MISTRAL

MIcro Simulation, TRansformation, and Analysis Language

Sprachdefinition und Handbuch

Technische Universität Darmstadt
Fachgebiet Statistik und Ökonometrie
Univers.-Prof. Dr. Hans-Dieter Heike

Prof. Dr. Thomas Sauerbier

Version 3.0

29.8.2001

Inhaltsverzeichnis

1 Einführung

Ein Hauptproblem der Mikrosimulation besteht seit jeher darin, daß es keine Standard-Software für diesen Einsatzbereich gibt. Im Bereich makroökonomischer Modelle steht dem Anwender eine Vielzahl von Systemen zur Verfügung, mit denen er seine Modelle auf Anwenderebene eingeben, schätzen und simulieren kann. Die Simulation mikroökonomischer Modelle erfordert jedoch nach wie vor eine Implementierung in einer normalen Programmiersprache. Hieraus ergeben sich vor allem zwei Probleme:

Zum einen unterstützen die vorhandenen Sprachen wichtige Belange der Mikrosimulation nicht oder nur sehr unzureichend (z.B. die zufällige Reihenfolge beim Abarbeiten der demographischen Module). Diese müssen dann mühevoll selbst implementiert werden, so daß ein großer Teil der verfügbaren Zeit der eigentlichen Aufgabe, nämlich dem Erstellen des ökonomischen Modells, entzogen wird.

Zum anderen wird eine Vielzahl von Hilfsfunktionen benötigt, deren Programmumfang den des eigentlichen Modells um ein Vielfaches übertrifft. Es handelt sich dabei z.B. um die Handhabung von Datenbeständen, die Reportgenerierung und Analyse. Selbst wenn hierfür ein arbeitsfähiges System zur Verfügung steht, muß sich der Modellimplementierer damit beschäftigen, die Schnittstellen zwischen seinem Modell und dem vorhandenen System zu realisieren. Neben dem enormen Einarbeitungsaufwand in ein fremdes Software-System besteht zudem die Notwendigkeit, sich der für das Simulationssystem verwendeten Sprache zu bedienen. Die so erstellten Modelle sind damit - selbst wenn sie in einer gängigen Programmiersprache geschrieben wurden - nicht portierbar.

Mit MISTRAL (**MI**cro **S**imulation, **TR**ansformation, and **A**nalysis **L**anguage) steht eine spezielle Mikrosimulationssprache zur Verfügung, die einerseits auf die Belange dieser Anwendung zugeschnitten ist und andererseits unabhängig von der für das Simulationssystem verwendeten Programmiersprache angewandt werden kann.

Wegen ihrer klaren Struktur und großen Verbreitung wurde MISTRAL stark an die Programmiersprache PASCAL angelehnt. Voraussetzung für die Anwendung von MISTRAL sind zumindest grundlegende Kenntnisse in einer höheren Programmiersprache, wie sie im allgemeinen auch innerhalb der universitären Ausbildung von Ökonomen vermittelt werden. Die Sprache unterstützt eine selbstdokumentierende Schreibweise, die es selbst Ökonomen mit nur geringen Programmiererfahrungen ermöglichen sollte, Programme auch ohne Kenntnis von MISTRAL zu lesen und in ihren Inhalt zu verstehen. Damit eignet sich MISTRAL nicht nur als Simulations-, sondern auch als Dokumentationssprache, mit der Modelle exakter als mit verbalen Mitteln beschrieben werden können.

Das vorliegende Papier dient vor allem dazu, die Sprache MISTRAL zu definieren. Dabei beschränken sich die Ausführungen nicht auf eine reine Syntaxbeschreibung (diese ist dem Anhang zu entnehmen). Vielmehr werden die Bedeutung der einzelnen Sprachelemente und ihre Wirkungen dargestellt. Soweit es sich um Elemente handelt, die direkt von PASCAL übernommen wurden, sind die Ausführungen sehr kurz gehalten. Hier sei auf die reichlich vorhandene Standardliteratur zu PASCAL verwiesen. Aufgrund der engen Anbindung an ein Simulationssystem werden insbesondere die Wirkungen bezüglich des Simulationsverlaufs beschrieben.

Dieses Papier soll künftigen Anwendern der Sprache als Einführung und Nachschlagewerk zu MISTRAL dienen. Für die praktische Anwendung muß jedoch die Kenntnis höherer Programmiersprachen sowie der Mikrosimulation selbst vorausgesetzt werden.

2 Grundlage und Motivation der Sprache

In diesem Kapitel wird die Ausgangssituation beschrieben, die der Definition der Sprache zugrunde lag. Ziel dieser Ausführungen ist es einerseits, die Wahl (bzw. das Weglassen) bestimmter Sprachelemente zu begründen. Damit kann die Sprache besser verstanden und auch praktisch eingesetzt werden. Andererseits können künftige Erweiterungen der Sprache in den vorhandenen Rahmen eingefügt werden, ohne diesen zu sprengen.

2.1 Notwendigkeit einer neuen Sprache

Wenn man hört, daß eine neue Programmiersprache entwickelt werden soll - noch dazu in Anlehnung an eine vorhandene -, stellt sich spontan die Frage: Warum dann nicht gleich die vorhandene nehmen?

Der Grund liegt nicht nur an der besseren Anpassung an die spezielle Aufgabe, sondern auch in ganz praktischen Notwendigkeiten, die die Realisierung und Verteilung eines konkreten Produktes betreffen. Da diese Gründe letztlich entscheidend für die völlige Neuentwicklung (z.B. anstelle einer Bibliothek mit speziellen Mikrosimulationsprozeduren und -funktionen) waren, werden sie hier kurz dargestellt.

Zunächst ist festzustellen, daß der Ausdruck "eine vorhandene Sprache verwenden" zu ungenau ist. Hierbei sind zwei grundsätzlich verschiedene Dinge zu unterscheiden: der Compiler und die Sprache.

2.1.1 Verwendung eines existierenden Compilers

Die Integration eines fremden Compilers in eine abgeschlossene Simulationsumgebung ist schwierig. Dabei besteht auch eine große Abhängigkeit von den Änderungen des Compilers, die bei jedem Update zu erwarten sind. Aufgrund der Marketingpolitik der meisten Hersteller sind solche Änderungen oft mindestens einmal pro Jahr zu erwarten, so daß ein dauernder äußerer Zwang zur Anpassung besteht, der speziell im Forschungsbereich nicht realisiert werden kann.

Für den Anwender besteht der Nachteil, daß er einen Programmteil schreiben soll, der sich später nahtlos in das übrige Simulationssystem einfügt. Hierbei ist zu bedenken, daß ein Mikrosimulationsmodell nur wenige KByte umfaßt (bis ca. 50 KB), während ein Mikrosimulationssystem in der Regel aus mehr als 1 MByte Code bestehen dürfte. Der Standardcompiler ist nicht imstande, die korrekte - auch semantische - Einbindung des Modells in das Gesamt-System zu prüfen. Das Simulationssystem kann hierbei auch keinerlei Hilfe bieten, da die Analyse des Modelltextes ja gerade auf den externen Compiler verlagert wurde. Zusätzliche Prüfungen würden es wieder erforderlich machen, doch einen eigenen Übersetzer zu entwikkeln.

Nachteilig ist zudem die Notwendigkeit für den Anwender, sein Modell in derselben Sprache zu formulieren, in der das Gesamtsystem entwickelt wurde. Die Anforderungen für beide Bereiche unterscheiden sich jedoch deutlich, so daß der Benutzer eventuell mit einer nicht problemadäquaten Sprache arbeiten muß. Es ist weiterhin unmöglich, ein Modell ohne völlige Neuimplementierung auf zwei unterschiedlichen Simulatoren laufen zu lassen. Wenn nicht zufällig Erfahrungen mit genau dieser Sprache und diesem Compiler vorliegen, entsteht ein

enormer Einarbeitungsaufwand. Erschwert wird dies noch dadurch, daß der typische Modellierer eher Ökonom als Informatiker ist.

Ein weiterer Problembereich betrifft überwiegend rechtliche Probleme. Um nämlich sein Modell entwickeln zu können, benötigt der Anwender genau den gleichen Compiler (meist mit gleichem Release-Stand) wie der Programmierer des Simulationssystems. Die Weitergabe des Compilers ist aus lizenzrechtlichen Gründen unmöglich. Der Anwender wäre also gezwungen, sich selbst den gleichen Compiler wie der Anbieter des Simulators zu kaufen und immer den zum Simulator passenden Release-Stand zu halten. Neben praktischen Problemen wird dies meist auch am Preis scheitern, da Hochschulen oft erhebliche Rabatte erhalten, während die Anwender oft Tausende von DM bezahlen müßten. Die Verwendung von Studentenversionen für häusliche Übungen scheitert damit ebenso wie die Weitergabe von Demoversionen an Dritte.

Die genannten Probleme zeigen, daß die Verwendung eines vorhandenen (kommerziellen) Compilers bereits innerhalb eines eng umgrenzten Forschungsbereichs (z.B. eines Lehrstuhls) schwierig ist. Die Weitergabe des Simulators an Dritte scheidet sogar nahezu aus.

2.1.2 Verwendung einer existierenden Sprache

Im letzten Abschnitt wurde deutlich, daß vorhandene Compiler nicht verwendet werden können, wenn ein auch von Dritten verwendbares Produkt entstehen soll. Dies bedeutet, daß ohnehin die Entwicklung eines vollständigen Compilers notwendig wird. In diesem Fall stellt sich die Frage, welche Vorteile es bringt, eine bestehende Sprache zu verwenden. Hier können vor allem zwei Argumente angeführt werden:

Zum einen kann der Entwickler auf eine vorhandene, erprobte und meist sehr ausführliche Sprachdefinition zurückgreifen, so daß der Aufwand für eine Neudefinition entfällt. Zudem werden so viele mögliche Fehler vermieden.

Für den Anwender ergibt sich bei geeigneter Sprachwahl zum anderen oft der Vorteil, daß er die Sprache bereits beherrscht oder sich anhand der reichlich vorhandenen Literatur aneignen kann. Bei einer völlig neuen Sprache müßte parallel auch ein regelrechtes Lehrbuch für diese Sprache geschrieben werden.

Vorhandene Sprachen besitzen jedoch einen erheblichen Umfang und decken damit auch viele Bereiche ab, die innerhalb der Mikrosimulation nicht benötigt werden. Beispiele sind interaktive Ein- und Ausgabefunktionen, Dateioperationen und maschinennahe Befehle. Auch bei den verbleibenden Bereichen ist es oft sinnvoll, Abstriche zu machen; in PASCAL z.B. variante Records und die Unterscheidung von je fünf Untertypen für INTEGER und REAL (z.B. BYTE, SHORTINT, LONGINT usw.). Läßt man diese Teile jedoch weg, so verändert man die vorhandene Sprachdefinition und hat de facto eine neue Sprache definiert. Das Hinzufügen einiger neuer Sprachelemente ist dann nur noch eine zusätzliche Veränderung.

Dieser Ansatz, nämlich eine neue Sprache in Anlehnung an eine existierende zu definieren, vereint weitgehend alle Vorteile in sich:

- Ein erprobtes, etabliertes Sprachkonzept bildet die Basis.
- Der Aufwand für eine Neudefinition muß nur insoweit getrieben werden, wie es für eine vereinfachte Implementierung oder eine Funktionserweiterung sinnvoll ist.
- Der Einarbeitungsaufwand für den Anwender ist bei Kenntnis der Basissprache sehr gering. Anderenfalls steht umfangreiche Literatur zum Erlernen zur Verfügung.

- Die Sprache und damit der Compiler wird nur so umfangreich, wie es für den jeweiligen Zweck wirklich erforderlich ist.

Eine völlige Neuentwicklung einer Spezialsprache bietet sich nur dort an, wo ein sehr spezielles, eng begrenztes Gebiet abzudecken ist und keine der bestehenden Sprachen angemessen erscheint. Es besteht dann jedoch die Gefahr, daß der Entwickler der Sprache damit den Rahmen für den späteren Anwender so eng steckt, daß künftige Anforderungen nicht abgedeckt werden können, weil sie schon an konzeptionellen Grenzen scheitern. Bei einer universellen Programmiersprache als Basis besteht diese Gefahr kaum.

2.1.3 Fazit

Die Verwendung eines vorhandenen Compilers bereitet immer dann Probleme, wenn der Systementwickler und der Modellentwickler nicht identisch sind. Eine Weitergabe eines solchen Systems scheitert de facto an praktischen, finanziellen und lizenzrechtlichen Gründen.

Um ein von Dritten nutzbares System zu entwickeln, das ohne Einschränkungen weitergegeben werden kann, muß ein Übersetzer (Compiler oder Interpreter) für die Modellierungssprache integrierter Bestandteil des Simulationssystems sein.

Die Modellierungssprache sollte auf der Grundlage einer geeigneten vorhandenen Universalsprache definiert werden. Dabei werden im allgemeinen größere Teile der ursprünglichen Sprache entfallen, während einige zusätzliche Konstrukte aufgenommen werden, die auf das Spezialgebiet zugeschnitten sind.

Für die Mikrosimulationssprache MISTRAL wurde PASCAL als geeignete Sprachbasis angesehen. In den folgenden Abschnitten werden die Bedingungen dargestellt, die dieser Entscheidung zugrunde lagen. Ihre Kenntnis hilft, das gesamte Sprachkonzept besser zu verstehen.

2.2 Fachliche Ausgangssituation

Der Anwender von MISTRAL ist typischerweise der Ökonom, der ein von ihm entworfenes Mikromodell eingeben und simulieren will. Für diese Aufgabe müssen zwar grundlegende Programmierkenntnisse vorausgesetzt werden, jedoch nicht auf dem Niveau eines normalen Programmierers.

Mikromodelle werden nach wie vor häufig in Form von Ablauf- bzw. Flußdiagrammen dargestellt. Hierbei werden zwar die Datenoperationen sehr ausführlich beschrieben, die eigentlichen Steuermechanismen (z.B. die Abfolge der Simulationseinheiten oder der Modellteile) werden jedoch meist als vorhanden vorausgesetzt. Die Sprache soll die Eingabe auf dieser Ebene unterstützen und möglichst weitgehend die Abläufe der Simulation selbst vor dem Benutzer verbergen.

Dies setzt die Entscheidung für eine bestimmte Form der Simulation voraus, die für den weiteren Ablauf implizit unterstellt wird. Der zugrundeliegende Simulationstyp ist die dynamische Querschnittssimulation. Das bedeutet, daß eine Menge von Simulationseinheiten bzw. Mikroobjekten (z.B. Haushalte und Personen) unter Beibehaltung ihrer Identität jeweils um eine Periode fortgeschrieben werden. Bei Simulationen über einen längeren Zeitraum werden zunächst alle Mikroobjekte für einen Periode simuliert. Erst danach werden die noch vorhandenen Einheiten um eine weitere Periode fortgeschrieben.

Die Modelle werden meist in Einheiten, die sogenannten Module, unterteilt (z.B. Tod, Geburt, Heirat usw.), die getrennt - z.T. von verschiedenen Anwendern - definiert werden. Die Module selbst besitzen typischerweise einen Umfang von 20 bis 200 Zeilen; ein gesamtes Programm dürfte - je nach Erklärungsumfang - einige hundert bis wenige tausend Zeilen umfassen.

2.3 Sonstige Voraussetzungen

Damit die hier definierte Sprache nicht nur eine akademische Fingerübung bleibt, sondern ein tatsächlich verwendbares Produkt werden kann, ist immer die Realisierbarkeit zu berücksichtigen. Dies bedeutet konkret, daß es mit geringem Personalaufwand möglich sein muß, in relativ kurzer Zeit einen funktionstüchtigen Compiler für die definierte Sprache zu entwickeln und in das vorhandene Simulationssystem zu integrieren (max. 1 Mannjahr).

Um dies zu erreichen, wurden vor allem folgende Dinge berücksichtigt:

- Die Sprache ist so einfach wie möglich zu halten.
- Für die Anwendung entbehrliche oder gar unnötige Konzepte entfallen.
- Die Syntax der Sprache ist so zu gestalten, daß eine einfache Realisierung des Compilers möglich ist.

2.4 Folgerungen für die Sprache

Viele der aktuellen Konzepte von Programmiersprachen dienen vor allem dazu, die Größe bzw. Komplexität heutiger Software-Projekte zu bewältigen. Beispielhaft seien hier Modulkonzepte, abstrakte Datentypen und Vererbung genannt. Diese Fähigkeiten belasten nicht nur den unerfahrenen Anwender solcher Sprachen mit einem enormen Einarbeitungsaufwand. Auch die Compiler für solche Sprachen sind nur noch von einem Team von Spezialisten mit einem Aufwand von vielen Mannjahren zu realisieren.

Der im letzten Kapitel angesprochene Programmumfang von maximal wenigen tausend Zeilen Code ist nach heutigen Maßstäben als klein zu bezeichnen. Die Entwickler von universellen Programmiersprachen orientieren sich hingegen zunehmend an der Eignung ihrer Systeme für große Projekte (über 100.000 Zeilen Code).

Auch die Vererbung - ein wesentlicher Bestandteil der Objektorientierung - setzt voraus, daß eine entsprechende Anzahl von Klassen vorhanden ist, zwischen denen Vererbungsbeziehungen bestehen können. Bei der Mikrosimulation des Haushaltssektors gibt es jedoch nur zwei Klassen (Haushalte und Personen), zwischen denen keine Vererbungsabhängigkeit hergeleitet werden kann. Gleiches gilt für Modelle des Unternehmenssektors, bei denen eine (Unternehmen) oder zwei (zusätzlich Branchen) Klassen zu erwarten sind. Da die Vererbung - gegebenenfalls zusammen mit spätem Binden - einen enormen Mehraufwand beim Implementieren einer Sprache bedeutet - dem zudem für diese Anwendung kein Nutzen gegenüber steht -, wurde bei MISTRAL darauf verzichtet.

Für die Aufgabe und den angestrebten Benutzerkreis ist vor allem die Einfachheit und rasche Erlernbarkeit der Sprache ein wesentliches Kriterium. Ebenso wichtig ist die Lesbarkeit der Sprache auch für Außenstehende, die mit MISTRAL selbst nicht oder kaum vertraut sind. Die Syntax sollte sich deshalb an bekannten Strukturen orientieren, wobei PASCAL aufgrund sei-

ner klaren Struktur und seiner Verbreitung sehr geeignet erscheint. Neuere PASCAL-Versionen unterstützen zudem die Objektorientierung, die in Teilen auch in MISTRAL realisiert wurde. Auch hier konnte PASCAL als Vorlage dienen.

2.5 Aktuelle Sprachdefinition

2.5.1 Allgemeines

Die Syntax der Sprache MISTRAL ist an die der Sprache PASCAL angelehnt, jedoch speziell auf die Bedürfnisse der Mikrosimulation zugeschnitten.

Dies bedeutet einerseits, daß sie nicht alle Elemente von PASCAL enthält. Ausgenommen sind z.B. die meisten Dateiausgabe-Operationen sowie eine größere Menge von PASCAL-Erweiterungen, wie sie z.B. in den neueren Versionen von Turbo-PASCAL zu finden sind.

Ergänzt wurden hingegen Konstrukte, die für die Handhabung und Simulation von Mikroobjekten benötigt werden. Hierzu gehören z.B. spezielle Vereinbarungen von Objekttypen, Schleifen über Objektmengen sowie eine zufällige Abarbeitungsreihenfolge.

Insgesamt ist MISTRAL eine prozedurale Sprache mit objektbasierenden Erweiterungen. Diese Erweiterungen umfassen in vereinfachter Form das Klassenkonzept, die Kapselung und den Polymorphismus (Überladen von Operatoren). Verzichtet wurde auf die Vererbung.

2.5.2 Unterschiede und Gemeinsamkeiten zu PASCAL im Überblick

Um dem Leser einen schnellen Überblick über die Unterschiede zwischen PASCAL und MISTRAL zu vermitteln, werden hier die wesentlichen Bereiche schlaglichtartig beleuchtet:

Datentypen

INTEGER- und REAL-Typen sind vorhanden, jedoch ohne die mehr maschinennahen Unterscheidungen in SHORTINT, LONGINT usw. Zusätzlich ist die Möglichkeit vorhanden, den Wertebereich auf positive Werte einzuschränken und Missing-Values aufzunehmen. Ebenfalls vorhanden sind BOOLEAN und STRINGS.

Aufzählungstypen in MISTRAL besitzen mehr Möglichkeiten als in PASCAL. So sind z.B. Aliase für Ausprägungen möglich sowie eine generelle DOES_NOT_APPLY-Ausprägung.

Der zentrale Datentyp in MISTRAL ist die Klasse, die weitgehend der objektorientierten Record-Variante OBJECT in Turbo-PASCAL (ab 5.5) entspricht. Nicht vorhanden sind jedoch Mechanismen wie Vererbung oder Private-Komponenten. Ein eigener Record-Typ ist daneben nicht vorhanden.

Ein Äquivalent zum Mengentyp SET in PASCAL ist in MISTRAL nicht für elementare Datentypen, sondern nur für Objekte definiert. Dieser besitzt jedoch erheblich mehr Möglichkeiten, z.B. SQL-ähnliche Abfragekonstrukte und eine Schleife über alle Objekte (foreach .. in .. do).

Arrays und FILE-Typen sind nicht vorhanden, jedoch ein array-ähnlicher Vektor-Typ.

Zeigertypen im PASCAL-Sinne sind weder vorhanden noch notwendig, da Objekttypen immer Zeiger auf Objekte sind (ähnlich wie bei SMALLTALK).

Operatoren und Ausdrücke

MISTRAL besitzt weitgehend den gesamten Umfang mathematischer und logischer Operatoren und Standardfunktionen. Nicht vorhanden sind Bit-Operationen.

Anweisungen

Abgesehen von der Sprunganweisung sind alle PASCAL-Anweisungen vorhanden: Verbundanweisung, IF, CASE, FOR, WHILE und REPEAT. Zusätzlich wurden eine FOREACH-Anweisung (Schleife über alle Objekte einer Menge) sowie verschiedene Möglichkeiten zur Abarbeitung in zufälliger Reihenfolge (für die stochastische Simulation) geschaffen.

Unterprogramme

Zusätzlich zu normalen Prozeduren und Funktionen sowie deren objektgebundenen Entsprechungen (Methoden) gibt es mit Modulen und Super-Modulen weitere Varianten, die der inhaltlichen Strukturierung in der Mikrosimulation dienen. Parameter werden grundsätzlich als "call-by-value" übergeben, allerdings gibt es bezüglich der Typen für Parameter und Funktionswert keine Einschränkungen. Unterprogramme können geschachtelt definiert werden. Rekursion ist nicht zulässig.

Ein- und Ausgabe, Datei-Operationen

Die Schnittstelle zu den übrigen Teilen des Simulationssystems (z.B. Mikrodatenbasis, Fortschreibungsparameter) wird implizit realisiert. Für die Ausgabe stehen neben einem erweiterten WRITE-Befehl Reportfunktionen für Ereignisse und Datenabfragen zur Verfügung. Alle Ausgaben erfolgen in einen Report; die notwendigen Bildschirmausgaben werden vom Simulationssystem erzeugt. Dateioperationen beschränken sich auf das (komfortable) Einlesen von Textdateien. Für die Interaktion mit dem Benutzer sind keine Anweisungen vorhanden (z.B. READ).

2.6 Teilsprachen

Bei MISTRAL handelt es sich nicht um eine monolithische Sprache, sondern vielmehr um ein geschlossenes System von aufeinander aufbauenden Teilsprachen, wie Abbildung 2-1 zeigt.

Gemeinsame Basis aller Mikromodellierungsteile (d.h. außer den Zeitreihen) ist die **Typ-Definition**, in der die Aufzählungstypen (ENUM) festgelegt werden, die unter anderem für Attribute der Mikroobjekte benötigt werden. Ein Beispiel ist die Definition der Familienstände mit den Ausprägungen ledig, verheiratet, geschieden und verwitwet.

Die zweite Grundlage bildet die **Klassen-Definition**, in der festgelegt ist, welche Attribute eine Mikroobjektklasse besitzt und von welchem Typ sie jeweils sind. Sie stellt die wichtigste Verbindung zwischen der Mikrodatenbasis und dem Fortschreibungsalgorithmus dar.

Die verschachtelten Balken in der Abbildung rechts oben bilden den eigentlichen Sprachkern von MISTRAL, die **Micro Modelling Language** (**MML**). Mit der Abbildung wird verdeutlicht, daß es verschiedene Sprachebenen (Teilsprachen) gibt, wobei die oberen die jeweils darunterliegenden mit umfassen.

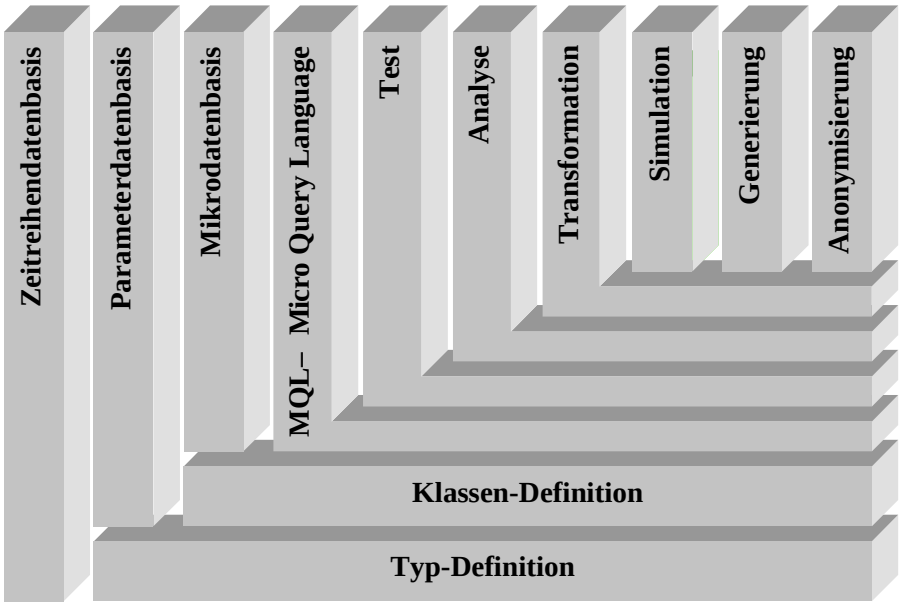

Abb. 2-1: Sprachebenen von MISTRAL

Die **Micro Query Language** (**MQL**) stellt ein Werkzeug dar, mit dem komfortabel auf Mikrodaten zugegriffen werden kann. Sie erlaubt nur lesenden Zugriff. MQL steht sowohl interaktive in verschiedenen Analysewerkzeugen des Mikrosimulators als auch innerhalb von MML-Programmen zur Verfügung. Als Ergebnis wird eine Menge von Objekten, eine ein- oder zweidimensionale Häufigkeitsverteilung oder ein einzelner Wert geliefert.

Die Sprachebene **Test** dient dazu, eine Mikrodatenbasis auf die Einhaltung bestimmter Regeln zu überprüfen. Sie besitzt ausschließlich lesenden Zugriff und erzeugt als Ausgabe einen Text. Wird kein Text erzeugt, wird dies vom Simulationssystem als "keine Fehler" interpretiert.

Die Ebene **Analyse** dient dazu, eine Mikrodatenbasis - ebenfalls nur mit lesendem Zugriff - umfassend zu analysieren. Als Ergebnis können neben einem Report (inkl. Häufigkeitstabellen usw.) auch Häufigkeitsverteilungen und Zeitreihenwerte (jedoch immer nur für die jeweils aktuelle Simulationsperiode) erzeugt werden. Diese stehen dann für die Auswertung mittels interaktiver Analysewerkzeuge zur Verfügung.

Die Ebene **Transformation** stellt im Gegensatz zu den übrigen keine eigenständige, als solche erkennbare Teilsprache dar, sondern umfaßt Befehle zum Verändern der Mikrodatenbasis, die in den darüberliegenden Ebenen verwendet werden.

Die wohl wichtigste Teilanwendung ist die **Simulation**. Durch sie wird eine Mikrodatenbasis von einer Ausgangsperiode t in die nächste Periode t+1 fortgeschrieben, wobei die Erhöhung der Periode automatisch durchgeführt wird. Die einzelnen Aktionen (z.B. Personen werden um ein Jahr älter, können Kinder bekommen oder auch sterben) lassen sich völlig frei im Simulationsprogramm definieren. Als Ergebnis werden also eine fortgeschriebene Mikrodatenbasis sowie in der Regel die Ausgaben der Analyseebene erzeugt.

Die Bereiche **Generierung** und **Anonymisierung** wurden nur zur Verdeutlichung der unterschiedlichen Aufgaben getrennt aufgeführt, sind jedoch nur zwei Anwendungen derselben Teilsprache. In beiden Fällen wird eine Mikrodatenbasis für eine bestimmte Periode erzeugt, wobei optional auf eine Parameterdatenbasis sowie externe Textdateien verwendet werden können. Bei der Generierung kann dabei auch auf eine bestehende Mikrodatenbasis zugegriffen werden, um z.B. fehlende Attribute der enthaltenen Mikroobjekte zu ergänzen (z.B. das

Attribut Vermögen hinzufügen). Bei der Anonymisierung ist eine bestehende Mikrodatenbasis als Ausgangspunkt dagegen zwingend notwendig. Der Zugriff auf eine bestehende Mikrodatenbasis ist in der aktuellen Sprachdefinition noch nicht enthalten und für die nächste Realisierungsstufe geplant.

Die drei senkrechten Balken links in der Abbildung stellen die Datenbasen dar, die im Rahmen der oben beschriebenen Aktionen benötigt werden.

Die **Mikrodatenbasis** enthält eine Menge von Mikroobjekten, die verschiedenen Klassen angehören können (z.B. Personen und Haushalte). Sie kann analysiert sowie im Rahmen von Simulation, (Teil-) Generierung und Anonymisierung verändert werden.

Die **Parameterdatenbasis** versorgt die Simulation und Generierung/Anonymisierung mit Werten und Wahrscheinlichkeiten, z.B. jährlich geänderte Sterbewahrscheinlichkeiten in Abhängigkeit vom Alter, Geschlecht und Familienstand einer Person.

Die **Zeitreihendatenbasis** umfaßt für jede Größe und Periode nur genau einen Wert und kann wie die Parameterdatenbasis als Input einer Fortschreibung verwendet werden (z.B. erwartete Inflationsrate). Daneben kann sie auch in der Analyse und Simulation als Ergebnis erzeugt werden.

Die nachfolgende Tabelle zeigt, welche Eingangs- und Ausgangsdaten die Teilsprachen der MML verwenden bzw. erzeugen:

Teilsprache	**Input**				**Output**			
	Mikrodatenbasis	Parameterdatenbasis	Zeitreihendatenbasis	Textdatei	Mikrodatenbasis	Report	Zeitreihendatenbasis	Verteilungsdatenbasis
Test	x	-	-	-	-	x	-	-
Analyse	x	-	-	-	-	x	o	o
Simulation	x	o	o	-	x	x	o	o
Generierung	o	o	-	o	x	x	-	-

Legende:

x: wird immer verwendet bzw. erzeugt

o: wird optional verwendet bzw. erzeugt

-: wird nie verwendet bzw. erzeugt

3 Grundregeln

3.1 Schreibweise

MISTRAL ist wie PASCAL eine formatfreie Sprache. Das bedeutet, daß zwischen den Elementen eine beliebige Zahl von Trennzeichen (Leerzeichen, Tab und Zeilenvorschub) eingefügt werden kann. Sofern die Schreibweise eindeutig ist, können die Trennzeichen auch entfallen.

Wie in PASCAL wird auch in MISTRAL nicht zwischen Groß- und Kleinschreibung unterschieden. Selbstdefinierte Namen können deshalb ebenso wie Schlüsselwörter Groß- und Kleinbuchstaben in beliebiger Anordnung enthalten. So ist z.B. der Name "StellungZumHv" möglich. Dieser Wechsel verbessert bei zusammengesetzten Namen die Lesbarkeit und ist oft der Verwendung des Unterstrichs ("_") bei der Definition von Namen vorzuziehen.

In dieser Beschreibung werden MISTRAL-Schlüsselwörter normalerweise groß (und in Programmbeispielen z.T. zusätzlich fett) gedruckt. Dies dient jedoch nur der besseren Erkennung durch den Leser und ist nicht zwingend.

3.2 Namen

Innerhalb des Programms vergibt der Programmierer eine Vielzahl von Namen, z.B. für Typen, Ausprägungen, Variablen und Prozeduren. Für alle diese Namen gelten einheitlich folgende Regeln:

- Ein Name beginnt mit einem Buchstaben, auf den Buchstaben und Ziffern folgen können. Als Buchstaben gelten "A" bis "Z" und der Unterstrich ("_"). Da Groß- und Kleinschreibung nicht unterschieden werden, sind natürlich auch "a" bis "z" zulässig. Deutsche Umlaute und das "ß" sind dagegen nicht erlaubt.
- Schlüsselwörter und - anders als bei PASCAL - Standardbezeichner sind für die Verwendung von Namen nicht zulässig. Eine Liste der Schlüsselwörter und Standardbezeichner befindet sich im Anhang.
- Je nach Implementierung kann die Länge der Namen begrenzt sein. Ebenso ist es möglich, daß nur eine kürzere Zahl von Stellen signifikant ist. Die aktuelle Implementierung kennt hier jedoch keine Beschränkungen.

3.3 Kommentare

Kommentare sind in drei Formen möglich:

(* *Kommentar, auch über mehrere Zeilen* ***)**

{ *Kommentar, auch über mehrere Zeilen* }

*Kommentar, endet automatisch am Zeilenende*

Kommentare eines Typs können durch einen anderen Typ umschlossen werden. Deshalb bietet sich zur normalen Kurzkommentierung der letzte Typ an. Mit den ersten beiden Typen können dann größere Bereiche - z.B. zu Testzwecken - auskommentiert werden.

4 Datentypen

4.1 Allgemeines

Als Mikrosimulationssprache basiert die Datenstruktur von MISTRAL auf empirisch erhobenen Mikrodaten. Gegenüber den Standardtypen, wie sie in jeder Programmiersprache zu finden sind, gibt es insbesondere zwei deutlich Unterschiede:

Fehlen Angaben zu einzelnen Attributen eines Mikroobjekt oder wurden sie aufgrund von Inkonsistenzen gelöscht, so entstehen **Missing-Values**, d.h. fehlende Werte. Für jede Variable muß es also möglich sein, neben den normalen Werten auch diesen Zustand anzunehmen. In Übereinstimmung mit der Kodierung innerhalb des Sozioökonomischen Panels (SOEP) des Deutschen Instituts für Wirtschaftsforschung wurde dafür der Wert -1 vorgesehen. Missing-Values können grundsätzlich bei allen Datentypen (z.B. INTEGER, REAL sowie Aufzählungstypen) vorkommen. Sie werden vom System automatisch erkannt, wenn dieser Wert im normalen Wertebereich nicht enthalten ist. Dies gilt für auf den positiven Bereich beschränkte Zahlen, BOOLEAN-Werte sowie Aufzählungstypen. Innerhalb eines MISTRAL-Programms kann dieser Zustand über den Pseudowert **MISSING_VALUE** erzeugt werden.

Einige Variablen können neben den normalen Ausprägungen auch einen Wert "trifft nicht zu" ("does not apply") besitzen. Dies kann einerseits numerische Werte betreffen (z.B. die Ehedauer bei Unverheirateten), andererseits auch BOOLEAN- oder Aufzählungstypen (z.B. "hierarchische Stellung im Beruf " bei Nichterwerbspersonen). In diesen Fällen ist es möglich, solchen Variablen den Pseudowert **DOES_NOT_APPLY** zuzuweisen, der intern mit -2 kodiert wird. Im Gegensatz zu den Missing-Values, die bei jeder Variablen auftreten können, ist dies jedoch nur bei den Typen zulässig, bei denen es in der Typdefinition ausdrücklich vorgesehen ist. Hierfür stehen spezielle Varianten der Typen INTEGER, REAL und BOOLEAN zur Verfügung; bei den selbstdefinierten Aufzählungstypen ist dies in der Definition mit anzugeben.

4.2 Standard-Datentypen

4.2.1 INTEGER

Der INTEGER-Typ entspricht weitgehend dem in PASCAL verwendeten. Da für aggregierte Geldwerte einer Volkswirtschaft Zahlen im mehrstelligen Milliardenbereich benötigt werden, ist der Zahlenbereich in MISTRAL praktisch unbeschränkt.

Es werden folgende Spezialtypen unterschieden:

INTEGER: normaler Ganzzahltyp für positive und negative Zahlen

POSINTEGER: beschränkt auf positive Ganzzahlen

POSINTEGER_DNA: neben positiven Ganzzahlen ist auch DOES_NOT_APPLY möglich

Eine Zuweisung negativer Werte an die beiden positiven Typen führt zu einem Laufzeitfehler mit Abbruch der Programmausführung.

INTEGER-Zahlen werden in der üblichen Form angegeben, dürfen jedoch auch einen positiven Exponenten besitzen. Die Syntax sieht damit folgendermaßen aus:

["+"|"-"] *Ziffer* {*Ziffer*} ["E" ["+"] *Ziffer* {*Ziffer*}]

Beispiele:

12

-27

3E6

4.2.2 INDEX

Dieser Typ, der nur innerhalb von Parametertabellen möglich ist, kann ausschließlich ganzzahlige Werte ab einschließlich 1 annehmen.

4.2.3 REAL

Der Typ REAL entspricht der üblichen Form der Fließkommazahlen.

Auch hier werden drei Spezialtypen unterschieden:

REAL: normaler REAL-Typ für positive und negative Zahlen

POSREAL: beschränkt auf positive REAL-Zahlen

POSREAL_DNA: neben positiven REAL-Zahlen ist auch DOES_NOT_APPLY möglich

Eine Zuweisung negativer Werte an die beiden positiven Typen führt zu einem Laufzeitfehler mit Abbruch der Programmausführung.

Syntax:

["+"|"-"] *Ziffer* {*Ziffer*} "." *Ziffer* {*Ziffer*} ["E" ["+"|"-"] *Ziffer* {*Ziffer*}]

Beispiele:

1.2

-3.76E-6

4.2.4 PROB

Der Typ PROB stellt einen Untertyp von REAL dar und kann nur Werte zwischen 0.0 und 1.0 annehmen. Er dient dazu, Wahrscheinlichkeiten im Zusammenhang mit Fortschreibungsparametern zu definieren. Dieser Typ kann weder für die Deklaration von Variablen noch von Funktionen verwendet werden, sondern ist nur als Angabe für Parametertabellen möglich.

Werden für eine Parametertabelle PROB-Werte erwartet, so können auch REAL-Konstanten bzw. -Ausdrücke verwendet werden. Liegen diese Werte außerhalb des Bereichs, werden sie auf die nächste Grenze (also 0.0 oder 1.0) gesetzt.

Lediglich die Standardfunktion "RANDOM" liefert Werte dieses Typs.

4.2.5 BOOLEAN

Der Typ BOOLEAN kann die beiden Wahrheitswerte "**TRUE**" und "**FALSE**" annehmen.

Der Typ BOOLEAN3 entspricht im wesentlichen dem Typ BOOLEAN, kann jedoch auch die Ausprägung "DOES_NOT_APPLY" annehmen. Dies ist dann nützlich, wenn keiner der Werte "TRUE" oder "FALSE" sinnvoll ist, z.B. das Attribut "arbeitslos" bei Kleinkindern.

Wird ein BOOLEAN3-Wert in einer Bedingung oder einer logischen Operation (z.B. AND, OR) verwendet, so wird die Ausprägung "DOES_NOT_APPLY" wie "FALSE" behandelt.

BOOLEAN-Werte können problemlos einer Variablen des Typs BOOLEAN3 zugewiesen werden, jedoch nicht umgekehrt. Gleiches gilt für den Aufruf von Unterprogrammen, die formale Parameter dieser Typen enthalten. Die Umwandlung von BOOLEAN3 in BOOLEAN kann aber bei Bedarf leicht über folgenden Ausdruck durchgeführt werden:

boolean3-wert = TRUE

Die Funktion

BTOI (*boolean-wert*)

liefert 1, wenn das Argument TRUE ist, sonst 0.

4.2.6 STRING

Als String wird in einem MISTRAL-Programm eine Zeichenkette interpretiert, die in einfachen Anführungszeichen eingeschlossen ist. Dabei kann sich der String grundsätzlich auch über mehrere Zeilen erstrecken. Die Länge von Strings ist praktisch unbegrenzt; eine Längenangabe (z.B. string[255]) ist nicht notwendig. Wird innerhalb eines Strings ein einfaches Anführungszeichen benötigt, so ist dieses Zeichen zweimal unmittelbar hintereinander zu schreiben.

Beispiele:

'Dies ist ein String!'

'Wie geht"s?' (gelesen als: "Wie geht's?")

Da innerhalb einer Mikrosimulation keine interaktiven Eingaben oder Zugriffe auf Textdateien erfolgen, sind nur die wichtigsten Funktionen zur String-Verarbeitung vorhanden:

Mit der Funktion

CONCAT (*string1*, *string2*, ...)

können String-Konstanten bzw. -Ausdrücke verkettet werden.

Die Funktion

COPYSTR (*string*, *start* [, *stop*])

liefert einen Teilstring von *string* beginnend mit der Position *start* bis zur Position *stop*. Fehlt die Angabe *stop*, wird der Rest bis zum Ende des Strings geliefert. Ist *start* größer als die Länge des Strings, wird ein leerer String zurückgegeben. Das erste Zeichen des Strings hat die Position 1.

Die Funktion

LENGTH (*string-ausdruck*)

liefert einen INTEGER-Wert mit der Anzahl der Zeichen des Strings zurück.

Mit der Funktion

STR (*ausdruck* [*format-angabe*])

wird ein String zurückgegeben, der dem Wert des Ausdrucks entspricht. Wie beim WRITE-Befehl (s. dort) können nicht nur numerische Ausdrücke verwendet werden, sondern auch

solche mit BOOLEAN-, ENUM-, Objekt- und SET-Typen. Auch die optionale Formatangabe entspricht der bei WRITE.

Für Dokumentationszwecke gibt es die Funktionen **DATE** und **TIME**, die einen String mit dem aktuellen Datum bzw. der aktuellen Uhrzeit zurückgeben. DATE erzeugt dabei ein englisches Format nach dem Muster 'mmm dd, yyyy' (z.B. 'Dec 12, 1995'), das TIME-Format lautet 'hh:mm:ss' (z.B. '17:56:04'). Die Funktion **PERIOD** liefert einen String mit der aktuellen Simulationsperiode.

4.2.7 VECTOR und RANGE

Die VECTOR-Typen dienen dazu, eine Liste zusammengehöriger Zahlen mit einem Zugriff aus der Parameterdatenbasis zu lesen. Dies ist vor allem für Koeffizienten innerhalb von Gleichungen von Bedeutung. Vektoren können nur innerhalb der Parameterdatenbasis definiert werden. Auf sie wird in einem Simulationsprogramm mit der Funktion PARAM_VALUE zugegriffen.

Vektoren ähneln den Arrays von PASCAL, da sie einen Zugriff auf einzelne Komponenten mit

***variable*[*index*]**

zulassen. Der Index muß immer ein positiver (> 0) INTEGER-Wert sein. Die Größe des Vektors ist in der Parameterdatenbasis festgelegt und kann schwanken. Dies entspricht dem Gebrauch von Strings in PASCAL. Ebenso wie dort ist ein Zugriff auf nicht vorhandene Komponenten unzulässig und führt zu einem Laufzeitfehler.

Eine Zuweisung an eine Variable ist nur als ganzes möglich, z.B.:

```
A := PARAM_VALUE (Test,maennlich,1);
```

Eine Zuweisung an einzelne Komponenten eines Vektors ist nicht möglich.

Es werden zwei VECTOR-Typen unterschieden:

IVECTOR und **RVECTOR**

Dabei handelt es sich um Vektoren, deren Komponenten INTEGER- bzw. REAL-Werte sind. Sie können wie die übrigen Typen verwendet werden, sind jedoch nicht als Inputs einer Parameterdefinition zulässig.

Eine Untervariante von VECTOR ist RANGE. Dieser Typ besteht aus jeweils genau zwei Komponenten, die die untere und obere Grenze eines Bereichs darstellen. Verwendet werden diese Objekte vor allem im Zusammenhang mit der Funktion PARAM_RANGEVALUE. Dort wird zunächst innerhalb der Parameterdatenbasis nach vorgegebenen Wahrscheinlichkeiten ein RANGE bestimmt. Innerhalb dieses Bereichs wird dann von der Funktion PARAM_RANGEVALUE gleichverteilt ein einzelner Zufallswert ermittelt.

Analog zu VECTOR werden auch hier

IRANGE und **RRANGE**

unterschieden.

4.2.8 DISTR

Der Typ DISTR unterscheidet sich grundlegend von üblichen Daten-Typen. Er nimmt die Ergebnisse von 1- und 2-dimensionalen Verteilungsanalysen auf. Weitere Informationen hierzu befinden sich im Abschnitt 7.4 zu den MQL-Ausdrücken.

4.2.9 Periode

Innerhalb der Fortschreibungsparameter und der Generierung neuer Mikrodatenbasen muß in der Regel eine bestimmte Periode definiert werden. Dies geschieht mit folgender Syntax:

Integer-Zahl [["S"|"Q"|"M"] *Integer-Zahl*]

Neben vollen Jahren können somit auch Halbjahre ("S" = Semi-Year), Quartale ("Q") und Monate ("M") angegeben werden.

Zweistellige Jahreszahlen im Bereich 0..29 werden als 2000..2029 interpretiert, solche im Bereich 30..99 als 1930..1999.

Auf die Daten der aktuellen Simulationsperiode kann mit folgenden Funktionen zugegriffen werden, die jeweils INTEGER-Zahlen liefern:

YEAR: vierstellige Jahreszahl

SEMIYEAR: Halbjahr (1..2)

QUARTER: Quartal (1..4)

MONTH: Monat (1..12)

4.3 Enumerations-Typen

Innerhalb der Mikrosimulation werden zu einem großen Teil auch qualitative Daten verarbeitet. Diese entsprechen weitgehend den Aufzählungstypen in PASCAL. Es gibt jedoch wichtige Unterschiede:

- Es werden nominal und ordinal skalierte Variablen unterschieden. Im ersten Fall ist keine Ordnung innerhalb der Werte (Merkmalsausprägungen) definiert, so daß nur Vergleiche auf gleich oder ungleich vorgenommen werden können. Bei ordinalen Variablen werden die Ausprägungen in der bei der Definition festgelegten Reihenfolge aufsteigend angeordnet. Somit sind Vergleiche wie "*wert1* < *wert2*" möglich. Ordinale Variablen sind eher die Ausnahme. Ein mögliches Beispiel wäre der Bildungsabschluß (z.B. "Hauptschule", "Realschule", "Abitur", "FH-Abschluß", "Universitätsabschluß").
- Für jede Ausprägungen sind optional beliebig viele Aliase möglich, die als Abkürzungen (z.B. "HV" statt "Haushaltsvorstand") oder alternative Bezeichnung (z.B. "Gatte", "Ehegatte", "Ehepartner") verwendet werden können. Sie werden zudem bei Ausgaben des Systems als Abkürzungen verwendet, wenn der Platz für die Standardbezeichnung nicht ausreicht.
- Optional können die Variablen zusätzlich den Wert "DOES_NOT_APPLY" annehmen, der als "trifft nicht zu" o.ä. interpretiert werden kann. Dies ist immer dann sinnvoll, wenn z.B. durch den Wert eines anderen Attributs ein Merkmal nicht relevant ist (z.B. der Beruf bei Nichterwerbspersonen). In diesem Fall ist die Ausprägung "DOES_NOT_APPLY" als er-

ste anzugeben (ohne Aliase). Bei Ordinalskalierung ist somit "DOES_NOT_APPLY" immer die kleinste Ausprägung.

Die Definition wird innerhalb der TYPE-Vereinbarung vorgenommen und hat beispielsweise folgendes Aussehen:

```
Geschlechter =
    maennlich (m),
    weiblich (w);
```

4.4 Objekte

Die wichtigsten Daten innerhalb einer Mikrosimulation sind die Mikrodaten. Dabei handelt es sich um eine Menge von Objekten (z.B. Personen und Haushalte) mit bestimmten Eigenschaften (z.B. Alter, Geschlecht, Familienstand). In MISTRAL sind dafür Objekttypen vorgesehen, die zwischen den Records im Standard-PASCAL und dem Datentyp "Objects" in den objektorientierten Erweiterungen (z.B. Turbo-PASCAL ab 5.5) einzuordnen sind. Den Objekttypen in MISTRAL werden zwar Methoden zugeordnet, diese werden jedoch - anders als bei den PASCAL-Objects - nicht innerhalb der Datenvereinbarung angegeben. Der Hauptgrund dafür liegt darin, daß die Datendefinition - z.B. innerhalb einer Mikrodatenbasis - auch ohne ein Simulationsmodell verwendet wird.

Die Objekte werden innerhalb der CLASS-Sektion definiert. Ein einfaches Beispiel dazu ist folgendes:

```
Person =
    Haushalt (HH): Haushalt,
    Alter: INTEGER,
    Geschlecht (Gesch): Geschlechter,
  TEMPORARY
    schonSimuliert: Boolean;
```

Wie schon beim Aufzählungstyp fällt auch hier im Unterschied zu PASCAL auf, daß Attribute des Objekts Alias-Bezeichnungen besitzen können. Als Typen der Attribute sind neben den oben genannten Elementartypen sowie dem Aufzählungstyp (hier der zuvor definierte Typ "Geschlechter") auch Zeiger auf andere Objekte (hier auf einen Haushalt) sowie auf Objektmengen (siehe unten) möglich.

Eine Besonderheit stellen die Attribute dar, die nach dem Schlüsselwort TEMPORARY aufgeführt sind. Sie werden zu Beginn jeder Simulation (für jede Periode neu) auf den Default-Wert gesetzt und stehen innerhalb der Simulation wie normale Attribute zur Verfügung. Nach der Simulation werden diese Attribute wieder automatisch gelöscht. Es handelt sich dabei meist um Hilfsgrößen, die zur Steuerung der Simulation benötigt werden und nicht Teil der Mikrodatenbasis sind.

Es ist wichtig festzuhalten, daß Objektvariablen und -attribute grundsätzlich nur Zeiger auf Objekte darstellen. Auf dasselbe Objekt können beliebig viele Zeiger gerichtet sein (z.B. Mutterzeiger, Gattenzeiger usw.). Wird von einem dieser Zeiger aus schreibend auf das Objekt zugegriffen, so ist diese Veränderung überall wirksam (da es sich immer nur um ein und dasselbe Objekt handelt). MISTRAL-Objekte sind also grundsätzlich mit Zeigervariablen in PASCAL vergleichbar, auch wenn dies hier nicht durch eine besondere Schreibweise deutlich

gemacht wird (anders als in PASCAL gibt es hier keine statische Verwaltung eines solchen Objekts).

Der lesende und schreibende Zugriff auf Attribute wird wie in PASCAL auch mit folgender Schreibweise realisiert:

objekt.attribut

Ist das Attribut wiederum ein Zeiger auf ein anderes Objekt, kann der Zugriff entsprechend fortgeführt werden:

Person1.Gatte.Alter

Soll ein Zeiger auf kein Objekt zeigen, so ist ihm der Wert **NIL** zuzuweisen:

zeiger := **NIL**

Wie Zeigertypen in PASCAL müssen Objekte explizit mit

variable := **NEW** (*objekt-typ*)

erzeugt werden. Gelöscht werden sie durch den Befehl:

DELETE (*objekt-zeiger*)

Es ist wichtig zu beachten, daß damit nur das Objekt aus der zentralen Mikrodatenbasis gelöscht wird. Existieren von anderen Objekten Zeiger auf das Objekt (z.B. Ehepartnerzeiger), so werden diese nicht automatisch gelöscht, sondern zeigen weiterhin auf das gelöschte Objekt. Sie sind durch explizite Zuweisungen auf NIL zu setzen. Alternativ kann mit dem Befehl

DELETEWITHPOINTER (*objekt-zeiger*)

das Objekt gelöscht werden, wobei gleichzeitig alle Pointer auf dieses Objekt auf NIL gesetzt werden. Ist das Objekt in einem SET enthalten, wird es daraus entfernt. Dem Komfort des automatischen Löschens steht eine etwas höhere Laufzeit gegenüber, so daß es sich z.B. bei Zeigern, die ausschließlich innerhalb eines Haushaltes verweisen, anbietet, diese selbst zu entfernen.

Ein Zeiger auf ein bestimmtes Objekt kann durch Angabe seiner Klasse und seiner innerhalb der Klasse einmaligen Schlüssel-Nr. (KEY) erzeugt werden:

GETOBJECT (*klassen-name*, *objekt-nr*)

Die Funktion

COPY (*objekt*)

erzeugt ein neues Objekt derselben Klasse wie das Argument und weist ihm für alle Attribute die gleichen Werte zu; lediglich die Zeiger werden auf NIL bzw. EMPTYSET gesetzt.

Vor allem zu Debugging-Zwecken kann der Status eines Objektes mit allen Zeigern und Attributen mit der Prozedur

WRITEOBJECTINFO (*objekt*)

in den Report ausgegeben werden.

4.5 Objektmengen

Innerhalb der Mikrodaten bzw. Mikrosimulation wird oft mit Mengen von Objekten operiert, z.B. der Menge der Personen eines Haushaltes. Hierfür stellt MISTRAL den Mengentyp "**SET OF**" zur Verfügung, der eine Menge von Objekten des gleichen Objekttyps beinhaltet. Die Mengen sind ausschließlich für Objekte vorgesehen und nicht wie z.B. in PASCAL für Aufzählungstypen o.ä.

Eine leere Menge ist durch den Wert **EMPTYSET** gekennzeichnet, der auch direkt einer SET-Variablen zugewiesen werden kann.

Mit der Prozedur

ADD (*set*, *objekt*)

können Objekte zu einer Menge hinzugefügt werden. Ist das Objekt bereits in der Mengen enthalten, hat der Aufruf keine Wirkung.

Einzelne Objekte können mit der Prozedur

REMOVE (*set*, *objekt*)

entfernt werden. Auch hier hat es keine Wirkungen, wenn ein Objekt entfernt werden soll, das nicht in der Menge enthalten ist.

Mit der Funktion

FIRST (*set*)

wird ein Zeiger auf das Objekt mit dem kleinsten Schlüssel innerhalb der Menge zurückgegeben.

Mit

SAMPLE (*set*, *zahl*)

wird eine zufällige Teilmenge (Stichprobe) des Sets erzeugt. Handelt es sich bei *zahl* um einen INTEGER-Wert, wird eine Menge mit genau dieser Anzahl von Elementen erzeugt (maximal bis zur ursprünglichen Anzahl). Handelt es sich um einen REAL-Wert zwischen 0.0 und 1.0, wird eine Teilmenge mit einem entsprechenden Anteil gezogen.

Mit

RANDOMOBJECT (*set*)

wird ein zufälliges Objekt einer Menge ausgewählt.

Mit

GETCLASS (*klassen-name*)

wird ein SET zurückgegeben, das alle Objekte der betreffenden Klasse aus der Mikrodatenbasis enthält.

Die Funktion

SIZEOF (*set*)

gibt die Anzahl der Elemente des SETs zurück.

Eine Menge kann auch direkt einer Mengenvariablen zugewiesen werden, indem auf der rechten Seite des Zuweisungszeichens ein (MQL-) Ausdruck steht, der eine Menge repräsentiert.

4.6 Default-Werte und Initialisierung von Variablen

Grundsätzlich müssen - wie auch in PASCAL - Variablen vor ihrer ersten Verwendung initialisiert werden. Dennoch erhalten alle Variablen beim Starten des Programms einen definierten, zulässigen Wert. Dabei sind folgende Default-Werte für die verschiedenen Variablen-Typen vorgesehen:

INTEGER	0
POSINTEGER	0
POSINTEGER_DNA	DOES_NOT_APPLY (-2)
REAL	0.0
POSREAL	0.0
POSREAL_DNA	DOES_NOT_APPLY (-2.0)
BOOLEAN	FALSE
BOOLEAN3	DOES_NOT_APPLY
Enumerations-Typ	1. Ausprägung bzw. DOES_NOT_APPLY, wenn zulässig
Objekte	NIL
SET	EMPTYSET (leere Menge)
STRING	" (leerer String)

Die Default-Werte werden einmalig zum Start des Programms für alle Variablen - auch die lokalen - zugewiesen, um eine zeitaufwendige Typprüfung zur Laufzeit zu umgehen. Dies bedeutet jedoch, daß nicht bei jedem Aufruf eines Unterprogramms erneut eine solche Initialisierung durchgeführt wird. Bei lokalen Variablen stehen die Default-Werte deshalb nur beim ersten Aufruf zur Verfügung, so daß sie für den Programmierer nicht verwendbar sind. Einen Nutzen innerhalb des Programms bieten die Default-Werte jedoch bei neu erzeugten Objekten. Diese erhalten bei jedem NEW-Befehl einen Satz von Attributen, die mit den Default-Werten initialisiert wurden. Damit ist es z.B. bei Neugeborenen überflüssig, eine Vielzahl von ökonomischen und Erwerbsbeteiligungs-Variablen zu setzen. Beim Generieren von Mikrodatenbasen ist diese Eigenschaft besonders wichtig.

5 Programmaufbau

Ein vollständiges MML-Programm besteht aus verschiedenen Teilen, die z.T. optional sind, wenn verwendet jedoch in dieser Reihenfolge vorkommen müssen. In den nachfolgenden Abschnitten wird für die vier Untersprachen von MISTRAL die jeweilige Programmstruktur mit den zulässigen Teilen im Überblick dargestellt.

5.1 Test

Ein Test-Programm hat folgenden Aufbau:

TEST *name*;

TYPE ...;	(optional)
CLASS ...;	
CONST ...;	(optional)
VAR ...	(optional)

optionale Unterprogramm-Definitionen (nur Funktionen und Prozeduren)

BEGIN
Haupt-Programm
END.

5.2 Analyse

Beim Analyse-Programm ist die INTERFACE-Section vorhanden, erlaubt jedoch nur die Unterabschnitte OUTPUT und DISTRIBUTION.

ANALYSIS *name*;

TYPE ...;	(optional)
CLASS ...;	
INTERFACE ... ;	(optional)
CONST ...;	(optional)
VAR ...	(optional)

optionale Unterprogramm-Definitionen (nur Funktionen und Prozeduren)

BEGIN
Haupt-Programm
END.

5.3 Simulation

SIMULATION *name*;

TYPE ...;	(optional)
CLASS ...;	
INTERFACE ... ;	(optional)
PARAMETER ... ;	(optional)
CONST ...;	(optional)
VAR ...	(optional)

optionale Unterprogramm-Definitionen

BEGIN
Haupt-Programm
END.

5.4 Generierung

GENERATION *name*;

TYPE ...;	(optional)
CLASS ...;	
FILE ... ;	(optional)
PARAMETER ... ;	(optional)
CONST ...;	(optional)
VAR ...	(optional)

optionale Unterprogramm-Definitionen

BEGIN
Haupt-Programm
END.

5.5 Include-Dateien

Bei größeren Projekten - insbesondere, wenn mehrere Personen zusammenarbeiten - ist es oft sinnvoll, ein Gesamtprogramm auf mehrere separate Dateien aufzuteilen. Damit können z.B. getrennte Module von verschiedenen Personen unabhängig voneinander entwickelt und verändert werden, ohne daß jeweils aufwendige Kopieraktionen zum Abgleich notwendig sind. Um die veränderten Programmteile einzubinden, wird lediglich mittels einer INCLUDE-Anweisung auf die entsprechende Datei verzeigert. Änderungen in der eingebundenen Datei werden beim nächsten Compilieren automatisch wirksam, ohne daß in der Hauptdatei etwas verändert werden muß.

Konkret kann in allen MISTRAL-Programmen (inkl. Parameterdatenbasis und Zeitreihen) die Anweisung

#INCLUDE *dateiname*

verwendet werden, um an dieser Stelle den Text der angegebenen Datei so einzubinden, als stünde er anstelle dieser Zeile im Haupttext. Realisiert wird dies von einem Präprozessor, der zunächst alle diese Zeilen durch den Inhalt der jeweiligen Datei ersetzt.

Die Anweisung muß (ohne Leerzeichen) am Beginn einer Zeile stehen (auch zwischen "#" und "INCLUDE" darf kein Leerzeichen stehen).

Der Dateiname kann relativ zur Position des jeweiligen Dokuments angegeben werden (z.B. "teil.mml" oder "sub/teil.mml") oder absolut (z.B. "c:\umdbs\projekt\teil.mml"; Laufwerksangabe notwendig). Eine Verwendung des Kürzels "..\", um zum übergeordneten Verzeichnis zu verweisen, ist möglich.

Hinter dem Dateinamen kann bis zum Zeilenende beliebiger Text folgen, der als Kommentar interpretiert wird.

INCLUDE kann auch innerhalb einer Datei verwendet werden, die selbst über INCLUDE eingebunden wurde. Damit läßt sich eine hierarchische Struktur von Include-Dateien erzeugen, die sinnvollerweise innerhalb des Dateisystems eine Entsprechung haben sollte.

Im praktischen Gebrauch innerhalb des UMDBS wird das Hauptdokument normal geladen, editiert und initialisiert. Während der Initialisierung werden die Teildokumente, die normalerweise im selben Verzeichnis stehen sollten, automatisch eingebunden. Tritt in einem Teildokument ein Compiler-Fehler auf, wird diese Datei automatisch in einem zusätzlichen Editor geöffnet und die Fehlermeldung und -position angezeigt. Der Fehler kann damit direkt behoben und die Datei gespeichert werden. Anschließend wird das Hauptdokument erneut initialisiert. Zum normalen Editieren der Teildokumente kann ein entsprechender Editor unter dem Menü "Datei | Werkzeuge | Editoren" aufgerufen werden.

Bei Fehlermeldungen des Laufzeitsystems wird bei der Verwendung von INCLUDE-Dateien der Name der Datei angezeigt, in der sich die betroffenen Zeile befindet, sowie die Zeilennummer innerhalb dieses Dokuments.

6 Vereinbarungsteil

6.1 TYPE

Als Enumerations- (ENUM-) Typen können nominale und ordinale Aufzählungstypen definiert werden. Wird der Typ durch das Schlüsselwort "**ORDINAL**" als Ordinaltyp definiert, so sind in logischen Vergleichen auch ">", "<" usw. möglich, sonst nur "=" und "<>". Zu jeder Ausprägung können optional beliebig viele Aliase definiert werden (vor allem Abkürzungen), die bei Abfragen und Zuweisungen anstelle des Grundnamens verwendet werden können. Bei Ausgaben, bei denen der Platz nicht ausreicht, wird vom System selbständig eine verfügbare kürzere Schreibweise verwendet. Als erste Ausprägung kann optional jeweils die Ausprägung "**DOES_NOT_APPLY**" (z.B. "trifft nicht zu") definiert werden; Aliase dafür sind nicht zulässig.

Beispiele:

```
TYPE
   Familienstaende =
      ledig (l, led),
      verheiratet (verh),
      geschieden (gesch),
      verwitwet (verw);

   StellungenZumHv =
      ORDINAL
      Haushaltsvorstand (HV),
      Ehepartner (Gatte),
      Lebenspartner (LP),
      Kind (K),
      Verwandter (Verw, V),
      Sonstiger (Sonst, S);

   Beruf =
      DOES_NOT_APPLY,
      Selbstaendiger (Selb),
      Arbeiter (Arb),
      Angestellter (Ang),
      Beamter (Beam),
      Mithelfender (Mith);
```

6.2 CLASS

Die CLASS-Definition eines Objekts ist der eines Records in PASCAL vergleichbar. Zu jedem Objekttyp werden die verfügbaren Attribute und deren Typen angegeben. Als Typen sind dabei alle Datentypen möglich. Eine Besonderheit stellen die Aliase dar, von denen beliebig viele für jeden Attributnamen definiert werden können.

```
CLASS
   Haushalt =
      Personen (P): SET OF Person;

   Person =
      Haushalt (HH): Haushalt,
      Gatte: Person,
      Alter: INTEGER,
      Geschlecht (Geschl): Geschlechter,
      Familienstand (Famst): Familienstaende,
    TEMPORARY
      schonSimuliert: Boolean;
```

Alle Objekte besitzen zudem zwei implizite Attribute, die für Simulationszwecke benötigt werden. Die Definition dieser Attribute kann vom Benutzer nicht geändert werden. Die Attribute können jedoch sowohl bei Abfragen als auch bei Zuweisungen normal verwendet werden.

Key	INTEGER	Objekt-Nr.; für jede Objektart (Klasse) getrennt vergeben
Factor	REAL	Hochrechnungsfaktor

Die CLASS-Definition entspricht grundsätzlich einer Klassendefinition in objektorientierten Sprachen.

Die so definierten Klassen müssen zum Inhalt der verwendeten Mikrodatenbasis passen. Das bedeutet, daß Namen (ohne den in Klammern möglichen Alias), Typ und Reihenfolge gleich sein müssen. Innerhalb des MISTRAL-Programms können jedoch zusätzliche Attribute nach dem Schlüsselwort TEMPORARY definiert werden, die nicht in der Mikrodatenbasis enthalten sind und auch anschließend nicht dorthin abgespeichert werden.

6.3 INTERFACE

Die INTERFACE-Vereinbarung definiert die Schnittstelle des MML-Programms mit der Systemumgebung und besteht aus drei Teilen:

Der **INPUT-Teil** übernimmt vom restlichen System Variablen vom Typ REAL. Es handelt sich dabei um Makrowerte, die entweder von einer Zeitreihe oder einem anderen Simulationsmodell (z.B. einem Makrosimulator) stammen. INPUT-Variablen darf innerhalb des Programms kein Wert zugewiesen werden; sie werden also wie Konstanten behandelt.

Der **OUTPUT-Teil** dient dazu, REAL-Werte an die Systemumgebung (z.B. für die anschließende Analyse) weiterzugeben. Den OUTPUT-Variablen werden dazu ebenso wie REAL-Variablen an beliebiger Stelle innerhalb des Programmablaufs Werte zugewiesen.

Der **DISTRIBUTION-Teil** definiert Variablen des Typs DISTR, die nach dem Programmablauf der Systemumgebung zu Analysezwecken zur Verfügung stehen. Ihre Werte werden über einen MQL-Ausdruck definiert.

Eine vollständige INTERFACE-Definition kann z.B. so aussehen:

```
INTERFACE
  INPUT
    Y, L, I;
  OUTPUT
    C, N;
  DISTRIBUTION
    Alter, GesamtEinkommen;
```

6.4 FILE

Die FILE-Vereinbarung ist nur innerhalb von Generierungsprogrammen zulässig. Dort werden logische Namen für Textdateien definiert und optional bereits mit dem Namen (inkl. absoluten oder relativen Pfad) der Datei innerhalb des Dateisystems versehen. Fehlt der Name, so wird er interaktiv vom Simulationssystem vor dem Start des Programms abgefragt. Da diese Möglichkeit flexibler ist und die Übertragung auf andere Rechner erleichtert, ist ihr der Vorzug zu geben.

Es ist zu berücksichtigen, daß sich relative Pfadangaben grundsätzlich auf das Verzeichnis des obersten Dokuments beziehen, auch wenn sie in einer Include-Datei vorkommen.

Die maximale Anzahl von Dateien ist in der aktuellen Version auf 15 beschränkt.

Hier ein Beispiel mit beiden Varianten:

```
FILE
  file1 ('c:\mikrosim\daten\hh_datei.txt');
  file2 ('daten\p_datei.txt');
  file3;
```

6.5 PARAMETER

Die PARAMETER-Vereinbarung definiert die Schnittstelle zur Parameterdatenbasis. Die Definition der einzelnen Parametertabellen entspricht der eines Funktionskopfes, bestehend aus Namen, formalen Parametern (jeweils Name und Typ) sowie dem Typ des Rückgabewertes.

Für die Verbindung mit der Parameterdatenbasis sind die Namen der formalen Parameter entscheidend, die Reihenfolge kann an beiden Stellen unterschiedlich sein.

Nachfolgend ein Beispiel einer PARAMETER-Sektion:

```
PARAMETER
  Geburt
    (Alter: INTEGER;
      Familienstand: Familienstaende;
      erwerbstaetig: BOOLEAN): PROB;
  KinderZahl
    (r: PROB): INTEGER;
```

6.6 CONST

Es sind Konstanten vom Typ REAL, INTEGER, BOOLEAN und Ausprägungen eines ENUM-Typs möglich.

Ein Beispiel:

```
CONST
    const1 = 1.34;
    const2 = 65;
    const3 = TRUE;
    const4 = ledig;
```

6.7 VAR

Jede Variable innerhalb eines MISTRAL-Programms muß vor ihrer Verwendung definiert werden. Dies geschieht in der in PASCAL üblichen Form:

```
VAR
    n, i: INTEGER;
    M1: SET OF Person;
    P: Person;
```

Auf der rechten Seite sind nur Standardtypen oder zuvor definierte Aufzählungs- oder Objekttypen bzw. Mengen von letzteren möglich. Anders als bei PASCAL ist keine direkte Typ-Definition zulässig.

Variablendefinitionen können innerhalb des Vereinbarungsteils des Gesamtprogramms stehen. In diesem Fall werden die Variablen als global definiert und sind überall im Programm verfügbar. Die Regeln des modernen Software-Engineerings lassen jedoch einen sehr sparsamen Umgang mit dieser Art von Variablen ratsam erscheinen.

Zusätzlich können lokale Variablen innerhalb von Unterprogrammen (z.B. Modulen, Prozeduren usw.) definiert werden und stehen nur dort zur Verfügung. Mit Verlassen des jeweiligen Unterprogramms geht der aktuelle Wert der Variablen verloren und kann beim nächsten Aufruf nicht mehr verwendet werden.

7 Operatoren und Ausdrücke

7.1 Arithmetische Operatoren und Ausdrücke

Es sind folgende arithmetische Operationen für Operanden der Typen INTEGER und REAL definiert:

-x	Vorzeichenumkehr
+x	Identität
x + y	Addition
x - y	Subtraktion
*x * y*	Multiplikation
x / y	Division; Ergebnis immer REAL
x DIV *y*	Ganzzahl-Division (Operanden und Ergebnis INTEGER)
x MOD *y*	Modulo-Funktion (Divisionsrest) (Operanden und Ergebnis INTEGER)

Daneben sind eine ganze Reihe von arithmetischen Standardfunktionen definiert:

ABS (*x*)	Betrag
ROUND (*x*)	normale Rundung
TRUNC (*x*)	Ganzzahlanteil (z.B. TRUNC(-1.5) = -1)
INT (*x*)	Ganzzahlanteil als REAL-Wert
FRAC (*x*)	gebrochener Teil eines REAL-Wertes (= *x* - INT(*x*); -> FRAC(-2.5) = -0.5)
SQR (*x*)	Quadrat
SQRT (*x*)	Quadratwurzel (Ergebnistyp immer REAL)
MAX (*x*, *y*)	Maximum
MIN (*x*, *y*)	Minimum
SIN (*x*)	Sinus-Funktion
COS (*x*)	Cosinus-Funktion
TAN (*x*)	Tangens-Funktion
LOG (*x*)	Logarithmus zur Basis 10
LN (*x*)	natürlicher Logarithmus
EXP (*x*)	Exponential-Funktion zur Basis e (e^x)

Für die stochastische Simulation sind Zufallszahlen von Bedeutung (siehe dazu auch Abschnitt 12.2). In MISTRAL stehen folgende Funktion als Zufallsgeneratoren zur Verfügung:

RANDOM	gleichverteilte Zufallszahl im Intervall (0.0; 1.0)
RANDOMUNIFORM (*a*, *b*)	gleichverteilte Zufallszahl; sind *a* und *b* beide vom Typ INTEGER, ist auch das Ergebnis ein INTEGER-Wert aus dem Intervall [*a*; *b*], sonst ein REAL-Wert aus (*a*; *b*)
RANDOMNORM (*my*, *s*)	normalverteilte Zufallszahl mit dem Erwartungswert *my* und der Standardabweichung *s*
RANDOMEXP (*x*)	exponentialverteilte Zufallszahl mit dem Erwartungswert 1/*x*

7.2 Logische Ausdrücke

Logische Ausdrücke liefern Boolean-Werte als Ergebnis zurück. Diese können auch als Konstanten direkt angegeben werden:

TRUE
FALSE

Mögliche Vergleichsoperatoren sind:

a = *b*
a <> *b*
a < *b*
a <= *b*
a > *b*
a >= *b*

Für INTEGER- und REAL-Zahlen stehen alle davon zur Verfügung. Bei Aufzählungstypen (ENUM) gilt dies nur für ordinale Typen. Nominale Typen lassen nur die Vergleiche "=" und "<>" zu. Gleiches gilt für Objekte und Mengen (SETs). Zwei Mengen gelten dann als gleich, wenn sie genau dieselben Objekte beinhalten.

Folgender Ausdruck liefert TRUE, wenn das Objekt in der Menge enthalten ist:

objekt **IN** *objekt-menge*

Ausdrücke können durch logische Operationen kombiniert werden:

NOT *a*
a **AND** *b*
a **OR** *b*

Die AND- und OR-Verknüpfung werden nur soweit von links nach rechts ausgeführt, bis das Ergebnis feststeht. Ist also der Wert von *a* bei der AND-Verknüpfung FALSE bzw. bei der OR-Verknüpfung TRUE, so wird *b* nicht berechnet. Dies dient der Beschleunigung der Berechnung. Zusätzlich kann diese Eigenschaft dazu genutzt werden, mit der Bedingung *a* sicherzustellen, daß *b* einen gültigen Wert besitzt. Beispiel:

(Gatte <> NIL) AND (Gatte.Alter >= 65)

Hier wird nur dann auf das Attribut "Alter" zugegriffen, wenn das Objekt "Gatte" existiert.

7.3 Mengen-Ausdrücke

Innerhalb von MISTRAL werden Mengen von Objekten in von SETs verwaltet.

Ein SET mit allen Objekten einer Klasse erhält man mit der Funktion

GETCLASS (*klassen-name*)

Ein leeres SET wird durch die Zuweisung

variable := **EMPTYSET**

erzeugt.

Zum Testen, ob ein bestimmtes Objekt in einer Menge enthalten ist, dient der Ausdruck

objekt **IN** *set*,

der einen BOOLEAN-Wert zurückliefert.

Mit der Funktion

SIZEOF (*set*)

wird die Anzahl der Elemente einer Menge als INTEGER-Wert geliefert.

Zwei Mengen können durch folgende Operationen verknüft werden:

set1 + set2	Vereinigungsmenge
*set1 * set2*	Schnittmenge
set1 - set2	Differenzmenge

Alle drei Ausdrücke liefern einen Wert, der z.B. einer Variablen zugewiesen werden kann. Die Ursprungsmengen werden dabei nicht verändert.

Für alle angegebenen Operationen gilt, daß die Mengen und Objekte immer zur selben Objektklasse gehören müssen.

7.4 MQL-Ausdrücke

Mit MQL (Mikro Query Language) wird ein spezieller Teil der Sprache MISTRAL bezeichnet, der sich deutlich von üblichen PASCAL-Anweisungen unterscheidet und eher Datenbank-Abfragesprachen wie SQL zuzuordnen wäre.

Ein MQL-Ausdruck besitzt folgende Grundstruktur:

set-ausdruck | *mql-befehl*

Da einige MQL-Befehle wiederum ein SET als Ergebnis liefern, können die Befehle verkettet werden:

set-ausdruck | *mql-befehl* | *mql-Befehl* | *mql-befehl*

Das Pipe-Symbol ("|") wird wie in UNIX oder DOS dazu verwendet, das Ergebnis des vorangegangenen Ausdrucks als Eingangswert für den folgenden zu verwenden. Auf diese Art werden Objektmengen von einem Befehl zum nächsten weitergereicht und dabei verändert. Hierfür stehen zwei MQL-Befehle zur Verfügung: SELECT und COLLECT.

Sollen aus einer Menge nur die Objekte verwendet werden, die eine bestimmte Bedingung erfüllen, so wird der **SELECT**-Befehl verwendet. Beispiel:

GETCLASS (Person) | SELECT (SELF.Geschlecht = maennlich)

Dieser Ausdruck liefert ein "SET OF Person" zurück, mit allen Männern, die in der Mikrodatenbasis enthalten sind. Die Pseudovariablen "SELF" bezeichnet nacheinander alle Objekte der Ursprungsmengen, für die diese Bedingung ausgewertet wird. Der Ausdruck, der zu SELECT gehört, muß vom Typ BOOLEAN sein und kann beliebig verknüpfte Teilausdrücke (mit AND, OR und NOT) enthalten.

Eine Besonderheit ist die Tatsache, daß Laufzeitfehler innerhalb des Booleschen Ausdrucks im SELECT-Befehl abgefangen werden. Beispiele sind das Teilen durch 0 oder Zugriffe über nicht existierende Objekte (z.B. SELF.Ehepartner.Alter). In diesen Fällen wird der Ausdruck

als FALSE interpretiert, so daß das entsprechende Objekt nicht in der Ergebnismenge enthalten ist. Auf diese Art können Abfragen wesentlich einfacher formuliert werden, da keine Ausnahmebehandlungen notwendig sind.

Um von externen Quellen eingelesene Mikrodatenbasen zu untersuchen und nachzubearbeiten, gibt es den Spezialbefehl

SELECT_MV,

mit dem alle Objekte selektiert werden, bei denen ein Attribut einen Missing-Value besitzt (bei reinen INTEGER- und REAL-Werten jedoch nicht feststellbar).

Zum Teil ist es notwendig, eine Menge von Objekten in eine Menge korrespondierender Objekte zu transformieren. So könnte zu einer Menge von Haushalten die Menge aller darin lebenden Personen gesucht werden, oder zu einer Menge von Personen die Menge ihrer Ehepartner. Hierfür wird der **COLLECT**-Befehl verwendet, der für das erste Beispiel wie folgt genutzt werden könnte:

GETCLASS (Haushalt) | COLLECT (SELF.Personen)

Der zu COLLECT gehörende Ausdruck muß entweder ein einzelnes Objekt (z.B. SELF.Ehepartner) oder ein SET (z.B. SELF.Personen) liefern. Das Ergebnis ist jeweils ein SET der Objektklasse. Wird für einzelne Objekte der Eingangsmenge des Ausdrucks ein leeres SET oder einen Objektzeiger "NIL" geliefert, bleiben sie unberücksichtigt. Gleiches gilt für die meisten Laufzeitfehler (z.B. Teilen durch 0), die abgefangen werden und zum Ausschluß des betroffenen Objektes führen.

Die beiden MQL-Befehle SELECT und COLLECT geben Mengen als Ergebnis zurück. Die übrigen MQL-Befehle liefern hingegen andere Ergebnistypen und können deshalb nur am Ende einer MQL-Kette stehen.

Der einfachste Befehl ist **SIZE**, der - ebenso wie die Funktion SIZEOF - die Anzahl der Elemente einer Menge bestimmt. Z.B. gibt

GETCLASS (Person) | SIZE

einen INTEGER-Wert mit der Anzahl aller Personen der Mikrodatenbasis zurück.

Die übrigen MQL-Befehle dienen dazu, auf bestimmte Attribute der Objekte zuzugreifen. Dies sind die Befehle

MINIMUM (*ausdruck*)

MAXIMUM (*ausdruck*)

SUM (*ausdruck*)

AVERAGE (*ausdruck*)

Mit "ausdruck" ist hierbei grundsätzlich ein beliebiger (arithmetischer) Ausdruck gemeint. Die Verwendung von Attributen der jeweiligen Objekte (mit "SELF.attribut") ist zwar nicht zwingend, anderenfalls ist die Abfrage jedoch nicht sinnvoll. Eine Beschränkung ergibt sich jedoch aufgrund der Funktion des Befehls für den Ergebnistyp des Ausdrucks:

- Die Verwendung der Befehle MINIMUM und MAXIMUM ist auf Ausdrücke beschränkt, die mindestens ordinal sind. Dies sind neben allen arithmetischen Ausdrücken auch ENUM-Typen, die in der TYPE-Definition als ordinal angegeben sind. Boolesche Ausdrücke sind nicht möglich.

- Die Ausdrücke für die Befehle SUM und AVERAGE sind selbstverständlich auf INTEGER- und REAL-Typen beschränkt.

Das Ergebnis hat jeweils denselben Typ wie der Ausdruck, bei der Durchschnittsbildung mit AVERAGE jedoch immer REAL.

Da die Objekte einer Mikrodatenbasis im allgemeinen als Teil einer Stichprobe eine größere Grundgesamtheit repräsentieren, besitzen sie meist einen Hochrechnungsfaktor, der von 1 verschieden ist. Um Ergebnisse zu erhalten, die auf die Grundgesamtheit hochgerechnet sind, können die Befehle SIZE, SUM und AVERAGE durch vorangestelltes "W" (für weight) entsprechend verändert werden (z.B. **WSIZE** usw.).

Der MQL-Befehl **DISTRIBUTION** unterscheidet sich grundlegend von den übrigen. Er liefert ein Ergebnis des Typs DISTR, der ausschließlich für Dokumentationszwecke verwendet wird. Allgemein ist ein Ausdruck des Typs DISTR eine 1- oder 2-dimensionale Verteilung, die sich auch als Tabelle auffassen läßt.

Ein einfaches Beispiel ist

GETCLASS (Person) | DISTRIBUTION (SELF.Familienstand)

Hierdurch entstehen vier Wertepaare aus den möglichen Familienständen (ledig, verheiratet, geschieden, verwitwet) zusammen mit den Häufigkeiten. Dabei werden gewichtete und ungewichtete Häufigkeiten gleichzeitig erfaßt und erst bei Ausgabe getrennt verwendet. Ein Befehl "WDISTRIBUTION" ist somit unnötig.

Liefert der Ausdruck, der zu DISTRIBUTION gehört, numerische Werte, so werden auch hier Werte/Häufigkeits-Paare gebildet, jedoch in einer unbekannten Anzahl. Sofern es sich nicht um INTEGER-Werte in einem beschränkten Bereich handelt, z.B. Kinderzahl oder auch Alter einer Person, muß spätestens bei der Ausgabe eine Klassifizierung vorgenommen werden (z.B. 0-9, 10-19, 20-29 ...). Da reale Mikrodatenbasen Tausende von Objekten enthalten, können Datenmenge und Rechenzeit stark ansteigen. Deshalb ist es bei numerischen Ausdrücken möglich, als zweiten Parameter eine Auflösung anzugeben. Im Ergebnis wirkt sich dies so aus, daß anstelle der originalen Werte solche gespeichert werden, die auf ganzzahlige Vielfache der Klassenbreite abgerundet wurden. Beispiel:

GETCLASS (Person) | DISTRIBUTION (SELF.Alter, 10)

Die Werte 0 bis 9 werden als 0 abgelegt, die Werte von 10 bis 19 als 10 usw. Es ist auch möglich, REAL-Zahlen als Klassenbreite anzugeben. Dies ist dann sinnvoll, wenn sehr kleine Werte wie z.B. Sparquoten o.ä. verarbeitet werden. Ansonsten ist dies aufgrund des höheren Speicherbedarfs und der ansteigenden Rechenzeit möglichst zu vermeiden.

Für die Ausgabe in Reports usw. sind zusätzliche Angaben möglich. So kann zusätzlich zur Auflösung eine Klassenbreite angegebenen werden, die in Tabellen und Histogrammen verwendet wird. Fehlt diese Angabe, so wird - wenn vorhanden - die Auflösung verwendet. Ansonsten ist der Default-Wert 1. Soll nur ein Ausschnitt aus der Verteilung untersucht werden, können eine Startklasse und die gewünschte Anzahl von Klassen angegeben werden. Ein String dient der Beschriftung der Tabellen. Insgesamt ergibt sich für eindimensionale Verteilungen folgende Syntax:

DISTRIBUTION (*ausdruck* [, *auflösung* [, *klassenbreite* [, *startklasse* [, *klassenzahl*]]]]
[, *string*])

Oft interessieren auch Zusammenhänge zwischen zwei Merkmalen, z.B. der Familienstand oder das Einkommen in Abhängigkeit vom Alter oder der Zusammenhang von Familienstand

und Stellung zum Haushaltsvorstand. Der Befehl DISTRIBUTION erlaubt deshalb auch die Angabe von zwei Merkmalen (bzw. Ausdrücken). Um den Ausdruck für die zweite Dimension sicher von den optionalen Angaben der ersten Dimension (z.B. Auflösung) trennen zu können, müssen entweder alle numerischen Parameter für den X-Wert vorhanden sein oder es ist zumindest der String (auch als Leerstring '') als Trennung einzufügen. Beispiel:

```
GETCLASS (Person) |
    DISTRIBUTION (SELF.Familienstand, 'Familienstand', SELF.Alter, 'Alter')
```

Die Typen der beiden Ausdrücke sind völlig unabhängig voneinander, so daß beliebige Kombinationen von BOOLEAN-, ENUM- und numerischen Ausdrücken zulässig sind. Die Beschränkung ergibt sich hierbei lediglich in der Ausgabe der Ergebnisse, die z.B. natürlich nur für zwei numerische Ausdrücke ein Streudiagramm ermöglicht.

Für die zweite Dimension sind alle Optionen wie bei der ersten Dimension möglich. Es ergibt sich also folgende Syntax:

DISTRIBUTION (*ausdruck1* [, *auflösung1* [, *klassenbreite1* [, *startklasse1*
[, *klassenzahl1*]]]] [, *string1*
[, *ausdruck2* [, *auflösung2* [, *klassenbreite2* [, *startklasse2*
[, *klassenzahl2*]]]] [, *string2*]]])

Der Befehl DISTRIBUTION besitzt in dieser Form die Einschränkung, daß er nur für Bestandsdaten geeignet ist. Will man z.B. die Verteilung des Alters der Mütter bei der Niederkunft bestimmen, so ist die Verteilung sukzessive jeweils bei einem Ereigniseintritt aufzubauen. Dazu dient die Anweisung DISTR_EVENT, die folgende Syntax besitzt:

DISTR_EVENT (*distr*-variable, *faktor*, *ausdruck1*, *string1* [, *ausdruck2*, *string2*])

Durch den wiederholten Aufruf der Anweisung wird eine Variable des Typs DISTR oder eine entsprechende Interface-Variable mit einer Verteilung gefüllt. Die Ausdrücke dürfen wie beim MQL-Befehl DISTRIBUTION alle dort zulässigen Typen besitzen. Es ist jedoch darauf zu achten, daß bei den Folgeaufrufen genau dieselben Typen als Argumente verwendet werden wie beim ersten Aufruf. Ansonsten ergibt sich ein Laufzeitfehler.

Die erzeugten ein- oder zweidimensionale Verteilungen können innerhalb von MISTRAL. mit dem Befehl

WRITETABLE (*distribution*, [, *string*] [, *breite*] [, *gewichtet* [, *relativ*]])

in den Report ausgegeben werden. Einzelheiten dazu in Abschnitt 10.2.

7.5 Prioritätsregeln für Ausdrücke

Die Reihenfolge der Auswertung innerhalb von Ausdrücken ist entscheidend für das Ergebnis. Dabei gelten folgende Regeln:

1. Das Innere von Klammern wird vor dem Äußeren ausgeführt.

2. Operatoren einer höheren Priorität (kleinere Stufe, s.u.) werden zuerst ausgeführt.

3. Operatoren gleicher Prioritätsstufe werden von links nach rechts ausgewertet.

Für die Operatoren in MISTRAL ergeben sich folgende Prioritäten:

Stufe	Operator	Operarandentyp(en)	Ergebnistyp	Bedeutung
einstellige:				
0	NOT	BOOLEAN	BOOLEAN	Negation
	+	INTEGER	INTEGER	Identität
	+	REAL	REAL	Identität
	-	INTEGER	INTEGER	Vorzeichenumkehr
	-	REAL	REAL	Vorzeichenumkehr
zweistellige:				
1	*	INTEGER	INTEGER	Multiplikation
	*	REAL	REAL	Multiplikation
	*	INTEGER, REAL	REAL	Multiplikation
	*	SET	SET	Schnittmenge
	/	INTEGER	INTEGER	Division
	/	REAL	REAL	Division
	/	INTEGER, REAL	REAL	Division
	DIV	INTEGER	INTEGER	Division ohne Rest
	MOD	INTEGER	INTEGER	Modulo (Divisions-Rest)
	AND	BOOLEAN	BOOLEAN	UND, Konjunktion
2	+	INTEGER	INTEGER	Addition
	+	REAL	REAL	Addition
	+	REAL, INTEGER	REAL	Addition
	+	SET	SET	Vereinigungsmenge
	-	INTEGER	INTEGER	Subtraktion
	-	REAL	REAL	Subtraktion
	-	REAL,INTEGER	REAL	Subtraktion
	-	SET	SET	Differenzmenge
	OR	BOOLEAN	BOOLEAN	ODER, Disjunktion
3	=	INTEGER, REAL	BOOLEAN	gleich
	=	BOOLEAN, BOOLEAN3	BOOLEAN	gleich
	=	ENUM	BOOLEAN	gleich
	=	Objekt	BOOLEAN	gleich (Identität)
	=	SET	BOOLEAN	gleich
	<>	INTEGER, REAL	BOOLEAN	ungleich
	<>	BOOLEAN, BOOLEAN3	BOOLEAN	ungleich
	<>	ENUM	BOOLEAN	ungleich
	<>	Objekt	BOOLEAN	ungleich (Nicht-Identität)
	<>	SET	BOOLEAN	ungleich
	<	INTEGER, REAL	BOOLEAN	kleiner
	<	ENUM (*ordinal*)	BOOLEAN	kleiner
	<=	INTEGER, REAL	BOOLEAN	kleiner oder gleich
	<=	ENUM (*ordinal*)	BOOLEAN	kleiner oder gleich
	>	INTEGER, REAL	BOOLEAN	größer
	>	ENUM (*ordinal*)	BOOLEAN	größer
	>=	INTEGER, REAL	BOOLEAN	größer oder gleich
	>=	ENUM (*ordinal*)	BOOLEAN	größer oder gleich
	a IN b	a = Objekt, b = SET	BOOLEAN	Enthaltensein
4	|	SET, MQL-Befehl	verschieden	verschieden

8 Anweisungen

8.1 Zuweisungen

Zuweisungen werden in folgender Form vorgenommen:

var-name := *ausdruck*

Die Variable muß zuvor deklariert worden sein.

Der Ausdruck muß vom gleichen Typ sein wie die Variable. Eine Ausnahme bilden INTEGER-Ausdrücke, die auch REAL-Variablen zugewiesen werden dürfen, sowie BOOLEAN-Ausdrücke und BOOLEAN3-Variablen. In diesen Fall findet eine automatische Umwandlung statt.

8.2 Verbundanweisungen

PASCAL kennt nur die bekannte Verbundanweisung:

BEGIN *anweisung* {; *anweisung*} **END**

In MISTRAL gibt es zusätzlich eine Verbundanweisung, mit der die enthaltenen Anweisungen in zufälliger Reihenfolge abgearbeitet werden. Dies ist in der Mikrosimulation zur Nachbildung von Simultanität notwendig.

BEGIN_RANDOM *anweisung* {; *anweisung*} **END**

8.3 Bedingte Anweisungen

8.3.1 IF-Anweisung

Die IF-Anweisung wird in der in PASCAL üblichen Form verwendet:

IF *boolean-wert* **THEN** *anweisung* [**ELSE** *anweisung*]

Ist die Anweisung im THEN-Teil selbst wieder eine IF-Anweisung mit einem ELSE-Teil, so wird die Mehrdeutigkeit dadurch aufgelöst, daß das ELSE zum jeweils letzten IF gehört.

Wie in PASCAL ist auch hier zu beachten, daß vor dem ELSE kein Semikolon stehen darf.

8.3.2 CASE-Anweisung

Für die Mehrfachauswahl steht - wie in PASCAL - die CASE-Anweisung zur Verfügung:

CASE *ausdruck* **OF**
const, const: *anweisung*;
const: *anweisung*;
const..const: *anweisung*
[**ELSE** *anweisung* {; *anweisung*}]
END

Der Ausdruck der CASE-Anweisung muß ein Wert vom Typ INTEGER, BOOLEAN (incl. BOOLEAN3) oder ein ENUM-Typ sein.

Die Auswahlwerte müssen natürlich vom gleichen Typ wie der Ausdruck sein und können beliebige Kombinationen aus Einzelwerten und Bereichen (oben vor 3. Anweisung) bestehen. Bereiche sind auch für nominale ENUM-Typen zulässig, wobei die Reihenfolge in der Definition (ev. beginnend bei DOES_NOT_APPLY) verwendet wird. Bei INTEGER-Werten ist im Gegensatz zu PASCAL keine Beschränkung des Bereichs (z.B. -32768 .. 32767) vorhanden.

8.4 Schleifenanweisungen

8.4.1 FOR-Anweisung

Die FOR-Anweisung wird in der üblichen Form verwendet, wenn die Anzahl von Schleifendurchläufen im voraus feststeht.

FOR *variable* := *ausdruck* (**TO** | **DOWNTO**) *ausdruck* **DO** *anweisung*

Die Variable und beide Ausdrücke müssen vom Typ INTEGER, BOOLEAN oder einem ENUM-Typen sein. Im letzten Fall wird die Reihenfolge in der TYPE-Definition als aufsteigend angesehen (gegebenenfalls mit DOES_NOT_APPLY beginnend). BOOLEAN-Typen besitzen die Reihenfolge DOES_NOT_APPLY, TRUE, FALSE.

Ist im Falle von "TO" der erste Ausdruck kleiner als der zweite, wird die Anweisung nicht ausgeführt. Gleiches gilt umgekehrt bei "DOWNTO".

8.4.2 WHILE-Anweisung

Bei der WHILE-Schleife wird vor jedem Durchlauf geprüft, ob die Bedingung erfüllt ist. Die Anweisung wird dann so oft ausgeführt, solange dies der Fall ist. Ist die Bedingung bei der ersten Abfrage nicht erfüllt, wird die Anweisung überhaupt nicht ausgeführt.

WHILE *ausdruck* **DO** *anweisung*

Der Ausdruck muß vom Typ BOOLEAN (oder BOOLEAN3) sein.

8.4.3 REPEAT-Anweisung

Die REPEAT-Schleife unterscheidet sich von der WHILE-Schleife dadurch, daß die Bedingung erst nach Abarbeiten der Anweisungen geprüft wird. Es wird also in jedem Fall mindestens ein Schleifendurchlauf ausgeführt.

REPEAT *anweisung* {; *anweisung*} **UNTIL** *ausdruck*

Auch hier muß der Ausdruck vom Typ BOOLEAN (oder BOOLEAN3) sein.

8.4.4 FOREACH-Anweisung

Neben den aus PASCAL bekannten Schleifenanweisungen existiert eine Schleife über alle Elemente einer Objektmenge. Dabei gibt es eine deterministische Variante und eine, bei der die Reihenfolge der durchlaufenen Objekte bei jedem Durchlauf neu durch einen (Pseudo-) Zufallsprozeß ermittelt wird. Ähnlich einer FOR-Schleife durchläuft die Variable alle möglichen Werte, diese sind hier jedoch (Zeiger auf) Objekte.

FOREACH *variable* **IN** *objekt-menge* (**DO | DO_RANDOM**) *anweisung*

Die Variable ist ein Objekttyp, die Objektmenge muß ein SET dieses Typs sein.

Es ist wichtig, sich die Wirkung einer FOREACH-Schleife als Abarbeiten einer Liste von Objekten klarzumachen. Aufgrund der inhaltlichen Gegebenheiten der Mikrosimulation kann es notwendig sein, die zugrundeliegende Menge während der Ausführung der Schleife zu verändern. Hierzu ein typisches Beispiel:

> Innerhalb eines Haushaltes wird eine Schleife über alle Personen dieses Haushaltes durchlaufen. Für jede dieser Personen werden demographische Module (z.B. Heirat, Scheidung Tod) durchlaufen. Heiratet z.B. eine Person, die Kinder hat, so werden diese bei einem Umzug in einen anderen Haushalt mitgenommen. Damit werden sie jedoch aus der Menge gelöscht, über die noch eine offene FOREACH-Schleife läuft. Umgekehrt können Frauen während des Schleifendurchlaufs Kinder bekommen, die ihrerseits noch in diesem Durchlauf simuliert werden sollen.

Für die Wirkung der Befehle ADD und REMOVE auf offene FOREACH-Schleifen, die über alle Objekte der betroffenen Menge laufen, gilt folgendes:

Wird ein Objekt mit REMOVE aus einer aktuell bearbeiteten Menge entfernt, so wird es auch nicht mehr in der Schleife bearbeitet, sofern das noch nicht geschehen ist. Wird ein Objekt zu einer Menge mit ADD hinzugefügt, so wird es an das Ende einer Schleife gesetzt, die über dieser Menge aktiv ist.

In diesem Zusammenhang entsprechen die Befehle NEW und DELETE einer Operation auf der Gesamtmenge der Objekte einer Klasse. Bei einer Schleife über eine mit „GETCLASS()“ erzeugte Menge werden die betreffenden Objekte deshalb auch hier aus der Verarbeitungsschleife entfernt bzw. hinzugefügt.

SETs sollten in MISTRAL grundsätzlich als ungeordnete Mengen aufgefaßt werden, so daß es keine Reihenfolge der Elemente gibt. Dennoch ist es aus Gründen der Reproduzierbarkeit von Simulationsergebnissen z.T. notwendig, eine feste Reihenfolge für die FOREACH-Schleife zu haben. Deshalb werden die Elemente in aussteigender Reihenfolge ihrer Schlüsselnummern verarbeitet. Soll eine zufällige Reihenfolge verwendet werden, ist "FOREACH ... IN ... DO_RANDOM" zu verwenden.

8.4.5 Vorzeitiger Schleifenabbruch

Ein vorzeitiger Abbruch aus einer Schleife ist mit dem Befehl

BREAK

möglich. Wird dieser Befehl außerhalb einer Schleife verwendet, so wird das aktuelle Unterprogramm verlassen (wirkt dann wie ein EXIT-Befehl).

9 Unterprogramme

9.1 Allgemeines

MISTRAL besitzt eine differenzierte Hierarchie von Unterprogrammen, deren Bedeutung sich z.T. aus der Situation der Mikrosimulation ergibt.

Zunächst ist eine Unterscheidung von objektgebundenen und nicht objektgebundenen Unterprogrammen sinnvoll. Die letzteren entsprechen den aus PASCAL bekannten normalen Prozeduren und Funktionen.

Die übrigen Unterprogramme sind - ähnlich den aktuellen objektorientierten PASCAL-Erweiterungen - einer Objektklasse zugeordnet und werden zusammenfassend als Methoden bezeichnet. Es werden folgende Methodenarten unterschieden, die hier kurz im Überblick dargestellt werden:

Super-Modul: Sie fassen mehrere Module zu größeren Einheiten zusammen und dienen damit der Strukturierung des Gesamtmodells. Beispiele sind die Zusammenfassung von demographischen, ökonomischen oder Erwerbsbeteiligungs-Modulen.

Modul: Die Module enthalten die eigentlichen Mikroprogramme (z.B. Tod, Geburt usw.).

Prozedur: Prozeduren sind Methoden, die bestimmte Teilaufgaben eines Objektes realisieren. Prozeduren können mit Werteparametern aufgerufen werden. Hier werden üblicherweise Hilfsaufgaben realisiert, z.B. der Umzug einer Person von einem Haushalt in einen anderen. Prozeduren verändern bestimmte Daten, liefern jedoch keinen Wert zurück.

Funktion: Funktionen realisieren wie Prozeduren bestimmte Teilaufgaben. Sie können ebenfalls mit Werteparametern aufgerufen werden. Der Unterschied besteht darin, daß Funktionen einen Wert zurückliefern.

Damit Methoden aufgerufen werden können, müssen sie zuvor definiert worden sein. Rekursion ist nicht zulässig; damit entfällt auch die Notwendigkeit einer Forward-Deklaration.

Das zentrale Element eines MISTRAL-Programms ist das Modul, in dem ein exakt definierter Teil des Gesamtmodells formuliert wird. Module sollten immer so aufgebaut sein, daß sie keinerlei globale Konstanten, Variablen oder Methoden benötigen. Sie sollten einzig und allein von der Definition der Mikrodaten in der TYPE- und CLASS-Sektion abhängen. Um die Abgeschlossenheit zu unterstützen, ist es möglich, innerhalb von Modulen lokale Prozeduren und Funktionen zu definieren. Weitere Schachtelungen (rekursive Definitionen) sind nicht möglich.

Alle Methodenarten erlauben die Definition lokaler Konstanten und Variablen. Hierzu werden wie im Gesamtprogramm die CONST- und VAR-Sektion verwendet.

Ebenso wie in PASCAL ist eine Schachtelung von Unterprogramm-Vereinbarungen möglich. Um der inhaltlichen Unterscheidung der verschiedenen Unterprogrammarten gerecht zu werden, wurden jedoch Einschränkungen vorgesehen. Folgende Schachtelungen sind möglich:

übergeordnetes Unterprogramm	**mögliche geschachtelte Unterprogramme**
Super-Modul	Super-Modul, Modul, Prozedur, Funktion
Modul	Prozedur, Funktion
Prozedur	Super-Modul, Modul, Prozedur, Funktion
Funktion	Prozedur, Funktion

Ein Unterprogramm kann vorzeitig mit dem Befehl

EXIT

verlassen werden. Befindet sich dieser Befehl im Hauptprogramm, wird die gesamte Programmausführung beendet.

9.2 Parameter und Rückgabewerte

Um Seiteneffekte weitgehend zu vermeiden, werden alle Parameter nur als Werteparameter übergeben ("call-by-value"). Eine Besonderheit stellen jedoch Objekte bzw. Mengen von Objekten dar, da hier der Wert der Variablen ja nur einen Zeiger auf ein Objekt darstellt. Auf diese Art kann z.B. einer globalen Variablen, die einer Prozedur als Parameter übergeben wird, kein anderes Objekt zugewiesen werden. Es ist jedoch möglich (und bei Prozeduren sogar die Hauptaufgabe), die Attribute des Objekts (und verbundener Objekte), auf das gezeigt wird, zu verändern.

Die Vereinbarung der formalen Parameter eines Unterprogramms hat folgende Syntax:

"("*name* {"," *name*}":" *typ* {";" *name* {"," *name*}":" *typ*}")"

Beispiel:

(a,b: INTEGER; c: REAL)

Bezüglich der Typen für die Parameter und Rückgabewerte kennt MISTRAL - anders als PASCAL - keine Einschränkungen. So können Mengen (SETs) und Objekte sowohl als Parameter als auch als Rückgabewerte (von Funktionen) vereinbart werden.

9.3 Unterprogrammaufruf

Unterprogramme, die einen Rückgabewert liefern, können nur innerhalb von Ausdrücken verwendet werden; Unterprogramme ohne Rückgabewert dürfen überall dort stehen, wo Anweisungen erwartet werden.

Sofern ein Unterprogramm Parameter erwartet, sind diese - jeweils durch Komma getrennt - hinter dem Unterprogrammaufruf in Klammern anzugeben. Als Parameter sind überall (beliebig komplizierte) Ausdrücke möglich, die jedoch vom passenden Typ sein müssen.

Objektgebundene Unterprogramme (Methoden) werden in der Form

objekt.methode

aufgerufen. Dabei kann auch das Objekt aus einem komplizierteren Aufruf entstanden sein, z.B.:

P.Ehepartner.*methode*

9.4 Super-Modul

Super-Module dienen dazu, den Aufruf mehrerer Module zu einer logischen Einheit zusammenzufassen. Zusätzlich können kleinere zentrale Modellteile realisiert werden, die sich schlecht in einem Modul unterbringen lassen (z.B. Alter um eins erhöhen).

Die Definition eines Super-Moduls hat folgendes Aussehen:

SUPERMODULE *klasse.name*;
[*konstanten-definition*]
[*variablen-definition*]
[*unterprogramm-vereinbarungen*]
BEGIN
{*anweisungen*}
END;

Als Unterprogramme innerhalb eines Super-Moduls können Super-Module, Module, Prozeduren und Funktionen vereinbart werden.

9.5 Modul

Das Modul ist das zentrale Element eines MISTRAL-Programms für die Simulation.

Innerhalb eines Moduls wird ein exakt definierter Modellteil (z.B. Geburt, Heirat, Scheidung, Tod usw.) abgearbeitet.

Die Definition eines Moduls hat folgendes Aussehen:

MODULE *klasse.name*;
[*konstanten-definition*]
[*variablen-definition*]
[*unterprogramm-vereinbarungen*]
BEGIN
{*anweisungen*}
END;

Als Unterprogramme innerhalb eines Moduls können Prozeduren und Funktionen vereinbart werden.

9.6 Prozedur

Die Prozedur dient dazu, wiederkehrende Aktionen mit veränderndem Charakter zentral zu definieren oder aus Gründen der besseren Übersichtlichkeit zu separieren.

Prozeduren können als Methoden oder unabhängig von einer Objektklasse vereinbart werden. Im ersten Fall ist vor dem Prozedurnamen der Klassenname anzugeben.

Prozeduren können optional formale Parameter besitzen.

Die Syntax der Prozedurvereinbarung lautet:

PROCEDURE [*klasse*.] *name* [*parameter-definition*];
[*konstanten-definition*]
[*variablen-definition*]
[*unterprogramm-vereinbarungen*]
BEGIN
{*anweisungen*}
END;

Innerhalb einer Prozedur können Prozeduren und Funktionen vereinbart werden.

9.7 Funktion

Funktionen liefern einen bestimmten Wert zurück, der innerhalb der Funktion - in Abhängigkeit von den optionalen Parametern - berechnet wird.

Auch Funktionen können - wie Prozeduren - objektunabhängig oder als Methoden vereinbart werden.

Die Syntax der Funktionsvereinbarung lautet:

FUNCTION [*klasse*.] *name* [*parameter-definition*] : *typ*;
[*konstanten-definition*]
[*variablen-definition*]
[*unterprogramm-definitionen*]
BEGIN
{*anweisungen*}
END;

Innerhalb einer Funktion können Prozeduren und Funktionen vereinbart werden.

Der Rückgabewert wird dadurch definiert, daß er dem Namen der Funktion (ohne Klassenprefix) wie einer Variablen zugewiesen wird. Beispiel:

f1 := 1

9.8 Vorzeitiges Verlassen

Unterprogramme können an jeder Stelle der Ausführung vorzeitig mit dem Befehl

EXIT

verlassen werden.

Eine andere, implizite Form des vorzeitigen Ausstiegs wird dann durchgeführt, wenn ein Objekt mit dem Befehl DELETE gelöscht wurde. In diesem Fall werden alle aktuellen Methoden dieses Objekts automatisch beendet. Dies ist beispielsweise dann der Fall, wenn eine Person in einem Modul "Tod" gestorben ist. Dann wird auch automatisch die Abarbeitung eines übergeordneten Super-Moduls abgebrochen.

Tritt während der Programmausführung ein Fehler auf, der zum sofortigen Programmabbruch führen soll, kann dies mit dem Befehl

ERROR (*string*)

erreicht werden. Es erscheint ein Fehlerfenster mit dem angegebenen String, das der Benutzer bestätigen muß. Zusätzlich wird der String mit einer entsprechenden Meldung auch in den Report geschrieben. ERROR kann an jeder Stelle des Programms verwendet werden.

10 Ausgabe

10.1 WRITE

Die einfachste Form der Ausgabe stellen die aus PASCAL bekannten Befehle WRITE und WRITELN dar. Die Ausgabe erfolgt in einen Report, der automatisch erzeugt wird. Die Aufrufe sehen folgendermaßen aus:

WRITE (*ausdruck* {, *ausdruck*})

WRITELN (*ausdruck* {, *ausdruck*})

Der Unterschied zwischen beiden Befehlen besteht darin, daß WRITELN am Ende eine Zeilenschaltung anhängt.

Wie zu sehen ist, kann eine beliebige Anzahl von Ausdrücken verwendet werden, deren Darstellungen in der Ausgabe direkt aneinander gehängt werden. Die Ausdrücke können von einem beliebigen Typ (außer DISTR) sein, also - anders als bei PASCAL z.B. auch Objekte, SETs oder ENUM-Werte sein. Das Ausgabeformat wird anhand folgender Beispiele deutlich:

Typ	**Beispiele**	
INTEGER	"1", "-23"	
REAL	"0.1", "-234.0"	
BOOLEAN	"TRUE", "FALSE"	
BOOLEAN3	"TRUE", "FALSE"	"-"(für DOES_NOT_APPLY)
ENUM	"hv", "gatte" (immer in Kleinbuchstaben)	"-"(für DOES_NOT_APPLY)
Objekt	"<123>" (Schlüssel-Nr.)	"-"(für NIL)
SET	"<101>,<102>,<103>"	"-"(für EMPTYSET)

Wie in PASCAL sind auch hier Formatierungsergänzungen möglich:

ausdruck:*b*	*b* ist vom Typ INTEGER und gibt die Breite der Ausgabe an für $b > 0$ wird der Ausdruck rechtsbündig geschrieben, für $b < 0$ linksbündig; ist der Betrag *b* kleiner als die benötigte Stellenzahl, hat der Zusatz keine Wirkung
real-ausdruck:*b*:*s*	*b* hat dieselbe Wirkung wie oben *s* gibt die Zahl der Nachkommastellen an

10.2 WRITETABLE

Ein- oder zweidimensionale Verteilungen können innerhalb von MISTRAL mit dem Befehl

WRITETABLE (*distribution*, [, *string*] [, *breite*] [, *gewichtet* [, *relativ*]])

in den Report ausgegeben werden.

Nach der Verteilung (z.B. ein Ausdruck oder eine Variable des Typs DISTR) folgen mehrere optionale Argumente. Der *string* wird als Überschrift der Tabelle ausgegeben. Der INTEGER-Wert *breite* gibt die maximal verfügbare Zeilenbreite in Anzahl der Zeichen an. Eine umfang-

reichere zweidimensionale Tabelle wird gegebenenfalls automatisch entsprechend umgebrochen.

Mit dem BOOLEAN-Wert *gewichtet* wird festgelegt, ob gewichtete (default) oder ungewichtete Werte ausgegeben werden. Der BOOLEAN-Wert *relativ* gibt an, ob absolute Werte (default) oder Prozentangaben geschrieben werden. Beide Angaben werden bei eindimensionalen Tabellen ignoriert, da dort immer alle vier Varianten angegeben werden.

Die Klassenbreite ergibt sich aus den entsprechenden Angaben, die beim DISTRIBUTION-Befehl gemacht wurden (Default = 1). Führt dies zu mehr als 150 Klassen, so wird die Klassenbreite mit Vielfachen von 2, 5, 10 usw. soweit erhöht, bis sich eine geringere Klassenzahl ergibt.

10.3 Events

Eine wichtige Anweisung, die vor allem innerhalb von Modulen vorkommt, ist folgende:

EVENT (*string* [, *hochrechnungs-faktor*])

Der String beschreibt das zu protokollierende Ereignis (z.B. "Tod"). Der Hochrechnungsfaktor dient dazu, die Mikroereignisse auf die Grundgesamtheit (z.B. die Bundesrepublik Deutschland) hochzurechnen. Dies ist notwendig, da z.B. die Personen einer Mikrodatenbasis meist unterschiedliche Hochrechnungsfaktoren besitzen. In den meisten Fällen kann diese Angabe entfallen, da als Default automatisch der Wert des aktuellen Objekts der Methode verwendet wird. Wichtig ist diese Angabe dann, wenn der Faktor eines anderen Objektes verwendet werden soll (z.B. der abweichende Faktor beider Ehepartner bei der Heirat).

Die Ereignisse werden während der Simulation gesammelt und können mit der Anweisung

WRITEEVENTTABLE

in Form einer umfangreichen Tabelle mit verschiedenen Auswertungen in den Report übernommen werden.

Ein direkter Zugriff auf einzelne Häufigkeiten von Ereignissen ist mit den Funktionen

EVENTCOUNT (*string*)

und

EVENTWCOUNT (*string*)

möglich. Die erste Funktion gibt eine INTEGER_Zahl mit der ungewichteten Anzahl von Ereigniseintritten zurück, die zweite eine REAL-Zahl für die mit Hochrechnungsfaktoren gewichtete Anzahl.

10.4 STATUS

Im Gegensatz zu den bisherigen Ausgabemethoden dient die Prozedur STATUS dazu, während des Programmablaufs Meldungen in die Statuszeile zu schreiben und die Ablaufanzeige zu steuern. Der Aufruf lautet:

STATUS ([*string*] [, *real-zahl*])

Der String wird direkt in die Statuszeile geschrieben. Bei umfangreicheren Simulationen, bei denen dort z.B. die gerade simulierte Periode steht, wird der String an diese Standardmeldung angehängt. Ein erneuter Aufruf überschreibt die bisherige Statusmeldung bzw. den von der STATUS-Anweisung erzeugten Teil.

Mit der Angabe einer REAL-Zahl zwischen 0.0 und 1.0 wird die Laufanzeige entsprechend gesetzt.

Prinzipiell ist es innerhalb eines Programm unmöglich, die genaue Ablaufdauer im voraus festzustellen. Läuft innerhalb eines MISTRAL-Programms eine FOREACH-Anweisung ab, so wird in der Regel automatisch deren prozentualer Abarbeitungsstatus innerhalb der Statuszeile und Ablaufanzeige verwendet. Insbesondere bei umfangreichen Importroutinen im Rahmen einer Generierung bietet es sich jedoch an, daß der Anwender z.B. den Stand innerhalb einer einzulesenden Datei ausgibt.

11 Lesender Zugriff auf Textdateien

Innerhalb eines Mikrosimulationssystems werden viele Daten benötigt, die aus externen Quellen stammen. Dies gilt insbesondere für Mikrodaten, die in der Regel in Form von Textdateien (ASCII-Text) zur Verfügung stehen. Um solche Daten einlesen zu können, wurden einige Funktionen und Prozeduren implementiert, die insbesondere das Einlesen und Erzeugen von Mikrodatenbasen erleichtern und damit externe Programme zum Erzeugen von Mikrodaten-Files überflüssig machen.

Hier zunächst eine Erläuterung des allgemeinen Modells, das dem Dateizugriff zugrunde liegt:

Innerhalb eines MISTRAL-Programms (nur Teilsprache Generierung) können gleichzeitig mehrere (z.Z. acht) Dateien gleichzeitig für einen (ausschließlich) lesenden Zugriff geöffnet sein. Diese Dateien müssen in der FILE-Sektion im Vereinbarungsteil mit einem logischen Namen definiert und entweder dort oder interaktiv im Simulationssystem mit einer Datei innerhalb des Dateisystems des Rechners verbunden werden.

Mit dem Start des Programms wird jede der Dateien automatisch geöffnet. Für jede wird ein Lesepuffer angelegt, der immer genau eine aktuelle Zeile beinhaltet, auf die mittels verschiedener Funktionen zugegriffen werden kann.

Mit dem Start des Programms ist automatisch die jeweils erste Zeile im Puffer vorhanden. Mit der Prozedur

GETLINE (*file*)

wird für die angegebene Datei (Argument ist der logische Name) jeweils die nächste Zeile eingelesen und im Puffer bereitgehalten.

Wurde die letzte Zeile der Datei überlesen, wird im Puffer ein leerer String gehalten. Zugleich liefert die Funktion

END_OF_FILE (*file*)

den BOOLEAN-Wert TRUE, um anzuzeigen, daß das Dateiende erreicht wurde.

Im wesentlichen für Zwecke der Statusanzeige dient die Funktion

POSITION_IN_FILE (*file*),

die einen REAL-Wert zwischen 0.0 und 1.0 liefert und damit den relativen Fortschritt innerhalb des Einlesens der Datei angibt.

Der Zugriff auf den Lesepuffer einer Datei ist am einfachsten mit der Funktion

READLINE (*file*)

möglich. Damit wird die gesamte aktuelle Zeile der Datei als String zurückgegeben.

In den meisten Fällen werden jedoch einzelne Informationen innerhalb der Zeile benötigt, auf die gezielt zugegriffen werden soll. Dazu gibt es zwei Zugriffsmodelle:

Zugriff auf Spalte: Es wird davon ausgegangen, daß die Informationen in einer durchgehenden Zeile an bestimmten Spalten angeordnet sind, die in der Regel für alle Zeilen der Datei gleich sind. Eine vierstellige INTEGER-Zahl könnte dann z.B. von Spalte 123 bis 126 stehen.

Zugriff auf Feld: Häufig werden die Informationen auch hintereinander geschrieben und durch einen Separator (häufig ein Semikolon) getrennt. Dies erlaubt z.B.

die kompaktere Speicherung von Informationen mit sehr unterschiedlicher Länge (z.B. Strings oder langer Zahlen). Bei Zugriff auf das dritte Feld wird dann der Text zwischen dem zweiten und dritten Separator (bzw. dem Zeilenende) zurückgegeben.

In MISTRAL sind beide Zugriffsarten möglich. Um besonders flexibel zu sein und auch innerhalb einer Datei mit mehreren Abschnitten wechseln zu können, wird die Zugriffsart bei jedem Aufruf der unten beschriebenen Zugriffsfunktionen angegeben.

Für den Zugriff auf bestimmte Spalten sehen die Argumente der Funktionen so aus:

(*file*, *spalten-nr1* [, *spalten-nr2*])

Damit werden alle Zeichen von jeweils einschließlich *spalten-nr1* bis *spalten-nr2* eingelesen. Fehlt die zweite Spalten-Nr., so wird bis zum Ende der Zeile gelesen. Für das Einlesen nur eines Zeichens sind beide Spalten-Nrn. identisch.

Für den Zugriff auf bestimmte Felder sehen die Argumente der Funktionen so aus:

(*file*, *string*, *feld-nr*)

Der String entspricht dem Separator und besteht aus einem Zeichen (z.B. ';'). Umfaßt der String mehr als ein Zeichen, wird nur das erste davon verwendet. Mit der Funktion

NO_OF_FIELDS (*file*, *string*)

kann bestimmt werden, wie viele Felder in der aktuellen Zeile enthalten sind. Dies entspricht der Anzahl der Separatoren plus 1. Bei einer leeren Zeile wird 0 zurückgegeben.

Um komplizierte Umwandlungen zu vermeiden, kann mit den nachfolgenden Funktionen beim Zugriff direkt ein bestimmter Datentyp erzeugt werden (Angabe ohne die eben beschriebenen Argumente):

READSTRING (...)

READINTEGER (...)

READREAL (...)

Diese Funktionen liefern einen Wert des entsprechenden Typs und können direkt in einer Zuweisung oder einem Ausdruck verwendet werden.

Die Prüfungen, ob auf einen ungültigen Teil der Zeile (z.B. Spalten-Nr. größer als Zeilenlänge) zugriffen wurde oder im Zugriffsbereich keine gültige Zahl (des angegebenen Typs) enthalten ist, lassen sich aus Gründen der Effizienz und Fehlertoleranz vom Benutzer im Simulationssystem konfigurieren.

Bei READSTRING wird im Fehlerfall generell ein leerer String zurückgegeben. Bei den beiden anderen Funktionen wird prinzipiell eine 0 bzw. 0.0 erzeugt. Je nach Konfiguration erfolgt eine Warnung im Report oder ein sofortiger Programmabbruch.

Eine Besonderheit ergibt sich für die Angabe von Objektzeigern von Mikrodaten (z.B. auf den Ehepartner) in einer Textdatei. Diese Zeiger sind innerhalb der Textdatei prinzipiell INTEGER-Zahlen und müssen auch so eingelesen werden. Bei Zuweisen an die entsprechenden Attribute eines Objekts ergeben sich jedoch zwei Probleme. Zum einen verbietet die Typprüfung des Compilers, einer Zeigervariablen einen Zahlenwert zuzuweisen. Zum anderen existiert das entsprechende Objekt, auf das gezeigt werden soll, möglicherweise noch gar nicht, da es erst später in der Textdatei definiert ist.

Um diese Probleme zu lösen, wurde die Funktion

READPOINTER

implementiert, die zunächst eine INTEGER-Zahl liefert, die jedoch einer beliebigen Zeigervariablen zugewiesen werden kann. Mit der Prozedur

CREATEPOINTER

werden alle INTEGER-Zeiger in echte Objektzeiger umgewandelt. Dazu wird jeweils das Objekt der Klasse des Zeigertyps mit der passenden Schlüssel-Nr. (Attribut KEY) in der Mikrodatenbasis gesucht. Es kann innerhalb des Simulationssystems konfiguriert werden, ob bei einem fehlenden Objekt (bzw. Schlüssel) ein NIL-Zeiger erzeugt oder das Programm abgebrochen wird. Wird vor der Umwandlung von INTEGER- auf Objektzeiger versucht, über diesen Zeiger auf das entsprechende Objekt zuzugreifen, erfolgt ein Fehlerabbruch des Programms.

Innerhalb umfangreicher Import-Programme kann es - insbesondere aus Gründen der Modularisierung - notwendig sein, eine Datei mehrfach zu durchlaufen. Hierzu ist der Lesepuffer mit der Anweisung

RESET (*file*)

auf den Beginn der Datei zurückzusetzen.

12 Verschiedenes

12.1 Schlüsselnummern von Objekten

Jedes Objekt besitzt ein Attribut namens "Key", das zu seiner eindeutigen Identifikation innerhalb seiner Klasse verwendet wird. Diese Schlüssel sind positive INTEGER-Zahlen, die in der Regel beim Erzeugen eines neuen Objekts automatisch vergeben werden. Im Rahmen des Imports externer Mikrodaten kann den Objekten auch explizit ein neuer Schlüssel zugewiesen werden, sofern er noch nicht existiert:

object.Key := *neuerSchlüssel*

Die Numerierung der Schlüssel wird für jede Klasse getrennt vorgenommen, so daß z.B. eine Person und ein Haushalt die gleiche Schlüsselnummer besitzen, jedoch über ihrer Klasse unterschieden werden können.

SETs sind grundsätzlich ungeordnete Mengen von Objekten. Um jedoch interne Abläufe effizient und reproduzierbar zu gestalten, sind die Objekte innerhalb eines SETs nach aufsteigenden Schlüsseln geordnet. Wird ein SET mittels einer FOREACH-Anweisung bearbeitet, werden die Objekte immer in der Reihenfolge ihrer Schlüssel aufgerufen, sofern nicht explizit mit DO_RANDOM die zufällige Reihenfolge erzwungen wird. Dies erhöht zwar die Laufzeit eines Programms, ist jedoch zum Teil unverzichtbar, um systematische Verzerrungen zu vermeiden.

12.2 Zufallszahlen

Insbesondere im Rahmen der stochastischen Mikrosimulation, aber auch beim Generieren künstlicher Attribute für Mikroobjekte werden Zufallszahlen an vielen Stellen benötigt. Deshalb wird hier kurz erläutert, wie Zufallszahlen innerhalb von MISTRAL organisiert und verwendet werden.

Jedes Objekt besitzt ein implizites Attribut als aktuellen Wert, der von einem Pseudozufallsgenerator (multiplikativer Kongruenzgenerator) verwendet wird, um den jeweils nächsten Zufallswert zu bestimmen.

Damit enthält jedes Objekt seinen eigenen Zufallsgenerator. Damit wird im Rahmen wiederholter Simulationen eine Varianzreduzierung erreicht. Führt z.B. eine kleine Änderung im Modell dazu, daß ein Objekt eine zusätzliche Zufallszahl zieht, so bleiben die Zufallszahlen der übrigen davon unberührt. Bei einem zentralen Zufallsgenerator würden ab dieser Stelle in der Simulation alle Objekte völlig andere Zufallszahlen erhalten und damit ein ganz anderes Simulationsergebnis liefern.

Die Mikrodatenbasis selbst enthält einen Startwert für einen zentralen Zufallszahlengenerator, der in der Mikrodatenbasis gespeichert ist und im Simulationssystem verändert werden kann. Beim Einlesen der Mikrodatenbasis aus einer Datei wird jedem Objekt ein Startwert für dessen individuelle Zufallszahlen vergeben, der sich aus dem zentralen Zufallsgenerator ableitet. Bei jedem (z.B. im Rahmen einer Simulation) neu erzeugten Objekt wird für dieses ebenfalls ein Startwert generiert.

Werden innerhalb eines MISTRAL-Programms Zufallszahlen erzeugt (z.B. mit RANDOM oder PARAM_EVENT), so wird nach folgender Reihenfolge auf einen der möglichen Zufallsgeneratoren zugegriffen:

1. Ist bei bestimmten Funktionen (z.B. RANDOM) ein Objekt angegeben, wird dessen Zufallsgenerator verwendet.
2. Befindet sich das Programm bei der Ausführung aktuell in einer objektbezogenen Methode, wird der Zufallsgenerator dieses Objekts verwendet. Gleiches gilt für nicht objektbezogene Unterprogramme, die aus einer objektbezogenen Methode heraus aufgerufen wurden.
3. Anderenfalls wird der zentrale Zufallsgenerator benutzt.

Explizite Funktionen zum Erzeugen von Zufallszahlen sind in Abschnitt 7.1 angegeben.

12.3 Hochrechnungsfaktoren

Bei Mikrodaten handelt es sich grundsätzlich um eine Stichprobe, die eine größere Grundgesamtheit repräsentiert. In den meisten Fällen sind einzelne Gruppen der Grundgesamtheit über- oder unterrepräsentiert, so daß dies durch die Angabe eines individuellen Hochrechnungsfaktors ausgeglichen werden muß. Dieser Faktor ist bei jedem Objekt automatisch als REAL-Attribut "Factor" vorhanden.

Beim Erzeugen neuer Objekte ist es wichtig, diesen einen sinnvollen Hochrechnungsfaktor zu vergeben. Bei einer Geburt soll das Kind in der Regel den gleichen Faktor wie die Mutter erhalten. Dies kann etwa so angegeben werden:

```
kind.Factor := mutter.Factor;
```

Wird kein Faktor durch eine explizite Zuweisung vergeben, so erhält ein Objekt automatisch den Faktor 1.0, der jedoch meist ungeeignet ist.

Faktoren werden im wesentlichen an zwei Stellen ausgewertet.

Zum einen erlauben die Analysewerkzeuge bei der Untersuchung der Bestände sowohl für Anzahl als auch Summen und Durchschnitte die Auswertung der gewichteten oder der ungewichteten Mikrodaten. Deutlich wird dies bei einigen MQL-Befehlen, die in zwei Versionen (z.B. SUM und WSUM) verfügbar sind.

Zum anderen betrifft dies auch die Auswertung von Ereignissen, die im Rahmen einer Simulation stattgefunden haben. Diese werden über die Prozedur EVENT aufgezeichnet, die neben der Angabe eines identifizierenden Strings auch die optionale Angabe eines Hochrechnungsfaktors erlaubt. Innerhalb einer objektbezogenen Methode kann dieser jedoch in der Regel weggelassen werden, da automatisch der Faktor des aktuellen Objekts verwendet wird. Beim Aufruf von EVENT außerhalb einer Methode und ohne explizite Angabe eines Faktors wird 1.0 verwendet.

13 Parameterdatenbasis

13.1 Allgemeines

Innerhalb der Simulation findet eine Vielzahl von Ereignissen statt, die bestimmte Eintrittswahrscheinlichkeiten besitzen. Meist sind diese Wahrscheinlichkeiten in Abhängigkeit von mehreren Größen definiert. Z.B. können die Sterbewahrscheinlichkeiten nach den zwei Geschlechtern, vier Familienständen und 100 Altersklassen getrennt festgelegt sein, so daß sich 800 Einzelwerte ergeben. Zusätzlich ist zu berücksichtigen, daß sich die Wahrscheinlichkeiten im Zeitverlauf ändern und somit oft für jedes Jahr getrennt angegeben werden.

Im Gegensatz dazu ist der Algorithmus für das Sterben (z.B. Person löschen, Familienstand eines vorhandenen Gatten auf verwitwet setzen usw.) meist unabhängig von diesen Einflußgrößen.

Zusätzlich zu Wahrscheinlichkeiten können weitere - meist zeitvariante - Daten vorliegen, die nicht als Teil des Algorithmus aufzufassen sind (z.B. die Beitragsbemessungsgrenzen der Sozialversicherungen o.ä.). Deshalb werden diese Daten (inkl. der Wahrscheinlichkeiten) zusammenfassend als Fortschreibungsparameter bezeichnet.

Es ist also notwendig, Simulationsalgorithmus und Fortschreibungsparameter getrennt zu halten. Es werden deshalb das Simulationsprogramm und die Parameterdatenbasis unterschieden. Letztere wird ebenfalls vom Benutzer in Form eines Textes definiert, dessen Syntax ein spezieller Teil von MISTRAL ist.

Im nächsten Abschnitt wird zunächst die Verwendung von Fortschreibungsparametern innerhalb des Simulationsprogramms beschrieben; die darauffolgenden Abschnitte beschreiben das Format der Parameterdatenbasis.

13.2 Verwendung im Simulationsmodell

Fortschreibungsparameter sind in mehrdimensionalen Tabellen organisiert. Um den gewünschten Einzelwert auszuwählen, werden Eingangsgrößen benötigt. Im oben genannten Beispiel der Sterbewahrscheinlichkeiten z.B. Alter, Geschlecht und Familienstand einer Person. Innerhalb des Simulationsprogramms stellen sich die Parametertabellen somit wie Funktionen dar, die in der üblichen Form unter Angabe von Argumenten aufgerufen werden. Der Unterschied besteht lediglich darin, daß die Definition dieser Funktionen in einem anderen Programmtext erfolgt, der getrennt compiliert wird.

Obwohl Parallelen zum UNIT-Konzept von Turbo-PASCAL bestehen, wurde hier ein etwas anderer Weg beschritten, der mit einer Mischung aus FORWARD- und Variablen-Deklaration vergleichbar ist. Hierbei werden in einer speziellen PARAMETER-Sektion im Vereinbarungsteil des Programms alle vom Programm benötigten Parameter angegeben (die Parameterdatenbasis kann mehr enthalten).

Hierzu ein Beispiel:

```
PARAMETER
  Geburt
    (Alter: INTEGER;
      Familienstand: Familienstaende;
      erwerbstaetig: BOOLEAN): PROB;
```

```
KinderZahl
  (r: PROB): INTEGER;
```

Nach dem Namen der Parametertabelle folgt die Angabe der formalen Parameter der Tabelle in gleicher Weise wie bei einer Funktion. Ebenso wird der Typ des Rückgabewertes definiert. Für den Datenaustausch zwischen Simulationsprogramm und Parameterdatenbasis ist nicht die Reihenfolge der formalen Parameter von Bedeutung. Statt dessen wird die Übereinstimmung der Namen bei der Initialisierung des Gesamtsystems geprüft.

Für Wahrscheinlichkeiten wird der Typ PROB verwendet, der ansonsten nicht für die Funktions- und Variablendeklaration zur Verfügung steht. Wird ein PROB-Wert als Eingangswert verlangt, so ist an dieser Stelle beim Aufruf der Parametertabelle die Standardfunktion RANDOM zu verwenden.

Sollen Werte aus einer Tabelle innerhalb des Simulationsprogramms verwendet werden (z.B. Geschlecht eines Neugeborenen oder eine Wahrscheinlichkeit), so wird folgende Funktion verwendet:

PARAM_VALUE (*tabellen-name, ausdruck* {, *ausdruck*})

Der Tabellenname ist in der PARAMETER-Sektion definiert, die nachfolgenden Ausdrücke entsprechen den aktuellen Parametern einer Funktion.

Zurückgeliefert wird ein Wert von dem Typ, der im Vereinbarungsteil in der PARAMETER-Sektion definiert wurde. Dieser Wert kann in gleicher Weise innerhalb eines Ausdrucks verwendet werden, die der einer normalen Funktion.

Beispiel:

```
neuePerson.Geschlecht := PARAM_VALUE (Kindgeschlecht, RANDOM);
```

Oft sind nicht einzelne Werte, sondern bestimmte Bereiche gegeben. Z.B. könnte eine Angabe lauten: 10% der Personen verdienen zwischen 2000 und 2500 DM. Es wäre nun nicht sinnvoll, 10% der Personen z.B. mit 2250 DM den Wert der Klassenmitte zu vergeben. Statt dessen sollte die Angabe so umgesetzt werden, daß für 10% der Personen ein Wert im angegebenen Intervall vorkommen soll, wobei diese Werte innerhalb des Intervalls gleichverteilt sein sollen. Für diesen Zweck steht die Funktion

PARAM_RANGEVALUE (*tabellen-name, ausdruck* {, *ausdruck*})

zur Verfügung. Die Tabelle muß dazu einen Wert des Typs RANGE liefern, also einem Vektor mit je einem Wert für die untere und obere Intervallgrenze. Aus diesem Bereich bestimmt die Funktion dann automatisch eine Zufallszahl, die gleichverteilt im Intervall liegt. Ist der Typ der Tabelle IRANGE, wird ein INTEGER-Wert im Intervall [a; b] geliefert (also inkl. der Intervallgrenzen), bei RRANGE wird ein REAL-Wert aus dem Intervall (a; b) zurückgegeben.

Handelt es sich bei den Tabellenwerten um Wahrscheinlichkeiten, so dient folgender Zugriff der stochastischen Abfrage, ob ein Ereignis stattfindet oder nicht:

PARAM_EVENT (*tabellen-name, ausdruck* {, *ausdruck*})

Das Ergebnis ist immer ein BOOLEAN-Wert. Es handelt sich dabei im Prinzip um eine Kurzschreibweise für den Ausdruck

```
PARAM_VALUE (...) <= RANDOM,
```

jedoch mit besserem Laufzeitverhalten.

13.3 Gesamtaufbau

Die Parameterdatenbasis besitzt folgenden Grundaufbau:

PARAMETER *name*;
Typen-Definitionen
Tabellen-Definitionen

Die Typendefinition entspricht in ihrem Format exakt der TYPE-Sektion im Simulationsmodell.

Der restliche Teil des Programms besteht aus Tabellendefinitionen. Für jede Tabelle werden dabei zwei Arten von Definitionen benötigt:

Zunächst muß für jede Tabelle eine Gesamtdefinition erfolgen. Diese entspricht weitgehend einer Funktionsvereinbarung und besteht aus einem Kopf mit den formalen Parametern und dem Typ des Rückgabewerts sowie einem Anweisungsblock mit der Abarbeitungsreihenfolge der Subtabellen.

Anschließend folgen die Subtabellen. Je nach Definition haben sie die Aufgabe, die Eingangswerte in Indizes für weitere Tabellenzugriffe umzusetzen, oder sie enthalten die eigentlichen Werte.

Die Reihenfolge, in der die Tabellen und Subtabellen im Programmtext aufgeführt werden, ist frei. Die einzige Einschränkung besteht darin, daß zunächst die Tabellendefinition erfolgen muß, bevor die Subtabellen definiert werden können.

Als mögliche Ordnung können z.B. jeweils die Tabelle und die dazu gehörigen Subtabellen zusammen definiert werden, bevor die nächste Tabelle mit ihren Subtabellen als neue Einheit folgen. Der Vorteil dieser Vorgehensweise liegt darin, daß jede Tabelle als geschlossene Einheit geschrieben und der Zusammenhang aller ihrer Teile deutlich wird. Nachteilig ist die Tatsache, daß es auf diese Art sehr schwer ist, sich einen Überblick über die gesamten Daten zu verschaffen.

Eine andere Möglichkeit besteht darin, zunächst alle Tabellendefinitionen nacheinander an den Beginn zu setzen und erst anschließend die Subtabellen folgen zu lassen. Diese Form hat den Vorteil, daß sich der Leser schnell einen Überblick über die gesamte Parameterdatenbasis verschaffen kann, da die kurzen Tabellendefinitionen praktisch wie ein Inhaltsverzeichnis wirken. Die sehr umfangreichen Tabellen, die zudem für mehrere Jahre vorliegen können, brauchen dann erst bei Bedarf betrachtet zu werden.

13.4 Zugriff innerhalb der Tabellen

In diesem Abschnitt wird anhand von zwei Beispielen gezeigt, wie ein Zugriff innerhalb einer Tabelle inkl. ihrer Subtabellen konkret abläuft. Dabei werden zwei grundlegende Arten des Zugriffs unterschieden.

Bei der ersten Art des Zugriffs arbeitet die Tabelle grundsätzlich wie eine Funktion: Es werden Argumente übergeben, und der dafür vorgesehene Funktionswert wird als Ergebnis zurückgegeben. Dies sei im folgenden am Beispiel einer Tabelle beschrieben, mit der die Wahrscheinlichkeit von Frauen definiert wird, ein Kind (inkl. Mehrlingsgeburt) zu bekommen.

Die Tabellendefinition sieht dabei so aus:

```
TABLE Geburt
    (Alter: INTEGER;
     Familienstand: Familienstaende;
     erwerbstaetig: BOOLEAN): PROB;
  BEGIN
    T1(Alter);
    T2(Familienstand);
    T3(erwerbstaetig);
    T4(T1,T2,T3)
  END;
```

Die Subtabellen sehen für das gewählte Beispiel folgendermaßen aus (Subtabelle T4 verkürzt):

```
SUBTABLE Geburt.T1 (INTEGER): INDEX;
  BEGIN
    16 .. 45
  END;

SUBTABLE Geburt.T2 (Familienstaende): INDEX;
  BEGIN
    ledig:2
    verheiratet:1
    verwitwet:2
    geschieden:2
  END;

SUBTABLE Geburt.T3 (BOOLEAN): INDEX;
  BEGIN
    TRUE   FALSE
  END;

SUBTABLE Geburt.T4 (INDEX,INDEX,INDEX): PROB;
  PERIOD: 1985;
  BEGIN
    ((0.05373115   0.21120866)   (0.00118378   0.00142275))
    ((0.10867247   0.31323114)   (0.00183293   0.00385779))
    ...
    ((0.00070398   0.00150272)   (0.00052733   0.00052864))
    ((0.00036077   0.00086530)   (0.00033030   0.00062219))
  END;
```

Angenommen, es soll jetzt die Wahrscheinlichkeit bestimmt werden, mit der eine 17jährige, ledige, erwerbstätige Frau ein Kind bekommt. Dazu werden die Argumente "Alter=17", "Familienstand=ledig" und "erwerbstaetig=TRUE" an die Tabelle Geburt übergeben. In der Subtabelle Geburt.T1 wird aus dem INTEGER-Wert 17 der Index 2, da 17 innerhalb der Reihe "16 .. 45" an zweiter Stelle steht. Der Familienstand "ledig" wird in der Subtabelle Geburt.T2 in den Index 2 umgesetzt, und der Wert TRUE für die Erwerbstätigkeit ergibt in der Subtabelle Geburt.T3 den Index 1. Mit den ermittelten Indizes (2, 2, 1) wird abschließend auf Subtabelle Geburt.T4 zugegriffen. Dort ergibt sich dafür der PROB-Wert 0.00183293, der als

Ergebnis des gesamten Tabellenzugriffs von der Parameterdatenbasis an das aufrufende MISTRAL-Programm zurückgegeben wird.

Allgemein wird beim Zugriff auf einzelne Werte (z.B. eine Wahrscheinlichkeit) für jedes Argument eine Subtabelle verwendet, um die Ursprungswerte in Indizes für den abschließenden Zugriff auf die letzte Subtabelle mit den eigentlichen Werten umzuwandeln. In der Regel muß nur die letzte Subtabelle für verschiedene Perioden mit den dort aktuellen Werten vorhanden sein; die übrigen Subtabellen können über alle Perioden unverändert bleiben.

Die zweite Art des Zugriffs auf Tabellen ist notwendig, wenn einer von mehreren möglichen Werten mit eine bestimmten Wahrscheinlichkeit ausgewählt werden soll. So muß z.B. für jedes neugeborene Kind in der Simulation das Geschlecht bestimmt werden. Die Wahrscheinlichkeit dafür, daß es sich um einen Jungen handelt, beträgt ca. 51%.

Im Gegensatz zum letzten Beispiel ist hier eines der Argumente (bzw. - wie hier - das einzige Argument) ein Zufallswert vom Typ PROB. Für das Beispiel des Kindgeschlechts sieht die Tabelle inkl. ihrer Subtabellen vollständig z.B. so aus:

```
TABLE KindGeschlecht (r: PROB): Geschlechter;
  BEGIN
    T1(r);
    T2(T1)
  END;

SUBTABLE KindGeschlecht.T1 (PROB): INDEX;
  PERIOD: 1985;
  BEGIN
    0.51195705    1.00000000
  END;

SUBTABLE KindGeschlecht.T2 (INDEX): Geschlechter;
  BEGIN
    maennlich  weiblich
  END;
```

Als Argument wird ein PROB-Wert übergeben, der beispielsweise dem Zufallsgenerator der Mutter entnommen wird. Dieser Wert wird in der Subtabelle KindGeschlecht.T1 in einen Index umgewandelt. Dazu sind dort die Wahrscheinlichkeiten kumuliert eingetragen. Von links beginnend wird der erste Wert in der Subtabelle gesucht, der größer oder gleich dem übergebenen PROB-Wert ist. Wurde z.B. der Wert 0,7 übergeben, so ist der zweite Wert der Tabelle (1.000) der erste, der diese Bedingung erfüllt. Dessen Position - also 2 - ist der Rückgabewert der Subtabelle (Typ INDEX). Mit diesem Index wird anschließend auf die Subtabelle KindGeschlecht.T2 zugegriffen und für das Beispiel (r = 0,7) "weiblich" als Gesamtergebnis der Tabelle KindGeschlecht an das aufrufende MISTRAL-Programm zurückgegeben.

Beide beschriebenen Arten des Zugriffs auf Parametertabellen können auch kombiniert vorkommen. Z.B. hängt die Stellung im Beruf für Berufsanfänger von ihrem Alter ab. 15jährige werden deutlich häufiger als Arbeiter anfangen als 25jährige, die (z.B. als Akademiker) meist eine Position als Angestellte erhalten.

13.5 Format der Tabellen

Nach dem Überblick über den grundlegenden Ablauf im letzten Abschnitt wird nachfolgend das Format mit allen Möglichkeiten beschrieben. Dazu wird erneut das erste Beispiel verwendet:

```
TABLE Geburt
   (Alter: INTEGER;
    Familienstand: Familienstaende;
    erwerbstaetig: BOOLEAN): PROB;
 BEGIN
   T1(Alter);
   T2(Familienstand);
   T3(erwerbstaetig);
   T4(T1,T2,T3)
 END;
```

Das Format des Kopfes ist - abgesehen vom Schlüsselwort TABLE - identisch mit dem einer Funktionsvereinbarung. Der Rückgabewert ist vom Typ PROB (REAL-Wert zwischen 0.0 und 1.0) und hängt von den Eingangswerten Alter, Familienstand und Erwerbstätigkeit ab, die als formale Parameter definiert sind.

Innerhalb des Blocks zwischen BEGIN und END stehen keine normalen Anweisungen, sondern die Aufrufe der Subtabellen. Zu beachten ist, daß hinter dem letzten Subtabellenaufruf kein Semikolon stehen darf. Die Namen der Subtabellen können frei gewählt werden, dürfen jedoch nicht mit den Namen der formalen Parameter übereinstimmen.

Alle Subtabellen außer der letzten liefern Werte vom Spezialtyp INDEX, der nur Ganzzahlwerte größer 0 annehmen kann. Über diese Werte erfolgt der Zugriff auf andere Tabellen. Die letzte Tabelle liefert immer Werte des Typs der Tabelle, in diesem Fall also PROB-Werte.

Für die Eingangswerte von Subtabellen gibt es folgende Möglichkeiten:

1 externer Wert:	In diesen Tabellen werden positive INTEGER-, REAL- und ENUM-Werte in den Typ INDEX umgesetzt, mit dem wiederum ein Zugriff auf andere Tabellen stattfindet. REAL-Werte werden in der aktuellen Implementierung über die Funktion TRUNC in INTEGER-Werte umgewandelt.
1 PROB-Wert:	Die Subtabelle enthält Wahrscheinlichkeiten, deren Summe 1.0 ergibt (eventuell in kumulierter Form). Als Ergebnis wird ein INDEX-Wert zurückgeliefert. Bei gleichverteilten PROB-Werten wird jeder Index mit der angegebenen Wahrscheinlichkeit gewählt.
n INDEX-Werte, 1 PROB-Wert:	Das Verhalten entspricht dem bei "1 PROB-Wert", über die Indizes wird jedoch erst eine entsprechende Wahrscheinlichkeitenliste ausgewählt. Der PROB-Wert muß immer der letzte Parameter sein.
n INDEX-Werte:	Es erfolgt ein direkter Zugriff auf eine n-dimensionale Tabelle. Bei Zwischentabellen, die für Umkodierungen verwendet werden können, wird ein INDEX-Wert gelie-

fert, bei Abschlußtabellen ein Wert vom Typ INTEGER, REAL, PROB, ENUM oder BOOLEAN.

Für die nachfolgenden Ausführungen hier noch einmal die schon oben dargestellten Subtabellen:

```
SUBTABLE Geburt.T1 (INTEGER): INDEX;
 BEGIN
  16 .. 45
 END;

SUBTABLE Geburt.T2 (Familienstaende): INDEX;
 BEGIN
  ledig:2
  verheiratet:1
  verwitwet:2
  geschieden:2
 END;

SUBTABLE Geburt.T3 (BOOLEAN): INDEX;
 BEGIN
  TRUE   FALSE
 END;

SUBTABLE Geburt.T4 (INDEX,INDEX,INDEX): PROB;
 PERIOD: 1985;
 BEGIN
  ((0.05373115   0.21120866)  (0.00118378   0.00142275))
  ((0.10867247   0.31323114)  (0.00183293   0.00385779))
  ...
  ((0.00070398   0.00150272)  (0.00052733   0.00052864))
  ((0.00036077   0.00086530)  (0.00033030   0.00062219))
 END;
```

Nach dem Schlüsselwort SUBTABLE folgen der Name der Tabelle und der Subtabelle, getrennt durch einen Punkt (ähnlich einer Methodenvereinbarung). Für die formalen Parameter sind nur noch - zur Konsistenzprüfung mit der Tabelle - die Typen anzugeben. Anschließend folgt der Typ des Rückgabewertes.

Im Anschluß an den Kopf können - in dieser Reihenfolge - optionale Angaben gemacht werden:

PERIOD: *perioden-wert* Gibt an, ab welcher Periode die Tabelle gelten soll. In der Simulation wird jeweils die Tabelle verwendet, die genau der Periode entspricht oder möglichst dicht davor liegt. Fehlt die Angabe, so wird das Jahr als 0 angenommen. Dies ist immer dann sinnvoll, wenn eine Tabelle für alle Perioden unverändert gelten soll. Die Periodenangabe ist bei jeder Subtabelle möglich, für eine bestimmte Subtabelle (gleicher Name) dürfen jedoch nicht zwei Subtabellen für die gleiche Periode definiert werden.

FACTOR: *zahl* Bei Subtabellen des Ausgangstyps INTEGER, REAL und PROB kann ein Faktor angegebenen werden, mit dem alle Tabellenwerte multipliziert werden. Bei einem Faktor von 0.01 können die PROB-Werte so z.B. in Prozent geschrieben werden.

PROB_MODE: *modus* Diese Angabe ist nur bei Subtabellen möglich, die als Eingangswert einen PROB-Wert erwarten. Ist der Modus "CUMULATED" (Default), so müssen die Wahrscheinlichkeitswerte innerhalb der Subtabelle in kumulierter Form angegeben werden (z.B. "0.3 0.8 1.0"). Beim Modus "NOT_CUMULATED" werden die direkten Wahrscheinlichkeiten angegeben (z.B. "0.3 0.5 0.2").

Der anschließende Block enthält die eigentlichen Tabellenwerte.

Das genaue Format hängt von den Typen der Eingangswerte ab:

INTEGER:

Im einfachsten Fall werden alle relevanten Werte einzeln geschrieben:

18 19 20 21 22

Für den Eingangswert 18 wird in diesem Fall der Index 1, für 19 der Index 2 usw. zurückgegeben.

Werte, die nicht explizit aufgeführt sind (inkl. Lücken innerhalb des angegebenen Bereichs), führen zum Rückgabewert 0. Sollen auch Werten oberhalb oder unterhalb des angegebenen Bereichs einen Index erhalten, so können als erster bzw. letzter Wert offene Randklassen in folgender Weise definiert werden:

<=17 18 19 20 21 22 >=23

Damit erhalten Werte bis einschließlich 17 den Index 1, Werte ab 23 den Index 7.

Sollen Eingangswerte eines Bereichs einen gemeinsamen Index erhalten, so ist dies so zu schreiben:

17 [18,19] 20 [21,25]

Den Werten 18 und 19 wird hier der Index 2 zugewiesen, Zahlen zwischen (einschließlich) 21 und 25 der Index 4.

Zur Verringerung des Schreibaufwands können aufeinanderfolgende Zahlen bzw. Bereiche gleicher Größe mit ".." abgekürzt werden. Davor und dahinter sind die erste und letzte Zahl bzw. der erste und letzte Bereich anzugeben. Beispiel:

18 .. 20 [21,25] .. [36,40]

Diese Kurzschreibweise entspricht folgender ausgeschriebener Angabe:

18 19 20 [21,25] [26,30] [31,35] [36,40]

Wie schon in den Beispielen erkennbar ist, können alle Formen in einer Tabelle kombiniert werden.

ENUM bzw. BOOLEAN:

Im einfachsten Fall werden die gewünschten Ausprägungen nacheinander angegeben:

ledig verheiratet geschieden verwitwet

Hierbei würde für den Eingangswert "geschieden" z.B. der Index 3 geliefert.

Im Beispiel der Tabelle Geburt sollen alle Familienstände, die nicht "verheiratet" sind, auf dieselben Tabelleneinträge referenzieren. Dies kann dadurch geschehen, daß den Ausprägungen - getrennt durch einen Doppelpunkt - der gewünschte Index nachgestellt wird:

ledig:2 verheiratet:1 verwitwet:2 geschieden:2

Somit liefert "verheiratet" den Index 1, alle anderen Ausprägungen den Index 2. Dabei sollten alle Ausprägungen die Indexangabe erhalten. Anderenfalls wird die normale Positions-Nr. vergeben, für "geschieden" also 4, obwohl 3 noch nicht vergeben ist.

PROB:

Wird ein PROB-Wert als Eingangswert erwartet, so werden die Wahrscheinlichkeiten in kumulierter Form z.B. folgendermaßen angegeben:

0.2 0.5 0.9 1.0

Damit erhalten Eingangswerte im Intervall [0.0, 0.2] den Index 1, im Intervall (0.2, 0.5] den Index 2 usw. Bei gleichverteilten Zufallswerten als Input entspricht die Größe des Intervalls somit der Wahrscheinlichkeit für einen bestimmten Index.

Wurde als PROB_MODE in der Subtabelle "NOT_CUMULATED" definiert, wäre diese Liste folgendermaßen anzugeben:

0.2 0.3 0.4 0.1

Man erkennt, daß bei der kumulierten Angabe der letzte Wert immer 1.0 sein muß, bei nicht kumulierten Daten muß die Summe aller Werte 1.0 betragen. Zur Vereinfachung der Eingabe und der besseren Lesbarkeit kann dieser Wert statt 1.0 auch 10^n betragen, wobei gilt: $n \in N_0$. Damit können auch direkt Prozentwerte eingegeben werden.

1 INDEX:

Bei diesen Tabellen bestimmt der Eingangswert exakt die Position in der Tabelle, deren Wert zurückgegeben wird:

ledig verheiratet geschieden verwitwet

Für den Eingangswert 3 wird hier "geschieden" zurückgeliefert.

Solche Subtabellen werden normalerweise als letzte verwendet. Eine Ausnahme bilden nur Zwischentabellen, die dem Umkodieren dienen. In diesem Fall werden Indizes für eine folgende Tabelle als Werte aufgeführt.

n INDEX:

Dieser Fall entspricht weitgehend dem letzten, außer daß ein Zugriff auf eine n-dimensionale Tabelle erfolgt. Die Ebenen der Dimensionen sind durch Klammerung anzugeben. Beispiel:

(1 2 3 4) (5 6 7 8)

Hierbei handelt es sich um eine zweidimensionale Tabelle, die für den Eingangswert (2, 3) den Rückgabewert 7 liefert. Der Abarbeitung der Eingangswerte von links nach rechts entspricht also ein Durchsuchen der Tabelle von außen nach innen.

n INDEX, 1 PROB:

Dieser Fall kombiniert die Fälle "PROB" und "n INDEX". Zunächst wird über die Indizes die richtige Wahrscheinlichkeitsliste ausgewählt, anschließend erfolgt die Auswahl innerhalb dieser Liste wie im Falle "PROB":

(0.1 0.2 0.9 1.0) (0.3 0.5 0.7 1.0)

Der Eingangswert (2, 0.25) liefert also den Index 1 zurück.

Mit diesen Informationen ist das oben abgedruckte Beispiel einfach zu verstehen:

Zunächst werden die Eingangswerte Alter, Familienstand und Erwerbstätigkeit in den ersten drei Subtabellen in Indizes umgewandelt, mit denen dann in der vierten Subtabelle der Zugriff auf den eigentlichen Rückgabewert erfolgt.

Für die konkreten Eingangswerte (17, ledig, FALSE) würde die Tabelle Geburt den Wahrscheinlichkeitswert 0.00385779 zurückgeben. Innerhalb der Mikrosimulation würde also für 0,38% der ledigen, nicht erwerbstätigen 17-jährigen Frauen das Ereignis "Geburt" simulieren.

Liefert eine der Subtabellen den Index 0, so hängt es vom Typ des Rückgabewerts der gesamten Tabelle ab, was geschieht. Ist dieser Typ PROB, so wird der Aufruf der folgenden Subtabellen nicht mehr durchgeführt, sondern als Ergebnis 0.0 zurückgegeben (d.h.: Ereignis findet nicht statt). Dies kann auch dazu genutzt werden, den Schreibaufwand zu verringern, da keine offenen Randbereiche angegeben werden müssen.

Anders verhält es sich, wenn ein Wert vom Typ INTEGER, REAL, BOOLEAN oder ENUM zurückgegeben werden soll. Ist z.B. das Geschlecht eines Neugeborenen zu bestimmen, muß ein definierter Wert ermittelt werden. Anderenfalls liegt ein Fehler vor, den das System nicht beheben kann. Ein solcher Fall führt demnach zu einem Laufzeitfehler des Systems und damit zum Abbruch der Simulation.

Eine Besonderheit stellen Ausgangswerte des Typs **VECTOR** (IVECTOR und RVECTOR) sowie des Untertyps **RANGE** (IRANGE und RRANGE) dar. Anstelle eines einzelnen Wertes vom Typ INTEGER oder REAL wird hier eine Liste mit beliebig vielen bzw. genau zwei Komponenten dieser Typen angegeben. Die Syntax entspricht der bisherigen, d.h., es wird eine Liste von Werte in Klammern gesetzt und mit Leerzeichen o.ä. getrennt. Bei gleicher Dimension der Eingangsgrößen wird hier also eine Klammerebene mehr verwendet. Die Elemente dieser letzten Ebene werden als ein Gesamtvektor übergeben. Ein Zugriff auf einzelne Komponenten innerhalb des Vektors wird erst im Simulationsprogramm vorgenommen. Da die Anzahl der Elemente eines Vektors nicht in der Deklaration festgelegt wird, kann jeder Vektor eine andere Größe besitzen (ähnlich Strings unterschiedlicher Länge trotz gleicher Deklaration).

13.6 Projektion und Interpolation

Geht man davon aus, daß sich bestimmte Werte (z.B. die Lebenserwartung) künftig in eine bestimmte Richtung entwickeln, so könnte man dies durch entsprechend veränderte Tabellen abbilden. Bleibt die grundsätzliche Struktur identisch, könnte man dazu eine vorhandene (Sub-) Tabelle unverändert verwenden und mit einer neuen Periode sowie einem entsprechenden Faktor versehen. Der Nachteil dieser Methode - insbesondere bei Langfristprognosen - besteht darin, daß man für jede Periode (oder zumindest für relativ viele) immer wieder eine Subtabelle kopieren und editieren müßte. Wesentlich einfacher ist die Möglichkeit, dies zen-

tral für das Gesamtergebnis der Tabelle in Form von Punkten aus Periode und Faktor anzugeben, zwischen denen automatisch interpoliert wird.

Als konkretes Beispiel hier eine Parametertabelle, die für die Simulation ab 1985 verwendet wird:

```
TABLE Test (...): REAL;
  FACTOR: 1988=2  1993=2.5;
  BEGIN
    ...
  END;
```

Die Angaben hinter dem Schlüsselwort FACTOR bestehen aus einer Periodenangabe (INTEGER-Zahl) sowie einem Faktor (der natürlich auch kleiner 1,0 sein kann), mit dem die Ergebnisse multipliziert werden.

Bis zur ersten angegebenen Periode bleiben die ermittelten Ergebnisse unverändert. Der Wert für 1988 wird mit 2,0 multipliziert, der für 1993 mit 2,5. Für Perioden zwischen 1988 und 1993 wird der Faktor durch Interpolation zwischen diesen Werten bestimmt. Ab 1993 bleibt der Faktor mit 2,5 konstant. Konkret ergeben sich folgende Werte für die einzelnen Perioden:

Periode:	85	86	87	88	89	90	91	92	93	94
Faktor:	1,0	1,0	1,0	2,0	2,1	2,2	2,3	2,4	2,5	2,5

Möglich ist dieses Verfahren nur bei Tabellen, die REAL, INTEGER oder PROB liefern. Bei Verteilungen (z.B. Berufsquoten) müßten gegenläufige Trends für die einzelnen Ausprägungen definiert werden, die zudem noch der Restriktion unterliegen, daß die Summe 1 betragen muß.

Bei wenigen Ausprägungen läßt sich im Einzelfall das Verfahren durch eine Umformung dennoch verwenden. Z.B. kann die Aufteilung bei Geburten auf Einling, Zwilling und Drilling in der üblich Form als Quoten erfolgen. Alternativ könnte auch die Mehrlingsquote als REAL- bzw. PROB-Wert Ergebnis einer Tabelle sein, der dann in der gezeigten Weise extrapoliert werden kann.

Eine andere Methode, gleitende Übergänge zwischen zwei Parameterwerten herzustellen, ist die Interpolation zwischen zwei Subtabellen. Voraussetzung dafür sind zwei zeitlich aufeinanderfolgende Subtabellen, die ausschließlich numerische Einträge enthalten. Dies gilt für Subtabellen, die entweder Index-Werte in einen numerischen Output (inkl. Vector) umwandeln oder die einen PROB-Input in einen Index umsetzen. Weiterhin müssen die beiden Subtabellen absolut identische Strukturen besitzen, also auch z.B. Vektoren mit jeweils identischer Länge an einander entsprechenden Stellen.

Eingeschaltet wird der Interpolations-Modus mit folgender Angabe:

```
INTERPOLATION: TRUE;
```

Diese muß im Kopf einer Subtabelle als letzte Angabe vor dem BEGIN stehen.

Wirksam wird der Modus dann, wenn die aktuelle Simulationsperiode zwischen den Perioden zweier zeitlich aufeinanderfolgender Subtabellen liegt und die frühere Subtabelle, auf die normalerweise direkt zugegriffen würde, den Interpolations-Modus aktiviert hat.

14 Zeitreihendatenbasis

Die Zeitreihendatenbasis dient dazu, Zeitreihenwerte für die Simulation zur Verfügung zu stellen. Dabei handelt es sich für jede spezifizierte Variable um genau einen REAL-Wert pro Periode.

Die Angabe einer Zeitreihe sieht z.B. so aus:

```
C  'privater Konsum'    1949
    48.300     62.500     67.100     72.600
    81.100     85.900     94.000    101.800
   109.200    113.300    119.000
```

Der erste Bezeichner gibt den Namen einer Zeitreihenvariablen an.

Der optionale String dient einer näheren Kommentierung und kann - im Gegensatz zu echten Kommentaren - auch von Analysewerkzeugen verwendet werden.

Die nächste Angabe bezeichnet die Periode, aus der der erste Wert der Zeitreihe stammt.

Alle nachfolgenden Werte werden als Zeitreihenwerte interpretiert und der jeweils um eins hochgezählten Periode zugewiesen. Im Beispiel ist also 48.3 der Wert für 1949, 62.5 der für 1950. Ist die Anfangsperiode in einer anderen Periodizität angegeben (z.B. 1949Q1), so wird in den entsprechend kleineren Periodenschritten hochgezählt (also z.B. 1949Q1, 1949Q2 usw.). Ist der Wert einer Periode unbekannt, wird anstelle einer Zahl ein "." geschrieben.

Das Ende einer Zeitreihe wird entweder durch den Beginn der nächsten (erkennbar am nicht-numerischen Bezeichner) oder durch das Dateiende markiert.

Innerhalb einer Zeitreihendatenbasis können grundsätzlich Zeitreihen mit unterschiedlichen Periodizitäten verwendet werden. Die vom Simulationsprogramm benötigten Zeitreihen müssen jedoch in derselben wie die Mikrodatenbasis vorliegen.

15 Mikrodatenbasis

Im Gegensatz zu den bisher beschriebenen Programmteilen und Datenbasen wird die Mikrodatenbasis in der Regel nicht manuell, sondern per MISTRAL-Programm erstellt. Um jedoch mögliche Zugriffe für Import und Export der Mikrodaten zu erleichtern, nachfolgend einige grundlegende Informationen zum Format (siehe auch Anhang A.3.4).

Nach einem einleitenden Schlüsselwort und einer Versionsangabe folgt als erster Teil der Datenbasis eine TYPE- und CLASS-Definition, deren Syntax exakt der im MML-Programm entspricht und die inhaltlich dazu passen muß. Dazu muß die der Mikrodatenbasis eine Untermenge der TYPE-Definition des MML-Programms darstellen, wobei jedoch Aliase nicht berücksichtigt werden. Die CLASS-Definition muß ebenfalls eine Untermenge der Definition im Simulationsprogramm darstellen und für alle vorhandenen Klassen mit allen Attributen (bis auf temporäre in der Simulation) übereinstimmen (ausgenommen auch hier Aliase).

Als nächstes folgt ein Teil, in dem Konfigurationsinformationen abgelegt sind. Zur Zeit werden dort die aktuelle Periode sowie ein Startwert (SEED) für den Zufallsgenerator festgelegt.

Anschließend werden die eigentlichen Daten definiert. Dazu wird zunächst der Name einer Klasse angegeben; dahinter folgen in jeweils einer eigenen Zeile die Daten der Objekte dieser Klasse. Die Werte der Attribute werden (beginnen mit den Attributen Key und Factor) durch Semikolon getrennt hintereinander geschrieben. Bei Objektzeigern wird die Schlüssel-Nr. des betreffenden Objekts abgelegt bzw. 0 wenn ein NIL-Zeiger vorliegt. Bei einem SET, das eine beliebige Anzahl von Objekten besitzen kann, werden die Schlüssel-Nrn. durch Komma getrennt angegeben. Der Wert EMPTYSET wird durch eine 0 repräsentiert.

Es können beliebig viele Klassen in einer Mikrodatenbasis enthalten sein. Ebenso ist es möglich, mehrere Abschnitte zu einer Klasse zu verwendet. So können z.B. für jeden Haushalt zunächst die Daten des einen Objekts der Klasse Haushalt anzugeben und anschließend die der Objekte der Klasse Person, die darin enthalten sind. Aus Effizienzgründen (Speicherplatz und Laufzeit) ist es jedoch sinnvoll, alle Objekte einer Klasse direkt hintereinander zu speichern.

Anhang

A.1 Standardprozeduren

ADD (*set, objekt*) Addiert das Objekt zum SET. Befindet sich das Objekt bereits im SET, geschieht nichts.

BREAK Verläßt die aktuelle Schleife (FOR, WHILE, REPEAT, FOREACH). Steht der Befehl außerhalb einer Schleife, wirkt er wie EXIT.

CHECK_MISSING_VALUES Überprüft alle Objekte der Mikrodatenbasis auf Missing-Values und schreibt entsprechende Meldungen in den Report.

CREATEPOINTER Wandelt alle mit READPOINTER erzeugten INTEGER-Zeiger in echte Objektzeiger um.

DELETE (*objekt*) Entfernt das Objekt aus der Mikrodatenbasis. Zugleich wird es aus allen offenen FOREACH-Schleifen entfernt. Offene Methoden des Objekts werden vorzeitig beendet. Es ist wichtig zu beachten, daß damit nur das Objekt aus der zentralen Mikrodatenbasis gelöscht wird. Existieren von anderen Objekten Zeiger auf das Objekt (z.B. Ehepartnerzeiger), so werden diese nicht automatisch gelöscht, sondern zeigen weiterhin auf das gelöschte Objekt. Sie sind durch explizite Zuweisungen auf NIL zu setzen.

DELETEWITHPOINTER (*objekt*) Wie DELETE. Zusätzlich werden alle Pointer auf dieses Objekt auf NIL gesetzt und das Objekt aus allen SETs entfernt.

DISTR_EVENT (*v, f, a1, s1* [, *a2, s2*]) Füllt die Variable *v* des Typs DISTR durch wiederholten Aufruf. Mit dem numerischen Ausdruck *f* wird ein (Gewichtungs-) Faktor angegeben, *a1* (ebenso *a2*) ist der Merkmalswert (BOOLEAN, ENUM oder numerisch), *s1* (ebenso *s2*) ein beschreibender String.

ERROR (*string*) Beendet die Programmausführung. Der String wird in einem Dialogfenster und im Report ausgegeben.

EVENT (*string* [, *number*]) Fügt einen Event unter dem Schlüssel *string* in die interne Eventliste ein. Dabei dient der optionale Parameter *number* als Hochrechnungsfaktor. Als Default wird bei objektgebundenen Methoden der Faktor des aktuellen Objekts verwendet.

EXIT Beendet das Unterprogramm vorzeitig. Steht der Befehl im Hauptprogramm, wird die gesamte Programmausführung beendet.

GETLINE (*file*) Liest die nächste Zeile des Files in den Eingangspuffer ein.

HALT	Beendet sofort die Programmausführung.
REMOVE (*set, objekt*)	Entfernt das Objekt aus der angegebenen Menge. Ist es nicht darin enthalten, geschieht nichts. Befindet sich das Objekt in einer offenen FOREACH-Schleife, wird es dort nicht mehr bearbeitet.
RESET (*file*)	Setzt den Eingangspuffer zurück auf den Beginn des Files.
SETPERIOD (*periode*)	Innerhalb eines Generierungs-Programms kann damit die Periode der Mikrodatenbasis auf den gewünschten Wert gesetzt werden. Dieser ist entweder als INTEGER-Zahl oder - bei unterjährigen Angaben - als String zu schreiben (z.B. '1999Q1').
STATUS ([*string*] [, *real-wert*])	Schreibt den String während der Laufzeit des Programms in die Statuszeile des Mikrosimulationssystems und stellt die Fortschrittsanzeige gemäß dem REAL-Wert (0.0 bis 1.0).
WRITE (*a* [:*b*[:*n*]] {, *a* [:*b*[:*n*]]})	Gibt die Werte der Ausdrücke *a* nacheinander in einer Zeile des Reports aus. Die Breite, auf die mit Leerzeichen aufgefüllt wird, wird mit der INTEGER-Zahl *b* angegeben (rechtsbündig bei $b > 0$, linksbündig bei $b < 0$). Bei REAL-Ausdrücken ist zusätzlich die INTEGER-Zahl *n* für die ausgegebenen Nachkommastellen möglich.
WRITELN (*a* [:*b*[:*n*]] {, *a* [:*b*[:*n*]]})	Wie WRITE, nur daß am Ende ein Zeilenvorschub ausgeführt wird.
WRITEEVENTTABLE	Schreibt eine Tabelle mit Auswertungen zu den zuvor gesammelten Events in den Report.
WRITEOBJECTINFO (*objekt*)	Schreibt den aktuellen Status mit allen Attributen des Objektes in den Report.
WRITETABLE (*d* [, *s*] [, *i*] [,*b1*[,*b2*]])	Gibt eine Tabelle mit Verteilungsinformationen im Report aus. *d* muß ein Ausdruck vom Typ DISTR sein. Der String *s* definiert eine Überschrift. Die Integer-Zahl *i* gibt an, wie viele Zeichen für die Breite der Tabelle zur Verfügung stehen. Der BOOLEAN-Wert *b1* gibt an, ob die Daten gewichtet (Default) ausgegeben werden; *b2* legt fest, ob relative Angaben erfolgen (Default: absolute).

A.2 Standardfunktionen

ABS (*x*) Betrag von *x*.

BTOI (*b*) Wandelt BOOLEAN-Werte (incl. BOOLEAN3) in INTEGER-Zahlen um. TRUE ⇒ 1, sonst ⇒ 0. Kann z.B. in Gleichungen mit qualitativen Größen (LOGIT-Modelle) verwendet werden.

CONCAT (*s1, s2* {, *s3***})** Liefert die Verknüpfung der aufgeführten String-Ausdrücke.

COPY (*objekt*) Gibt eine Kopie des Objekts zurück, die für alle Attribute den gleichen Wert besitzt. Objektzeiger der Kopie erhalten den Wert NIL.

COPYSTR (*s, i1* [, *i2***])** Gibt einen Teilstring von *s* beginnend mit der Position *i1* bis zur Position *i2* zurück. Fehlt *i2*, wird der String bis zum Ende angegeben. Ist *i1* größer als die Länge des Strings, wird ein leerer String zurückgegeben. Die Position des ersten Zeichens ist 1.

COS (*x*) Cosinus von *x*.

DATE Gibt einen String mit dem aktuellen Datum zurück (englisches Format: 'mmm dd, yyyy').

END_OF_FILE (*file*) Gibt TRUE zurück, wenn das Ende der angegebenen Datei erreicht wurde, d.h., der Puffer hinter die letzte Zeile zeigt.

EVENTCOUNT (*string*) Gibt einen INTEGER-Wert mit der Anzahl der Events des angegebenen Strings zurück.

EVENTWCOUNT (*string*) Gibt einen REAL-Wert mit der mit Hochrechnungsfaktor gewichteten Anzahl der Events des angegebenen Strings zurück.

EXP (*x*) Exponential-Funktion e^x

FIRST (*set*) Liefert das Element der Menge mit der niedrigsten Schlüssel-Nr. Ist die Menge leer, wird NIL zurückgegeben.

FRAC (*x*) gebrochener Teil von *x* (= *x* - INT(*x*))

GETCLASS (*klasse*) Gibt ein SET mit allen Objekten der Klasse zurück.

GETOBJECT (*klasse, key*) Gibt einen Zeiger auf das Objekt der angegebenen Klasse und Objekt-Nr (KEY) zurück. Wenn nicht vorhanden, wird NIL zurückgegeben.

INT (*x*) Ganzzahliger Anteil von *x*. Ergebnis ist vom Typ REAL.

LENGTH (*string*) Länge eines String-Ausdrucks.

LN (*x*) Natürlicher Logarithmus von *x* (zur Basis e).

LOG (*x*)	Logarithmus zur Basis 10.
MAX (*1*, *y*)	Maximum von *x* und *y*.
MIN (*x*, *y*)	Minimum von *x* und *y*.
MONTH	Gibt einen INTEGER-Wert mit dem Monat der Periode der aktuellen Mikrodatenbasis zurück. Ist deren Periodendefinition größer, wird 0 zurückgegeben.
MS	Gibt einen Wert zurück, der den Millisekunden seit Mitternacht entspricht. Die Differenz zweier Werte kann zur Messung von Programmlaufzeiten verwendet werden.
NEW (*klasse*)	Erzeugt ein neues Objekt der angegebenen Klasse und gibt es zurück.
NO_OF_FIELDS (*file*, *string*)	Gibt an, wie viele Felder (getrennt am Separator *string*) die aktuelle Zeile der Datei hat.
ODD (*x*)	Gibt TRUE zurück, wenn der INTEGER-Wert *x* gerade ist, sonst FALSE.
PARAM_EVENT (*p*, *a* {, *a*})	Greift bei einer Parametertabelle *p* vom Typ PROB auf den zu den Argumenten gehörenden Wahrscheinlichkeitswert zu, berechnet eine Zufallszahl und gibt TRUE zurück, wenn der Zufallswert kleiner als die Wahrscheinlichkeit ist (sonst FALSE).
PARAM_RANGEVALUE (*p*, *a* {, *a*})	Liefert bei einer Parametertabelle *p* vom Typ IRANGE oder RRANGE gleichverteilt einen zufälligen Wert vom Typ INTEGER bzw. REAL aus diesem Bereich.
PARAM_VALUE (*p*, *a* {, *a*})	Gibt den Wert der Parametertabelle *p* zurück, der sich für die angegebenen Argumente ergibt.
PERIOD	Gibt einen String mit der Periode der aktuellen Mikrodatenbasis zurück.
POSITION_IN_FILE (*file*)	Gibt eine REAL-Zahl zwischen 0.0 und 1.0 zurück, welche der aktuellen Position des Puffers in der Datei entspricht.
QUARTER	Gibt einen INTEGER-Wert mit dem Quartal der Periode der aktuellen Mikrodatenbasis zurück. Ist deren Periodendefinition größer, wird 0 zurückgegeben.
RANDOM (*objekt*)	Gibt eine Zufallszahl im Intervall (0; 1) vom Typ PROB zurück. Innerhalb einer Methode oder einem daraus aufgerufenen nicht objektgebundenen Unterprogramm wird der Zufallsgenerator des aktuellen Objekts verwendet, sonst der der Mikrodatenbasis.

Alternativ kann auch explizit ein Objekt angegeben werden, dessen Zufallsgenerator zu verwenden ist.

RANDOMEXP (*x*) Gibt eine exponentialverteilte Zufallszahl mit dem Mittelwert 1/*x* zurück. Innerhalb einer Methode oder einem daraus aufgerufenen nicht objektgebundenen Unterprogramm wird der Zufallsgenerator des aktuellen Objekts verwendet, sonst der der Mikrodatenbasis.

RANDOMOBJECT (*set*) Gibt ein zufällig ausgewähltes Objekt des SETs zurück. Ist die Menge leer, wird NIL zurückgegeben.

RANDOMNORM (*my, s*) Gibt eine normalverteilte Zufallszahl mit dem Erwartungswert *my* und der Standardabweichung *s* zurück. Innerhalb einer Methode oder einem daraus aufgerufenen nicht objektgebundenen Unterprogramm wird der Zufallsgenerator des aktuellen Objekts verwendet, sonst der der Mikrodatenbasis.

RANDOMUNIFORM (*a, b*) Gibt eine gleichverteilte Zufallszahl zurück. Sind *a* und *b* INTEGER-Werte, ist das Ergebnis eine INTEGER-Zahl im Intervall [*a*; *b*], ansonsten eine REAL-Zahl im Intervall (*a*; *b*). Innerhalb einer Methode oder einem daraus aufgerufenen nicht objektgebundenen Unterprogramm wird der Zufallsgenerator des aktuellen Objekts verwendet, sonst der der Mikrodatenbasis.

READINTEGER (*f* [,*s*] ,*n1* [,*n2*]) Liest einen Teilstring der aktuellen Zeile der Datei ein (siehe auch READSTRING) und wandelt ihn in eine INTEGER-Zahl um. Bei Positionen außerhalb der Zeile oder einem nicht passenden Text an der angegebenen Stelle wird - je nach Konfiguration des Systems - 0 zurückgegeben und/oder eine Fehlermeldung in den Report geschrieben.

READLINE (f) Gibt die aktuelle Zeile der Datei als String zurück.

READPOINTER (*f* [,*s*] ,*n1* [,*n2*]) Liest einen Teilstring der aktuellen Zeile der Datei ein (siehe auch READINTEGER) und wandelt ihn in eine INTEGER-Zahl um, die einen Objektzeiger repräsentiert. Dieser kann zunächst einer Zeigervariablen zugewiesen werden, muß jedoch vor Gebrauch mit CREATEPOINTER in einen echten Zeiger umgewandelt werden.

READREAL (*f* [,*s*] ,*n1* [,*n2*]) Liest einen Teilstring der aktuellen Zeile der Datei ein (siehe auch READSTRING) und wandelt ihn in eine REAL-Zahl um. Bei Positionen außerhalb der Zeile oder einem nicht passenden Text an der angegebenen Stelle wird - je nach Konfiguration des Sy-

	stems - 0.0 zurückgegeben und/oder eine Fehlermeldung in den Report geschrieben.
READSTRING (*f* [,*s*] ,*n1* [,*n2*])	Gibt einen Teilstring der aktuellen Zeile der Datei zurück. Bei Angabe von (*f, s, n1*) wird das *n1*-te Feld - getrennt durch den String-Separator *s* gewählt; bei (*f, n1, n2*) der String, der von einschließlich Spalte *n1* bis *n2* reicht. Fehlt *n2*, wird der String bis zum Zeilenende eingelesen. Bei Positionen außerhalb der Zeile wird ein Leerstring zurückgegeben.
ROUND (*x*)	Gibt den auf die nächste ganze Zahl gerundeten Wert von *x* zurück.
SAMPLE (*set, n*)	Gibt eine zufällige Teilmenge (Stichprobe) des SETs zurück. Ist *n* ein INTEGER-Wert, enthält diese die angegebene Anzahl von Objekten. Ist *n* ein REAL-Wert zwischen 0.0 und 1.0, wird ein entsprechender Anteil der Objekte als SET zurückgegeben. Das ursprüngliche SET bleibt dabei unverändert.
SEMIYEAR	Gibt einen INTEGER-Wert mit dem Halbjahr der Periode der aktuellen Mikrodatenbasis zurück. Ist deren Periodendefinition größer, wird 0 zurückgegeben.
SIN (*x*)	Sinus von *x*.
SIZEOF (*set*)	Gibt die Anzahl der Elemente zurück, die im SET enthalten sind.
SQR (*x*)	Quadrat von *x*.
SQRT (*x*)	Quadratwurzel von *x* (immer REAL).
STR (*ausdruck* [:*b*[:*n*]])	Gibt einen String zurück, der den angegebenen Ausdruck repräsentiert. Die optionalen Formatangaben entsprechen denen beim WRITE-Befehl.
TAN (*x*)	Tangens von *x*.
TIME	Gibt einen String mit der aktuellen Uhrzeit zurück ('*hh*:*mm*:*ss*').
TRUNC (*x*)	Ganzzahliger Anteil von *x*. Ergebnis ist vom Typ INTEGER.
YEAR	Gibt einen INTEGER-Wert mit dem Jahr der Periode der aktuellen Mikrodatenbasis zurück.

A.3 Syntax-Definition

Zur Definition der Syntax wird hier die Erweiterte Backus-Naur-Form (EBNF) verwendet.

A.3.1 Micro Modelling Language

Hier wird die Gesamt-Syntax angegeben, von der in den verschiedenen Teilsprachen jedoch jeweils einzelne Teile fehlen können.

program = "PROGRAM" *ident* ";"
definition
block ".".

definition = [*type-def*]
[*class-def*]
[*interface-def*]
[*file-def*]
[*parameter-def*]
[*const-def*]
[*var-def*]
[*routine-def*].

type-def = "TYPE"
{*ident* "=" ["ORDINAL"]
["DOES_NOT_APPLY" ","]
ident ["(" *ident* {"," *ident*} ")"]
{"," *ident* ["(" *ident* {"," *ident*} ")"]} ";"}

class-def = "CLASS"
{*ident* "=" *ident* ["(" *ident* {"," *ident*} ")"] ":" *var-type*
{"," *ident* ["(" *ident* {"," *ident*} ")"] ":" *var-type* }
["," "TEMPORARY"
ident ["(" *ident* {"," *ident*} ")"] ":" *var-type*
{"," *ident* ["(" *ident* {"," *ident*} ")"] ":" *var-type* }] ";"}.

interface-def = "INTERFACE"
["INPUT" *ident* {"," *ident*} ";"]
["OUTPUT" *ident* {"," *ident*} ";"]
["DISTRIBUTION" *ident* {"," *ident*} ";"].

file-def = "FILE" *ident* ["(" *string* ")"]
{"," {*ident* ["(" *string* ")"] } ";".

parameter-def = "PARAMETER" {*ident formal-param* ":" *var-type* ";"}

const-def = "CONST" {*ident* "=" (["+"|"-"] *number* | *boolean* | *ident*) ";"}.

var-def = "VAR" *ident* {"," *ident*} ":" *var-type* ";")
{*ident* {"," *ident*} ":" *var-type* ";"}.

routine-def = {*function-def* | *procedure-def* | *modul-def* | *supermodul-def* }

function-def = "FUNCTION" *qualident* [*formal-param*] ":" *var-type* ";"
[*const-def*] [*var-def*] [*routine-def*] *block* ";".

procedure-def = "PROCEDURE" *qualident* [*formal-param*] ";"
[*const-def*] [*var-def*] [*routine-def*] *block* ";".

modul-def = "MODULE" *qualident* [*formal-param*] ";"
[*const-def*] [*var-def*] [*routine-def*] block ";".

supermodul-def = "SUPERMODULE" *qualident* [*formal-param*] ";"
[*const-def*] [*var-def*] [*routine-def*] *block* ";".

formal-param = "(" *ident* {"," *ident*} ":" *var-type*
{";" *ident* {"," *ident*} ":" *var-type* } ")".

var-type = *ident* | "SET" "OF" *ident*.

block = "BEGIN" *statement* {";" *statement*} "END".

statement = [*qualident* ":=" *expression* |
procedure-call |
"BEGIN" *statement* {";" *statement*} "END" |
"BEGIN_RANDOM" *statement* {";" *statement*} "END" |
"IF" *expression* "THEN" *statement* ["ELSE" *statement*] |
"CASE" *expression* "OF" *case* {";" *case*}
["ELSE" *statement* {";" *statement*} "END" |
"FOR" *ident* ":=" *expression* "TO" *expression* "DO" *statement* |
"WHILE" *expression* "DO" *statement* |
"REPEAT" *statement* {";" *statement*} "UNTIL" *expression* |
"FOREACH" *ident* "IN" *qualident* ("DO"|"DO_RANDOM") *statement*].

procedure-call = *qualident* [*actual-param*].

actual-param = "(" *expression* {"," *expression*} ")".

case = *constant* [".." *constant*] {"," *constant* [".." *constant*]} ":" *statement*.

constant = *ident* | *number* | *boolean* | "DOES_NOT_APPLY".

expression = *complex-expr* {"|" *mql-expression*}.

mql-expression = "COLLECT" *actual-param* |
"SELECT" *actual-param* |
"SELECT_MV" |
"MINIMUM" *actual-param* |
"MAXIMUM" *actual-param* |
"SUM" *actual-param* |
"WSUM" *actual-param* |
"AVERAGE" *actual-param* |
"WAVERAGE" *actual-param* |
"DISTRIBUTION" *actual-param* |
"SIZE" |
"WSIZE".

complex-expr = *simple-expr* ["="|"<>"|">"|">="|"<"|"<="|"IN") *simple-expr*].

simple-expr = *term* {("+"|"-"|"OR") *term*}.

term = *signed-factor* {("*"|"/"|"DIV"|"MOD"|"AND") *signed-factor*}.

signed-factor = ["-" | "+"] *factor*.

factor = *qualident* | *function-call* | "(" *expression* ")" | "NOT" *factor* | *number* |
boolean | *string* | "DOES_NOT_APPLY" | "MISSING_VALUE" |
"NIL" | "EMPTYSET".

function-call = *qualident* [*actual-param*].

qualident = *ident* {"." *ident*}.

ident = *letter* {*letter* | *digit*}.

boolean = "TRUE" | "FALSE".

number = *unsigned-integer* | *unsigned-real*.

unsigned-integer = *digit* {*digit*} ["E" ["+"] *digit* {*digit*}].

unsigned-real = *digit* {*digit*} "." *digit* {*digit*} ["E" ["+"|"-"] *digit* {*digit*}].

digit = "1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"|"0".

letter = "A"|...|"Z"|"a"|...|"z"|"_".

string = """ {*character* | """ """} """.

A.3.2 Parameterdatenbasis

In diesem Abschnitt wird die Syntax der Parameterdatenbasis beschrieben. Um Wiederholungen zu vermeiden, werden dabei Definitionen aus A.3.1 vorausgesetzt.

program = "PARAMETER" *ident* ";"
[*type-def*]
tables.

tables = {*table-def* | *subtable-def*}.

table-def = "TABLE" *ident formal-param* ":" *var-type* ";"
["FACTOR" ":" *unsigned-integer* "=" *number*
{*unsigned-integer* "=" *number*} ";"]
"BEGIN"
ident "(" *ident* {"," *ident*} ")"
{";" *ident* "(" *ident* {"," *ident*} ")"}
"END" ";".

subtable-def = "SUBTABLE" *ident* "." *ident*
"(" *var-type* {"," *var-type*} ")" ":" *var-type* ";"
["PERIOD" ":" *period* ";"]
["FACTOR" ":" *number* ";"]
["INTERPOLATION" ":" *boolean* ";"]
["PROB_MODE" ":" ("CUMULATED" | "NOT_CUMULATED") ";"]
"BEGIN"
(*enum-to-index* | *number-to-index* | *prob-to-index* | *index-to-output*)
"END" ";".

enum-to-index = {*ident* [":" *unsigned-integer*]}.

number-to-index = ["<=" ["-"] *unsigned-integer*]
{["-"] *unsigned-integer* |
".." |
"[" ["-"] *unsigned-integer* "," ["-"] *unsigned-integer* "]"}
[">="["-"] *unsigned-integer*].

prob-to-index = *pti-entry* {*pti-entry*}.

pti-entry = *number* | "(" *prob-to-index* ")".

index-to-output = *list-item* {*list-item*}.

list-item = *number* | *ident* | *list* | "DOES_NOT_APPLY".

list = "(" *list-item* {*list-item*} ")".

period = *number* [("S" | "Q" | "M") *number*].

A.3.3 Zeitreihendatenbasis

In diesem Abschnitt wird die Syntax der Zeitreihendatenbasis beschrieben. Um Wiederholungen zu vermeiden, werden dabei Definitionen aus A.3.1 und A.3.2 vorausgesetzt.

program = "TIME_SERIES" *ident* ";"
time-serie
{*time-serie*}.

time-serie = *ident* [*string*] *period number* {*period-value*}.

period-value = *number* | ".".

A.3.4 Mikrodatenbasis

In diesem Abschnitt wird die Syntax der Mikrodatenbasis beschrieben. Um Wiederholungen zu vermeiden, werden dabei Definitionen aus A.3.1 vorausgesetzt.

program = "MICRODATABASE"
"1.1"
[*type-def*]
[*class-def*]
config-options
data.

config-options = "CONFIGURATION"
{*ident* "=" (*string* | *number* | *period*)}
"END".

data = {*class-name object-data* {*object-data*}}.

object-data = *unsigned-integer* ";" *unsigned-real* {";" *attribute-data*} <Zeilenvorschub>.

attribute-data = *number* | *string* | *object-list*.

object-list = "0" | *unsigned-integer* {"," *unsigned-integer* }.

Für die Konfiguration werden zur Zeit folgende Angaben verwendet (die Werte rechts vom Gleichheitszeichen sind Beispiele):

```
CONFIGURATION
   PERIOD = 1984
   SEED = 10000
END
```

Die ersten beiden Einträge zu jedem Mikroobjekt (unter *object-data*) entsprechen der Schlüssel-Nr. (Attribut *Key*) und dem Hochrechnungsfaktor (Attribut *Factor*).

A.4 MISTRAL-Schlüsselwörter

analysis	and	average	begin	begin_random
case	class	collect	configuration	const
distribution	div	do	does_not_apply	downto
do_random	else	emptyset	end	exit
file	for	foreach	function	generation
if	in	index	input	interface
maximum	minimum	missing_value	mod	module
nil	nominal	not	of	or
ordinal	output	param	parameter	procedure
program	repeat	select	select_mv	self
set	simulation	size	subtable	sum
supermodule	table	temporary	test	then
time_series	to	type	until	var
waverage	while	wsize	wsum	

A.5 MISTRAL-Standardbezeichner

Zu den Standardbezeichnern gehören neben den Namen der Standardfunktionen und -prozeduren folgende:

boolean	boolean3	distr	integer	irange
ivector	posinteger	posinteger_dna	posreal	posreal_dna
prob	real	rrange	rvector	string

A.6 Implementierungsspezifische Details

Die Beschreibung der Sprache MISTRAL bezieht sich zunächst auf die konkrete Implementierung, die zur Zeit verfügbar ist. Die hierbei verwendete Sprache SMALLTALK weist an vielen Stellen deutlich weniger Beschränkungen auf, als dies für übliche Compiler z.B. für PASCAL gilt.

An dieser Stelle soll deshalb - auch für Programmierer von MISTRAL-Compilern auf anderen Plattformen - eine Auflistung von Punkten vorgenommen werden, die bei künftigen Implementierungen zu beachten sind.

Wertebereiche bestimmter Typen:

INTEGER SMALLTALK - und damit MISTRAL - können INTEGER-Werte in bis zu 64KB (!) großen Strukturen speichern, so daß praktisch keine Einschränkung des Wertebereichs existiert. Künftige Implementierungen könnten den Wertebereich beschränken. Jedoch sollte mindestens der Bereich bis 10^{12} abgedeckt werden, um auch aggregierte Werte der gesamten Volkswirtschaft als Summe der einzelnen INTEGER-Attribute abbilden zu können.

STRING Strings können in MISTRAL beliebig lang sein und sich über mehrere Zeilen erstrecken. Hier ist eine Beschränkung auf die in PASCAL üblichen ca. 255 Zeichen in nur einer Zeile sinnvoll.

SET Anders als in PASCAL darf die maximale Anzahl von Elementen einer Menge keinen Beschränkungen unterliegen. Dies ergibt sich zwingend daraus, daß reale Mikrodatenbasen zehn- oder gar hunderttausende Objekte (z.B. Personen) enthalten können.

Namen:

In MISTRAL können Namen für Variablen, Konstanten, Unterprogramme usw. beliebig lang sein und sind auch in voller Länge signifikant. Hier könnten PASCAL-übliche Beschränkungen eingeführt werden, z.B. max. 127 Zeichen, davon die ersten 63 signifikant.

A.7 Hinweise zur Programmierung in MISTRAL

In diesem Abschnitt werden einige Hinweise und Tips zur Programmierung in MISTRAL gegeben.

Da Simulationen relativ hohe Laufzeiten aufweisen können, ist besonderes Augenmerk auf eine diesbezüglich effiziente Programmierung zu legen. So sind - wie in allen Programmiersprachen - nach Möglichkeit alle Anweisungen außerhalb einer Schleife zu plazieren, die nicht unbedingt in ihr stehen müssen. Speziell für MISTRAL gilt bezüglich der Laufzeit:

- Wird ein Objekt mit einem bestimmen Key gesucht, sollte nur GETOBJECT(*key*) verwendet werden.
- Die Anweisung FOREACH ... DO_RANDOM ist insbesondere bei größeren Objektmengen wesentlich langsamer als die Version ohne RANDOM.
- RANDOMOBJECT(*set*) ist wesentlich schneller als alle anderen Konstruktionen, um ein zufälliges Objekt auszuwählen.
- Abfragen innerhalb der MQL-Anweisung SELECT laufen deutlich schneller ab, als Konstruktionen der Art "FOREACH P IN ... DO IF (P = ...) THEN ...".
- Mehrfache Boolesche Ausdrücke (mit AND und OR) werden von links nach rechts nur solange ausgewertet, bis das Ergebnis feststeht. Bei AND sollte deshalb der Teilausdruck mit der größeren Wahrscheinlichkeit für FALSE zuerst angegeben werden, bei OR umgekehrt. Zudem läßt sich durch entsprechende Anordnung auch teilweise eine vorherige IF-Anweisung sparen, mit der nur unzulässige Zugriffe abgefangen werden sollen.
- Es ist zwar sinnvoll, den Programmfortschritt mit "STATUS()" anzuzeigen (sofern das aufgrund des Programms nicht ohnehin automatisch geschieht). Da aber gerade solche Bildschirmausgaben (in allen Systemen) viel Zeit benötigen, sollte man keinesfalls bei einer Schleife über 16.000 Personen den Status nach jeder einzelnen aktualisieren. Das kann die Laufzeit um ein Vielfaches verlängern. Sinnvoll ist es, statt dessen eine Variable hochzuzählen und z.B. alle 100 Durchläufe (leicht mit "IF (n MOD 100 = 0) ..." zu realisieren) eine Ausgabe zu erzeugen. Auch intensive Ausgaben (Traces) in Reports mit WRITELN sind grundsätzlich laufzeitintensiv.

Ein weiterer Punkt zum Thema Effizienz ist der benötigte Speicherplatz. Da sich aufgrund des internen Speicherkonzeptes der zugrundeliegenden Programmiersprache Smalltalk der Speicherbedarf nicht exakt angeben läßt, hier eine grobe Abschätzung einiger wichtiger Eckwerte:

- Jedes Objekt belegt allein durch seine Existenz sowie seine impliziten Attribute (Key, Hochrechnungsfaktor, Zufallswert) in der Mikrodatenbasis ca. 50 Byte.
- Jedes zusätzliche einfache Attribut belegt weitere 4 - 6 Byte.
- Ein SET als Attribut belegt bis 5 Elemente ca. 60 Byte (zusätzlich zum Speicherbedarf für die darin enthaltenen Elemente der Mikrodatenbasis). Bei mehr vorhandenen Elementen erhöht sich der Speicherbedarf weiter.

Werden in der Test-Phase innerhalb eines Programms viele Variablen für Werte bzw. Zähler benötigt, kann es sinnvoll sein, diese als Events abzuspeichern. Nach "EVENT(*string*, *wert*)" ist *wert* ohne die Notwendigkeit einer Variablendefinition mit "VAR" jederzeit mit "EVENTWCOUNT (*string*)" abrufbar und zudem automatisch in der Ergebnis-Zeitreihendatenbasis enthalten.

A.8 Änderungen zu früheren Versionen von MISTRAL

Änderungen der Version 3.0 gegenüber Version 2.0

Umbenennungen:

- Die Standard-Prozedur SETPERIODE wurde in SETPERIOD umbenannt.
- Die Standard-Funktion PERIODE wurde in PERIOD umbenannt.
- Anstelle des Schlüsselwortes RULE_CHECK muß jetzt ein Test-Programm mit TEST eingeleitet werden.

Die alten Bezeichnungen werden noch einige Zeit lang funktionstüchtig bleiben, sollten aber mittelfristig umgestellt werden.

Neue Befehle o.ä.:

- DISTR_EVENT
- RESET
- Modus INTERPOLATION bei Subtabellen in der Parameterdatenbasis

Ergänzungen bzw. Änderungen bei vorhandenen Befehlen:

- DELETEWITHPOINTER löscht jetzt auch Zeiger auf das zu löschende Objekt, die sich in lokalen oder globalen Variablen (inkl. SET) befinden.
- STR erlaubt jetzt auch Format-Anweisungen (wie WRITE)

Sonstiges:

- Innerhalb von Prozeduren können jetzt auch Super-Module und Module definiert werden.
- INCLUDE kann jetzt geschachtelt werden.
- Es können jetzt bis zu 15 Dateien im Definitionsteil FILE (im Generierungs-Programm) verwendet werden. Die Pfadangaben können jetzt auch relativ zum Verzeichnis der obersten Programmdatei erfolgen.

Index