terminierendes und nicht-terminierendes Markov-Benutzungsmodell. Dazu betrachten wir eine Markov-Kette mit folgenden Eigenschaften:

1. Der Zustandsraum der Markovkette ist endlich und die Markovkette zeitdiskret.
2. Die Transitionen der Markov-Kette sind mit Stimuli beschriftet, und die Menge der Stimuli ist endlich.
3. Es kann mehrere Transitionen von einem Zustand a in einen Zustand b geben.
4. Es gibt zwei ausgezeichnete Stimuli, END und STUTTER. Es gibt höchstens einen Zustand, der eine ausgehende Kante mit der Beschriftung END hat; wir bezeichnen diesen Zustand als Endzustand. Anmerkung: Diese Aussage bedeutet, dass es höchstens einen Endzustand gibt. Sie bedeutet nicht, dass dieser Zustand auch tatsächlich existieren muss.
5. Der Endzustand ist absorbierend und hat genau eine ausgehende Kante, nämlich die mit END beschriftete.
6. Es gibt einen ausgezeichneten Startzustand.
7. Für jeden Zustand z gilt: Es gibt eine Folge von Transitionen, über die z erreichbar ist.(D.h. es existieren Profile, für die gilt: Vom Startzustand aus ist die Wahrscheinlichkeit, z zu erreichen, größer 0.
8. Wenn es einen Endzustand gibt, so ist für jeden Zustand z, der mit einer Wahrscheinlichkeit größer 0 erreicht wird, auch die Wahrscheinlichkeit, von z aus den Endzustand zu erreichen, größer 0.

Markov-Ketten dieses Typs *mit* Endzustand sind genau die von Poore beschriebenen Markov-Benutzungsmodelle. Wir wollen sie als *terminierende Markov-Modelle* bezeichnen. Umgekehrt soll eine Markov-Kette mit den oben angegebenen Eigenschaften, aber

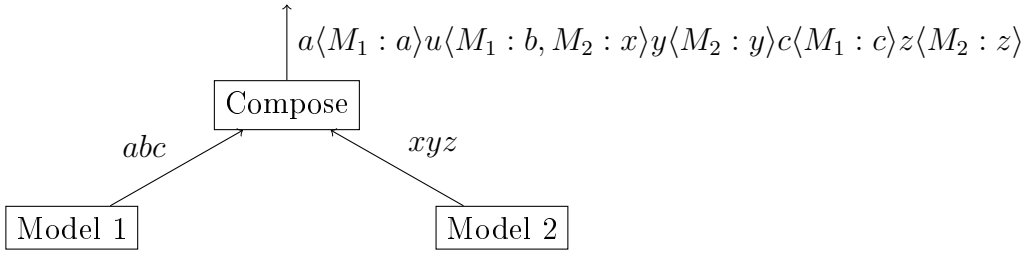

Abbildung 8.1: Ausgabefolgen in einem kompositen Markov-Benutzungsmodell, mit Vorgeschichten

ohne Endzustand, als *nicht-terminierendes Markov-Modell* bezeichnet werden. Die Menge alle solchen Markov-Ketten soll als Menge der *Markov-Modelle* bezeichnet werden.

Sei s die Folge der Stimuli, die ein Random Walk durch ein Markov-Modell M induziert. Wir nennen diese Folge einen *Trace* von M. Man kann leicht beweisen, dass folgendes gilt:

1. Ist M nicht-terminierend, dann kommt END in s nicht vor.
2. Ist M terminierend, dann ist s mit Wahrscheinlichkeit 1 von der Form $s = s_0\mathsf{END}^\omega$ für eine endliche Folge s_0, in der END nicht vorkommt.

Damit haben wir die notwendige Fallunterscheidung zwischen Modellen, die fast sicher nur endliche Traces ausgeben, und Modellen, die garantiert nur unendliche Traces ausgeben, festgelegt.

8.2.2 Projektion und starke Trace-Erhaltung

Seien $M_1, \ldots, M_k$ Instanzen von Markov-Modelle, f ein Kompositionsoperator und $M := f(M_1, \ldots, M_k)$ das komposite Modell, das durch Anwendung von f auf die Submodelle entsteht.

Wir wollen aus mehreren Gründen ermöglichen, nachzuvollziehen, welche Stimulus-Folge eines Submodells zu einer bestimmten Ausgabe kompositen Modells geführt hat:

1. Es soll möglich sein, die statistische Auswertung von Testergebnissen nicht nur auf dem kompositen Modell, sondern auch auf einzelnen der Submodelle durchzuführen.
2. Man will nachweisen können, dass die Anwendung eines Kompositionsoperators nicht die „Markov-Eigenschaft eines Submodells“ zerstört.
3. Man will Coverage-Aussagen über die Submodelle treffen können.
4. Falls ein Fehler im zu testenden System gefunden wird, will man ermitteln, welche Stimuli in den Submodellen ihn aufgedeckt haben.

Dazu führen wir den Begriff der *Stimulus-Vorgeschichte* ein. Jedem Stimulus, der von einem kompositen Modell erzeugt wird, ordnen wir eine Funktion zu, die von Eingabemodellen auf Stimulus-Sequenzen abbildet. Wir schreiben sie als Menge von Paaren der Form $M_i : v_i$. Notiert wird eine solche Zuordnung für einen Stimulus s als $s\langle M_{j_1} : v_1, \ldots, M_{j_k} : v_k\rangle$. Wir legen weiterhin fest, dass jedem nicht explizit angegeben Modell die leere Sequenz zugeordnet wird. Dadurch können wir ohne Beschränkung der Allgemeinheit alle Traces in der Form $\langle M_1 : v_1, \ldots, M_n : v_n\rangle$ schreiben.

Anschaulich gesprochen sind die Stimulus-Folgen der Vorgeschichte diejenigen Stimulus-Folgen, die der Kompositionsoperator von den entsprechenden Submodellen seit der letzten Stimulus-Ausgabe eingelesen hat, und die zur Ausgabe des aktuellen Stimulus führen. Eine Beispiel findet sich in Abbildung 8.1.

Darauf aufbauend ergibt sich der Begriff der *Projektion von Stimulusfolgen*. Wir können für eine Stimulus-Folge mit Vorgeschichte zu jedem Submodell eine Stimulus-Folge dieses Submodells angeben, die an der Entstehung der Ausgabe beteiligt war. Dazu gehen wir folgendermaßen vor:

Sei $\sigma = s_1\langle M_1 : v_{1,1}, \ldots, M_n : v_{1,n}\rangle, s_2\langle M_1 : v_{2,1}, \ldots, M_n : v_{2,n}\rangle \cdots$ eine Folge von Stimuli mit Vorgeschichte. Dann ist die Projektion auf das Modell M_i (bzw. die Projektion auf eine konkrete Instanz eines Modell) definiert durch $\pi_{M_i}(\sigma) = v_{i,1} \cdot v_{i,2} \cdots$.

Mit Hilfe dieser Definition können wir nun Bedingungen angeben, die beschreiben, wann man die Ergebnisse einer Stichprobe von Testfällen aus dem Modell M ohne weiteres auf die Modelle M_i übertragen kann.

Wir sagen, $f(M_1, \ldots, M_n)$ *ist stark trace-erhaltend für ein terminierendes Modell* M_i, wenn für jeden Trace $t \in \mathrm{Tr}(M_i)$

$$P_{M_i}(t) = \sum_{\pi_{M_i}(t')=t} P_M(t')$$

gilt. Ist M_i nicht-terminierend, so fordern wir, dass die Bedingung für jedes endliche Präfix t der Traces von M_i erfüllt wird.

Wenn wir M_i beliebig wählen können, so nennen wir f *stark trace-erhaltend für das* i*-te Modell*. Ist f stark trace-erhaltend für alle Eingabe-Modelle, so nennen wir f stark trace-erhaltend. Viele der vorgestellten Kompositionsoperatoren erfüllen diese Eigenschaft oder sind zumindest für einige ihrer Eingabemodelle stark trace-erhaltend.

π_A bildet immer auf eine konkrete Instanz des Modells ab. Sollte in einem kompositen Modell M das Submodell A mehrfach auftauchen, muss angegeben werden, auf welche Instanz von A abgebildet werden soll. Sonst würde die mehrfache Eingabe des selben Modells die strake Trace-Erhaltung in fast allen Operatoren zerstören machen.

8.2.3 Andere Formen der Trace-Erhaltung

Leider sind nicht alle sinnvollen Operatoren stark trace-erhaltend. Aus diesem Grund betrachten wir im Folgenden einige Varianten der starken Trace-Erhaltung, die in eingeschränktem Maße die gleichen Aussagen erlauben.

Von den im Operatorkatalog beschriebenen Operatoren fallen hierbei besonders zwei ins Auge, nämlich „fail use“ und „make terminating“. Beide Operatoren gehen in jedem

Zustand mit einer gewissen (festen) Wahrscheinlichkeit in einen Zustand und von dort, ohne das Originalmodell zu durchqueren, in den Endzustand. Somit entstehen in beiden Fällen neue Traces, deren Projektion nur ein Präfix eines Traces im Original-Modell ist. Aufgrund der momentan von JUMBL angebotenen Testergebnis-Beschreibungen und Testfall-Auswertungen können wir mit den vorhandenen Mitteln keine Auswertung des Eingabemodells an diese Operatoren durchführen. Allerdings erscheint es einsichtig, dass bei entsprechenden Erweiterung der Zuverlässigkeits-Schätzung auch Präfixe einer Benutzung in die Zuverlässigkeitsberechnung mit einfliessen können. TODO gibt es hier eine einfache Form, so etwas wie Präfix-Traceerhaltung zu definieren? Bisherige Ansätze zu komplex.

8.3 Rück-Abbildbarkeit von Testergebnissen

Die Benutzung von Kompositionsoperatoren, die stark trace-erhaltend sind, erlaubt es uns, die Ergebnisse von Testfällen, die vom kompositen Modell erzeugt wurden, wieder auf die Submodelle abzubilden. Wir betrachten zunächst den einfachsten Fall: Zu jedem Stimulus s des ausgegebenen Modells gibt es eine Vorgeschichte der Form $\langle M_i : s' \rangle$. Wenn bei der Ausführung eines Stimulus-Traces σ ein Fehler im j-ten Schritt auftritt, so kann man die Vorgeschichte $\langle M_i : s' \rangle$ zu s_j ermitteln und weiß auf diese Art, zu welchem Submodell der Fehler offenbarende Stimulus gehört. Durch Anwendung der Trace-Projektion $\pi_{M_i}(s_1 \cdots s_j)$ erhält man dann den Trace von M_i, der den Fehler offenbart, und kann dann so die fehleroffenbarende Transition von M_i ermitteln.

8.3.1 Komplexe Vorgeschichten: aktive Stimuli

Nicht alle Operatoren, die stark trace-erhaltend sind, liefern so einfache Vorgeschichten wie die eben beschriebene. Um das Problem zu erläutern, betrachten wir ein Beispiel mit einem Submodell: Das Hauptmodell M ist in Abbildung 8.2 abgebildet, das verfeinernde Modell S in Abbildung 8.3. Die Verfeinerung wird folgendermaßen durchgeführt: Immer, wenn das Hauptmodell confirm ausgeben würde, wird stattdessen das Submodell ausgeführt. Abhängig vom Ergebnis des letzten Schrittes des Submodells wird im Hauptmodell weiter gemacht.

Betrachten wir einen möglichen Trace des kompositen Modells, z.B *begin.new.exit.ok*. Wie sieht nun ein Trace mit Vorgeschichte aus? Für die ersten Stimuli ist das kein Problem: $begin\langle M : begin\rangle.new\langle M : new\rangle.exit\langle M : exit\rangle$. Allerdings braucht man für *ok* eine komplexere Vorgeschichte: Zum einen muss M sowohl confirm ausgegeben haben als auch den *ok*-Schritt durchgeführt haben; andererseits wurde der *ok*-Stimulus ja von S verursacht. Aus diesem Grund hat man für *ok* die folgende Vorgeschichte: $\langle M : \mathsf{confirm}.ok, S : ok\rangle$. Andererseits garantiert diese Art der Vorgeschichten-Konstruktion, dass die starke Trace-Erhaltung stattfindet.

Das Problem bei solchen Vorgeschichten ist, dass die Zuweisung eines aufgetretenen Fehlers zu einer Transition eines der Submodelle nicht mehr eindeutig ist. Wir lösen das Problem dadurch, dass wir einen der Stimuli in einer Vorgeschichte als *aktiven Stimu-*

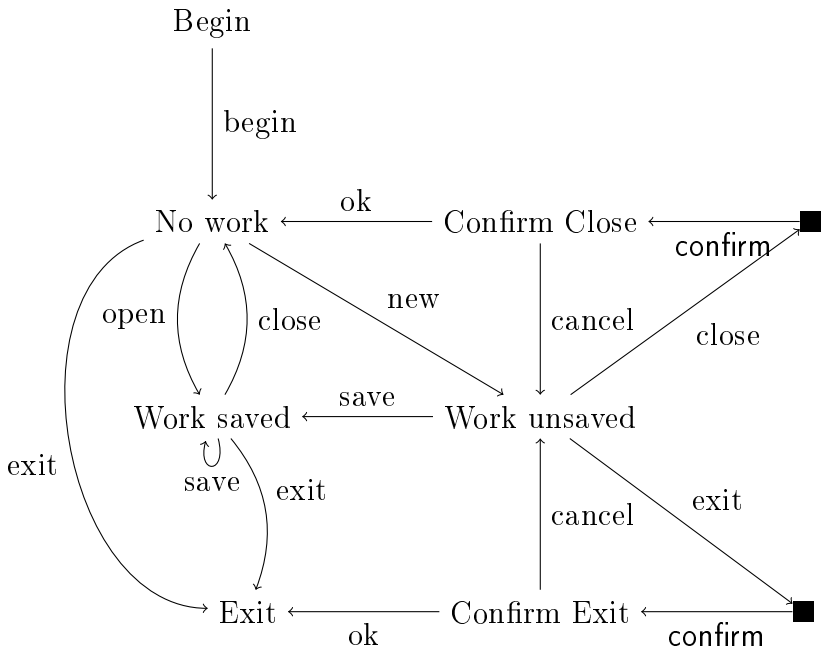

Abbildung 8.2: Ein zu verfeinerndes Benutzungsmodell (nach dem Beispiel-Modell von Prowell)

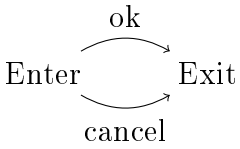

Abbildung 8.3: Das Verfeinerungsmodell

lus auszeichnen; tritt ein Fehler auf, so wird die „Schuld“ daran dem aktiven Stimulus zugewiesen.

Möglicherweise kann es nötig werden, mehrere aktive Stimuli für eine Vorgeschichte festzulegen; das wird allerdings operatorabhängig geschehen müssen, und momentan liegt uns kein Beispiel vor, wo das der Fall wäre.

8.3.2 Abbildung auf Submodelle

Bei manchen Operatoren (beispielsweise „Robustness“) wird man manchen Stimuli keine Vorgeschichte zuweisen können; in diesem Fall muss man mit Zuverlässigkeitsaussagen entsprechend vorsichtig sein. Man kann insbesondere nur solche Testfälle auf ein Submodell zurück abbilden, bei denen entweder kein Stopping-Fehler auftrat, oder bei denen der Stopping-Fehler genau in diesem Submodell auftrat. Trotz allem muss man hier ganz besonders beachten, dass die Stimuli, die Fehler offenbaren, nicht unbedingt diejenigen sind, die die Fehlerursache sind.

Aus den oben genannten Punkten kann man also wie folgt ein Verfahren konstruieren, um die Testergebnisse von einem kompositen Modell auf ein Submodell abzubilden. Wir stellen dazu zwei Vorbedingungen: Es muss starke Trace-Erhaltung für das Submodell gelten und das Submodell muss terminierend sein.

1. Wurde der Testfall mit einem stopping failure abgebrochen, so muss man im Allgemeinen eine Kompatibilitätsbedingung betrachten. Sie besagt, dass eine Abbildung auf das Submodell nur dann möglich ist, wenn die Trace-Projektion des Testfalls bis zum Punkt des Abbruchs gleich der Trace-Projektion des gesamten Testfalls ist, oder wenn der aktive Stimulus zur Zeit des Abbruchs zum Submodell gehört.

2. Man bestimmt zunächst durch Trace-Projektion des gesamten Testfalls einen Testfall des Teil-Modells; wir bezeichnen diesen als Teil-Testfall.

   Aufgrund der starken Trace-Erhaltung können wir eine (Multi)menge solcher Testfälle als Stichprobe des Submodells ansehen.

3. Für jeden Fehler des (kompositen) Testfalls bestimmt man den jeweils aktiven Stimulus. Ist dieser vom untersuchten Submodell verursacht, so findet man den entsprechenden Testschritt im Teil-Testfall und markiert diesen als fehlerhaft. Handelt es sich sogar um einen stopping failure, so markiert man die entsprechende Stelle im Teil-Testfall als fehlerhaft.

Ein letzter Punkt, den es zu beachten gilt, ist die geänderte Interpretation der Single Use Reliability: Während dieser Wert bisher die Aussage „Bei einem zufällig gewählten Testfall werde ich mit dieser Wahrscheinlichkeit keinen Fehler aufdecken“bedeutete, muss er jetzt für Submodelle als „Bei einem zufällig gewählten Testfall und der Komposition entsprechendem Umgebungsverhalten werde ich mit dieser Wahrscheinlichkeit keinen Fehler aufdecken“.

8.4 Feststellung von Modelleigenschaften

Im Folgenden sollen abschließend noch Verfahren beschrieben werden, um festzustellen, ob ein Modell terminierend, nicht-terminierend bzw. fair bezüglich einer Menge von Stimuli ist. Wir formulieren jedes dieser Probleme zunächst als stochastische Aussage über die zugrunde liegende Markov-Kette und geben anschließend einen einfachen Algorithmus an, um diese Frage zu lösen.

8.4.1 Terminierendes Modell

Um zu entscheiden, ob ein Modell M terminierend ist, müssen wir entscheiden, ob $P_M(\Sigma^* \mathsf{END}^\omega) = 1$ ist. Da es nach Voraussetzung höchstens einen Endzustand gibt, können wir die Bedingung wie folgt darstellen:

1. Es gibt einen Endzustand.
2. Der Endzustand ist der einzige absorbierende Zustand.

Aufgrund eines Satzes über absorbierende Markov-Ketten genügt es, nachzuweisen, dass in jedem Nicht-Endzustand s gilt: $P(s \rightarrow^* \mathsf{end}) > 0$.

Somit sieht der Überprüfungsalgorithmus wie folgt aus:

1. Zuerst prüfe, ob genau ein absorbierender Zustand vorliegt, und ob dieser der Endzustand ist.
2. Für jeden Knoten, suche einen Weg mit Wahrscheinlichkeit $\neq 0$ zum Endzustand (das geht z.B. mit Hilfe einer Tiefensuche).

Wenn beide Bedingungen erfüllt sind, ist das Modell terminierend.

8.4.2 Nicht-terminierendes Modell

Um zu entscheiden, ob ein Modell M nicht-terminierend ist, genügt es, zu überprüfen, ob M einen Endzustand enthält. Wenn nicht, ist es nicht-terminierend.

8.4.3 Fairness bezüglich Stimulusmenge

Sei Σ eine Menge von Stimuli, und M ein nicht-terminierendes Modell. Die Stimuli, die von M ausgegeben werden, nennen wir Ω. Definiere $|t|_\sigma$ als die Zahl der Vorkommen von σ in t (hierbei ist ∞ ein gültiger Wert). Dann ist M fair bezüglich Σ, wenn $P(|t|_\sigma = \infty) = 1$ für alle $\sigma \in \Sigma$ ist.

TODO Saubere Begründung des Überprüfungsalgorithmus:

1. Alternative Charakterisierung: Immer eins mehr für jedes σ
2. Argumentation: $P(\sigma \text{ kommt}) > 0$ genügt (Argument über absorbierende Markov-Ketten? Muss ausformuliert werden, Modell-Unrolling, σ-Transitionen gehen in absorbierenden Zustand)

3. Pfad zu σ mit Wahrscheinlichkeit > 0 finden

Der Überprüfungsalgorithmus sieht also wie folgt aus:

- Für jeden Zustand bestimme die Menge der Stimuli, die an seinen ausgehenden Kanten mit Wahrscheinlichkeit $\neq 0$ erzeugt werden.
- Vereinige die Stimulusmenge jedes Zustands mit denen seiner mit Wahrscheinlichkeit $\neq 0$ erreichbaren Nachbarn.
- Wiederhole den letzten Schritt, bis keine Änderung mehr stattfindet.

Wenn dann Σ in der Stimulusmenge jedes Zustands enthalten ist, so ist das Modell fair bezüglich Σ.

8.5 Analyse von Nebenläufigkeitsproblemen

Bei manchen Operatoren (konkret: Barrier und Suspend) ergeben sich zusätzliche Nebenläufigkeits-Probleme, die mit den bisher beschriebenen Modell-Analysen nicht aufdeckbar sind. Für diese beiden Operatoren müssen also spezielle Modell-Analysen beschrieben werden.

8.5.1 Monopol-Freiheit

Beim suspend-Operator besteht die Gefahr, dass ein Prozeß das stop-Signal an den Operator sendet, aber danach nie das resume-Signal ausgibt. Glücklicherweise ist hier die Modellanalyse sehr einfach: Man muss nur überprüfen, ob von jedem Zustand, der mit Hilfe einer stop-Kante erreicht werden kann, mit Wahrscheinlichkeit 1 ein Pfad gewählt wird, in dem resume ausgegeben wird. Dazu benutzt man im Wesentlichen den Algorithmus der Fairness-Analyse (mit $\Sigma = \{\text{resume}\}$), überprüft aber am Ende nur die Zustände, die eine Eingangskante mit der Beschriftung stop haben.

8.5.2 Deadlock-Freiheit

Die Analyse der Deadlock-Freiheit beim Barrier-Operator ist etwas komplizierter. Prinzipiell gäbe es eine sehr einfache Möglichkeit, vollständige Deadlocks zu erkennen, bei denen alle Prozesse beteiligt sind (man müsste nur überprüfen, ob alle beteiligten Prozesse blockiert sind), allerdings würde man auf diesem Weg niemals Livelocks entdecken, also Fälle, in denen ein Teil der Prozesse unwiderbringlich blockiert ist, aber einige der beteiligten Prozesse trotzdem noch laufen.

Für den Fall, dass alle beteiligten Modelle terminierend sind, gibt es eine einfache Lösung: Man berechnet das komposite Modell und überprüft, ob von jedem Zustand aus der Endzustand erreicht werden kann. Ist das nicht der Fall, muss ein Deadlock vorliegen, denn in diesem Fall kann mindestens einer der Prozesse den Endzustand nicht erreichen.

Der allgemeine Fall ist etwas komplexer und muss deshalb auf einem anderen Weg gelöst werden. Wir können einen Deadlock in unserem Modell wie folgt garantieren: Ein Modell hat einen Deadlock in Zustand s, wenn ein Submodell von s ausgehend niemals einen Zug machen kann. Damit können wir die Analyse der Deadlock-Freiheit in diesem Fall auf die π-Kalkül-Ebene herunterbrechen, wo noch genug Informationen für diese Untersuchung vorliegen. Das Vorgehen sieht wie folgt aus:

1. Bestimme die Markov-Kette, die zum kompositen Modell gehört, und speichere für jeden Zustand den zugehörigen π-Kalkül-Prozeßausdruck.

2. Für jeden dieser Ausdrücke ermittle die Menge der benutzten Modell-Eingabekanäle und speichere, von welchen Modellen dieser Ausdruck Stimuli annimmt (die sogenannte Annahme-Menge).

3. Füge zu jeder Annahme-Menge die Annahme-Mengen der Nachfolgerzustände hinzu, bis keine Änderung mehr auftritt.

4. Wenn alle Zustände die Menge aller Eingabe-Modelle als Annahme-Menge hat, ist das Modell deadlock-frei, sonst existiert ein Deadlock.

9 Ermittlung von Stimulationssequenzen für vorgegebene Testfälle

9.1 Problemstellung

Oftmals wird (beispielsweise in Standards) verlangt, bestimmte vorgegebene Testfälle durchzuführen. Soll nun der Test eines Systems mit Hilfe des statistischen Testens geschehen, so klafft zwischen der Modellierung der Testfälle als Folge von Stimuli des Systems und den abstrakt vorgegebenen Testfällen eine Lücke. Es stellt sich die Frage, ob es möglich ist, die vorgegebenen Testfälle auf die Ebene des Testmodells abzubilden.

9.2 Annahmen über die vorgegebenen Testfälle

Wir gehen davon aus, dass ein vorgegebener Testfall als eine Folge von Schritt beschrieben ist. Jeder Schritt beschreibt eine Menge von Eingaben an das System. Zusätzlich werden auch erwartet Ausgaben und Umgebungsbedingungen beschrieben; wir nehmen an, dass diese Informationen sich nicht auf die von Stimulationsmodell beschriebenen Eingaben beziehen, sondern unabhängig davon betrachtet werden können.

TODO Beispiel

9.3 Abbildung eines Testschritts auf eine Stimulus-Folge

Der Kern des hier beschriebenen Verfahrens ist die Abbildung der einzelnen Eingaben und Testschritte auf Stimuli und Stimulationssequenzen. Dazu machen wir die folgende Annahme:

Jede im vorgegebenen Testfall beschriebene Eingabe kann in eine oder mehrere Folgen von Stimuli des Testmodells umgesetzt werden.

Wir können also annehmen, dass wir zu jeder Eingabe einen endlichen Automaten oder regulären Ausdruck angeben können, der die Stimulus-Repräsentation dieser Eingabe beschreibt. TODO geeignete Darstellung für Anwender? Jeder Darstellung einer Eingabe wird ein Name zugeordnet, welcher den dahinterliegenden Automaten repräsentiert.

Weiterhin nehmen wir an, dass wir zu jedem Testschritt beschreiben können, in welcher Reihenfolge die Eingaben gemacht werden; möglicherweise können die Eingaben

auch überlappen. Als Darstellung für den Anwender schlagen wir folgende Syntax vor: Wir beschreiben die Eingabe mit Hilfe regulärer Ausdrücke über den Eingabe-Namen, erlauben aber auch zusätzlich noch die Operatoren || (beliebiges Interleaving) und <Name1, ..., Namen> (führe jedes der Modelle genau einmal aus, beliebige Reihenfolge).

Durch Auswertung dieses Ausdrucks erhalten wir wiederum einen Automaten für jeden Testschritt.

9.4 Abbildung eines Testfalls

Nachdem beschrieben wurde, wie Testschritte abgebildet werden, ist die Abbildung eines Testfalls prinzipiell recht einfach: Man fügt einfach die Automaten, die die einzelnen Testschritte beschreiben, sequentiell zusammen. Um allerdings zu garantieren, dass ein Testfall in der Form generiert wird, wie im Stimulationsmodell beschrieben wird, sind noch zwei weitere Schritte nötig: 1. Möglicherweise ist zwar kein einziger Trace des Automaten ein gültiger Testfall, durch Einfügen gewisser „unkritischer“ Stimuli in einen Trace kann aber ein gültiger Testfall erzeugt werden. Deshalb kann der Testfall-Automat mit einer Menge unkritischer Stimuli „aufgeblasen“ werden, indem man in jedem Zustand eine Menge von Selbstkanten einfügt, die mit den unkritischen Stimuli beschriftet sind. 2. Die Sprache der Stimulus-Sequenzen, die den Testfall implementieren, muss noch mit der Menge der Traces des Stimulationsmodells geschnitten werden. Dazu bildet man den dem Stimulations-Modell zugrunde liegenden endlichen Automaten und konstruiert daraus den zum Schnitt der Sprachen gehörenden Produktautomaten.

Ergebnis dieser Berechnung ist ein endlicher Automat, dessen Traces mögliche Konkretisierungen des abstrakten Testfalls sind. Ist die Ausgabesprache des Automaten leer, so existiert keine Konkretisierung.

Wenn man möchte, kann man auf diesem Weg sogar ein Markov-Modell erzeugen: Da man jeder Kante des Produktautomaten eine Transition des Markov-Modells zuordnen kann, kann man auch die Wahrscheinlichkeit der Transition übernehmen. Wenn man anschließend die Wahrscheinlichkeitssummen für jeden Zustand normalisiert, erhält man ein Markov-Benutzungsmodell. Allerdings gilt hier definitiv keine Form der Trace-Erhaltung zum Ursprungsmodell. Aus diesem Grund sollte man nur mit äußerster Vorsicht die hier generierten Testfälle in die Berechnung der Zuverlässigkeit des Systems einbeziehen.

Literaturverzeichnis

[BEH+09a] Thomas Bauer, Robert Eschbach, Tanvir Hussain, Johannes Kloos, and Fabian Zimmermann. Komposition von benutzungsmodellen: Anforderungen und konzepte. Iese-report, Fraunhofer IESE, 2009.

[BEH+09b] Thomas Bauer, Robert Eschbach, Tanvir Hussain, Johannes Kloos, and Fabian Zimmermann. Komposition von benutzungsmodellen: Operatoren und anwendungsbeispiel. Technical report, Fraunhofer IESE, 2009.

[KLN07] Céline Kuttler, Cédric Lhoussaine, and Joachim Niehren. A stochastic pi calculus for concurrent objects. In *Second International Conference on Algebraic Biology*, volume 4545 of *Lecture Notes in Computer Science*, pages 232–246. Springer Verlag, Juli 2007.

[Mil99] Robin Milner. *Communicating and Mobile Systems: The pi-Calculus*. Cambridge University Press, 1999.

Dokumenten Information

Titel:	Komposition von Benutzungsmodellen: Spezifikation und Testtechniken
Datum:	Januar 2009
Report:	IESE-114.09/D
Status:	Final
Klassifikation:	Öffentlich