UDML – Universelle Deterministische Maschinensprache

Struktur, Syntax, Semantik und Ausführungsmodell

Contents

Präambel

Die Universelle Deterministische Maschinensprache (UDML) definiert eine formale, transparent überprüfbare und vollständig deterministische Beschreibungsebene für Systeme, die strukturierte Zustände, eindeutige Operatoren und reproduzierbare Übergänge benötigen. Sie bildet eine neutrale, deklarative Grundlage für technische Abläufe, ohne selbst eine bestimmte Architektur, Technologie oder Interpretationsmaschine vorzugeben.

Die UDML verfolgt das Ziel, Abläufe und Informationsstrukturen so darzustellen, dass sie durch beliebige deterministische Ausführungsmechanismen eindeutig interpretierbar bleiben. Sie dient somit als formale Sprache der Klarheit, der Nachvollziehbarkeit und der auditierbaren Reproduzierbarkeit von Systemlogik.

UDML versteht sich nicht als Programmiersprache im klassischen Sinn. Sie legt keine algorithmischen Verfahren fest, sondern stellt eine strukturierte Beschreibungsebene bereit, auf der Systeme eindeutig definieren können, *was* ausgeführt werden soll und *unter welchen Bedingungen* Zustandsübergänge stattfinden. Der Schwerpunkt liegt auf formaler Ordnung, Determinismus, expliziter Struktur und konsequenter Eliminierung impliziter Bedeutungen.

0.1 Zweck der Sprache

Die UDML dient der Beschreibung deterministischer Ausführungsabläufe in einer klar strukturierten, blockorientierten Form. Jede deklarierte Komponente besitzt eine wohldefinierte Bedeutung, jede Veränderung eines Zustands ist explizit sichtbar und jede Ausführung ist vollständig rekonstruierbar.

Zentrale Ziele:

- Herstellung deterministischer Reproduzierbarkeit technischer Abläufe.
- Formale Trennung zwischen Beschreibung (Deklaration) und tatsächlicher Ausführung.
- Vollständige Sichtbarkeit aller Zustandsänderungen.
- Eliminierung impliziter Annahmen zugunsten expliziter Strukturen.
- Bereitstellung einer Sprache, die auditierbare Systemlogik ermöglicht.

Die Sprache ermöglicht somit den Aufbau von Systemen, deren Verhalten nachweisbar und dauerhaft stabil bleibt, selbst wenn sie in komplexe oder dynamische Umgebungen eingebettet werden.

0.2 Nicht-Ziele und Abgrenzungen

UDML ist kein Algorithmus, keine Entscheidungslogik und kein Kontrollsystem. Sie spezifiziert keine technischen Verfahren, keine Modelle, keine Optimierer und keine Architekturprinzipien, die über die Beschreibungsebene hinausgehen.

Die UDML:

- führt keine Berechnungen aus,
- trifft keine Entscheidungen,
- optimiert kein Verhalten,
- erzwingt keine Sicherheitslogik,
- und bindet sich an keine spezifische Technologie.

Sie stellt ausschließlich die deklarative Form bereit, in der Systeme ihre Abläufe beschreiben können. Jede tatsächliche Bewertung, Ausführung oder Interpretation liegt außerhalb der Sprache und wird durch externe Mechanismen realisiert.

0.3 Rolle der UDML in technischen Systemen

UDML fungiert als neutraler Kern einer logischen Beschreibungsschicht. Systeme, die deterministisch, transparent und nachvollziehbar arbeiten sollen, können ihre Strukturen, Operatoren und Übergänge in UDML ausdrücken.

Die Sprache:

- definiert formale Strukturen für Zustände und Operationen,
- stellt einen eindeutigen Bezugsrahmen für Ausführungsmodelle bereit,
- ermöglicht die Abstraktion komplexer Abläufe auf ein prüfbares, deterministisches Schema,
- und trennt die technische Realität eines Systems von seiner deklarativen Darstellung.

Damit wird UDML zu einer *Grundlage*, auf der beliebige deterministische oder hybride Systeme logisch stabilisiert und formalisiert werden können.

0.4 Verhältnis zu interpretierenden oder kontrollierenden Architekturen

UDML ist unabhängig von jeglicher Art von Kontroll-, Ausführungs- oder Bewertungssystem. Sie kann von einfachen deterministischen Engine-Strukturen ebenso interpretiert

werden wie von komplexeren architektonischen Schichten, solange diese die deklarativen Regeln der Sprache respektieren.

Die Sprache:

- definiert, wie eine Struktur aussehen muss,
- aber nicht, wie eine Maschine sie technisch auswertet.

Interpretierende Systeme dürfen:

- Zustände lesen und schreiben, sofern dies in UDML explizit erlaubt ist,
- Operatoren ausführen, sofern sie deterministisch spezifiziert sind,
- und Übergänge realisieren, sofern ihre Bedingungen klar definiert sind.

Damit bildet UDML eine reine Beschreibungsschicht: Sie ist *strukturell bindend*, aber *technisch neutral*. Jede Logik, die auf ihr aufbaut, bleibt austauschbar, solange sie sich an das deklarative Modell hält.

1 Fundament der UDML

Dieses Kapitel beschreibt die konzeptionellen und formalen Grundlagen der Universellen Deterministischen Maschinensprache (UDML). Die hier definierten Prinzipien bilden den logischen Kern der Sprache und bestimmen, wie Zustände, Operatoren und Übergänge strukturiert und interpretiert werden. Ziel dieses Fundaments ist es, eine konsistente, widerspruchsfreie und vollständig deterministische Basis zu schaffen, auf der alle späteren Konstrukte methodisch aufbauen können.

1.1 Ontologischer Rahmen der Sprache

UDML operiert innerhalb eines klar definierten ontologischen Rahmens, der zwischen drei fundamentalen Entitäten unterscheidet:

1. **Strukturelle Einheiten** (z. B. BLOCK, STATE, OPERATOR), welche die formale Struktur eines Systems beschreiben.
2. **Zustandsräume**, die alle explizit deklarierten, veränderbaren Werte eines Systems enthalten.
3. **Deterministische Übergangsregeln**, welche die Transformation des Zustands definieren.

Die Sprache ist so gestaltet, dass jede Entität klar abgegrenzt ist und keine verdeckten Wechselwirkungen existieren. Jede Information wird explizit ausgewiesen, jede Bedeutung formal beschrieben, jeder Zustand vollständig sichtbar gehalten. UDML kennt keine impliziten Mechanismen, keine automatischen Interpretationen und keine kontextabhängige Semantik. Alles, was bedeutungsrelevant ist, muss innerhalb der Sprache explizit definiert werden.

Damit bildet der ontologische Rahmen der UDML einen stabilen, vollständig deterministischen Referenzraum für technische Abläufe.

1.2 Formale Eigenschaften

UDML beruht auf fünf grundlegenden formalen Eigenschaften:

1. **Determinismus**: Bei gleichem Eingangszustand und identischen Operationen entsteht stets derselbe Ausgangszustand. Es gibt keine nicht-deterministischen Elemente.
2. **Reproduzierbarkeit**: Jeder Ausführungspfad ist vollständig rekonstruierbar. Jede Transformation ist nachvollziehbar und ergibt sich ausschließlich aus expliziten Angaben.
3. **Transparenz**: Alle Zustandsveränderungen sind sichtbar. Es existieren keine impliziten Nebenwirkungen oder versteckten Abhängigkeiten.

4. **Abstraktionsebene**: UDML definiert nur die Beschreibungsebene, nicht die technische Umsetzung. Interpreten oder Engines müssen die Sprache respektieren, werden jedoch nicht durch sie festgelegt.

5. **Strikte Syntax-Semantik-Bindung**: Jedes syntaktische Konstrukt besitzt eine klar definierte semantische Wirkung. Es gibt keine offenen Begriffe oder kontextabhängige Bedeutungen.

Diese Eigenschaften gewährleisten, dass UDML als formales, maschinenlesbares und auditierbares System fungiert, unabhängig von der tatsächlichen Umgebung oder technischen Ausführung.

1.3 Abstrakte Maschine

Zentrales Modell der UDML ist eine abstrakte deterministische Maschine. Sie besteht aus folgenden Elementen:

- einem **Zustandsraum**, der alle deklarativ definierten, veränderbaren Werte enthält,
- einem **Operatorraum**, der die Menge aller zulässigen Operationen beschreibt,
- einer **Übergangsfunktion**, welche die Transformation des Zustands gemäß der angewendeten Operatoren festlegt.

Die abstrakte Maschine kennt keine impliziten Steuergrößen. Sie führt nur Operationen aus, die eindeutig spezifiziert sind. Alle Zustandsänderungen sind Ergebnis explizit deklarierter Regeln. Jede Ausführung entspricht einem gerichteten Übergang von einem Zustand in den nächsten:

$$\text{STATE}_{n+1} = \mathcal{T}(\text{STATE}_n, \text{OP})$$

wobei $\mathcal{T}$ die deterministische Übergangsfunktion darstellt. Diese Funktion ist vollständig definiert durch:

1. die Syntax des Operators,
2. seine deklarierte Semantik,
3. den aktuellen Zustand.

Damit existiert eine eindeutige, mathematisch präzise Beschreibung des Verhaltens der Maschine.

1.4 Universeller Operatorraum

Der Operatorraum der UDML umfasst alle formal definierbaren Operationen, die Zustände verändern oder interpretieren können. Jeder Operator besitzt:

- eine eindeutige Bezeichnung,
- eine wohlgeformte Parameterstruktur,
- eine klar definierte semantische Wirkung,
- deterministische Auswirkungen auf den Zustand.

Es existieren keine Operatoren mit impliziten Nebenwirkungen. Jeder Operator ist nur dann gültig, wenn seine Wirkung vollständig und eindeutig deklariert werden kann. Dadurch entsteht ein universeller, erweiterbarer Satz deterministischer Operationstypen, der beliebige logische Konstruktionen abbildbar macht.

1.5 Semantische Invarianten

UDML definiert mehrere Invarianten, die in jeder korrekt formulierten Struktur zwangsläufig gelten müssen:

1. **Keine impliziten Zustände**: Alle existierenden Zustände müssen im STATE-Bereich eines Blocks deklariert sein.
2. **Keine semantischen Lücken**: Jeder syntaktische Ausdruck besitzt eine definierte Bedeutung. Unbestimmte Konstrukte sind ungültig.
3. **Keine verdeckten Übergänge**: Jede Transformation muss explizit durch eine TRANSITION oder einen Operator veranlasst werden.
4. **Kontextunabhängigkeit**: Die Bedeutung eines Konstrukts verändert sich nicht durch äußere Faktoren. UDML hat keine semantische Abhängigkeit von Umgebung oder Laufzeit.
5. **Endliche strukturelle Interpretierbarkeit**: Jede UDML-Struktur muss durch einen deterministischen Parser vollständig interpretierbar sein. Unendliche oder rekursiv unauflösbare Definitionen sind unzulässig.

Diese Invarianten stellen sicher, dass UDML eine vollständig geschlossene, formale und überprüfbare Sprache bleibt, die sich für technische, hochzuverlässige Systeme eignet.

2 Architekturprinzipien

Die UDML folgt einer Reihe grundlegender Architekturprinzipien, die sicherstellen, dass deklarative Strukturen eindeutig, transparent und deterministisch interpretierbar bleiben. Diese Prinzipien bilden das methodische Rückgrat der Sprache. Sie verhindern semantische Unschärfe, verdeckte Mechanismen und unstabile Ausführungsmodelle. Jede spätere Konstruktion der UDML ist an diese Prinzipien gebunden und muss mit ihnen konsistent bleiben.

2.1 Deklaration versus Ausführung

Die UDML trennt strikt zwischen der Beschreibung eines Ablaufs (*Deklaration*) und seiner technischen Realisierung (*Ausführung*). Diese Trennung ist ein zentrales Merkmal der Sprache und gewährleistet:

- Neutralität gegenüber beliebigen Interpretations- oder Ausführungsmaschinen,
- vollständige Auditierbarkeit der deklarierten Struktur,
- reproduzierbare Logik unabhängig von der konkreten technischen Umgebung.

Deklarationen sind vollständig statisch. Sie beschreiben:

- welche Operatoren existieren,
- welche Zustände betroffen sind,
- welche Übergänge erlaubt sind,
- welche Bedingungen mit welcher Bedeutung verknüpft sind.

Die Ausführung hingegen ist lediglich ein *externer Prozess*, der diese Deklaration liest und deterministisch interpretiert. Die Semantik der UDML steht damit unabhängig von jedem Umsetzungskontext.

2.2 Blockorientierung als Ausführungsmodell

Alle deklarativen Strukturen der UDML basieren auf dem Konzept des *Blocks*. Ein Block ist eine geschlossene logische Einheit, die:

1. eine definierte Eingangs- und Ausgangssituation besitzt,
2. einen lokalen Zustandsraum enthält,
3. eine Menge deterministischer Operationen bereitstellt,

4. und definierte Übergänge zu nachfolgenden Blöcken beschreibt.

Der Block stellt die kleinste vollständige Struktureinheit in UDML dar. Blöcke können miteinander verknüpft werden, jedoch nur auf deterministisch beschriebene Weise. Ein Block darf keine Effekte besitzen, die außerhalb seiner formal deklarierten Bestandteile liegen.

Die blockorientierte Struktur verhindert:

- unkontrollierte globale Seiteneffekte,
- semantisch mehrdeutige Ablaufpfade,
- nicht reproduzierbare Zustandssprünge.

Jeder Block ist somit zugleich ein abgeschlossener Container und ein definierter Baustein größerer Prozesse.

2.3 Deterministische Zustandstransformation

Kern der UDML ist die deterministische Transformation eines Zustands in den nächsten. Jede Veränderung folgt dabei einem explizit definierten Mechanismus:

$$STATE_{n+1} = T(STATE_n, OP)$$

wobei T die deklarierte Übergangsfunktion eines Operators repräsentiert.

Durch diese Festlegung entsteht ein formal kontrolliertes System, das keine überraschenden Ergebnisse produzieren kann. Jede erlaubte Transformation muss:

- syntaktisch gültig sein,
- semantisch eindeutig definiert sein,
- explizit im Block oder Operator beschrieben sein,
- und vollständig sichtbar im resultierenden Zustand erscheinen.

Es existieren keine impliziten oder automatischen Zustandseffekte.

2.4 Verbot impliziter Bedeutung

Ein fundamentales Prinzip der UDML lautet:

Keine Bedeutung ohne Deklaration.

Die Sprache kennt weder:

- semantische Ableitungen aus natürlicher Sprache,
- implizite Standardwerte,
- automatische Interpretationen,
- kontextabhängige Bedeutungsverschiebungen.

Dies gewährleistet vollständige Vorhersagbarkeit und eliminiert Mehrdeutigkeiten. Jeder Ausdruck muss seine Bedeutung explizit mitführen. Alle benötigten Felder, Werte und Bedingungen müssen vollständig deklariert sein, bevor sie zur Ausführung gelangen können.

2.5 Auditierbarkeit und formale Nachvollziehbarkeit

Eine deklarative Sprache, die deterministisch wirken soll, muss ihre gesamte Struktur nachvollziehbar halten. UDML erfüllt diese Anforderung durch:

- vollständige Sichtbarkeit aller Zustände,
- gute Strukturierbarkeit durch konsistente Syntax,
- eindeutige Pfade zwischen initialen und finalen Zuständen,
- explizite Definition jedes Operationsschrittes.

Eine externe Ausführungsinstanz kann daher:

1. jeden Übergang rekonstruieren,
2. jede Operation überprüfen,
3. jeden Zustand vollständig auswerten,
4. jeden Ablauf deterministisch reproduzieren.

Damit erfüllt UDML die Voraussetzungen für technische und regulatorische Auditierbarkeit.

2.6 Mechanismen zur Driftprävention

Da UDML keine dynamischen Bedeutungen, keine statistischen Verfahren und keine selbstmodifizierenden Strukturen enthält, verhindert die Sprache inhärent jede Form semantischer Drift. Dies wird erreicht durch:

- feste Definition aller Operatoren und Felder,
- unveränderliche syntaktische und semantische Regeln,

- strikte Trennung zwischen Deklaration und Ausführungslogik,
- vollständige Sichtbarkeit aller Bedeutungsfelder.

Eine UDML-Struktur kann sich nur ändern, wenn sie explizit neu deklariert wird. Es gibt keine latenten oder unbeabsichtigten Veränderungen. Dadurch eignet sich die Sprache für Anwendungen, in denen langfristige Konsistenz und absolute Verlässlichkeit zwingend sind.

3 Syntax der UDML

Die Syntax der UDML definiert die äußere Form der Sprache. Sie legt fest, wie Strukturen aufgebaut, wie Elemente benannt und wie logische Beziehungen ausgedrückt werden. Die Syntax ist vollständig deterministisch gestaltet: Jedes Element besitzt eine eindeutige grammatische Stellung, jede Kombination folgt festen Regeln, und alle zulässigen Formen sind durch klar spezifizierte Produktionensregeln beschreibbar.

Die Syntax bildet die Basis für interpretierende Systeme, um UDML-Strukturen mechanisch und ohne Bedeutungsambiguitäten verarbeiten zu können. Sie ist absichtlich minimalistisch gehalten, um maximale Klarheit, Vorhersagbarkeit und Auditierbarkeit zu ermöglichen.

3.1 Lexikalische Einheiten

Alle lexikalischen Einheiten der UDML sind strikt definiert. Die Sprache kennt nur deterministisch interpretierbare Token. Zu den zentralen Tokenklassen gehören:

- **Bezeichner**: alphanumerische Namen für Blöcke, Felder, Operatoren oder Konstanten.
- **Schlüsselwörter**: reservierte Begriffe wie `BLOCK`, `STATE`, `OPERATOR`, `TRANSITION`.
- **Literalwerte**: numerische, boolesche oder textuelle Literale, jeweils explizit typisiert.
- **Strukturzeichen**: Klammern, Doppelpunkte, Trennzeichen, Gleichheitszuweisungen.

Jede lexikalische Einheit besitzt eine eindeutige Interpretation. Es existieren keine kontextabhängigen Tokens, und es gibt keine Überladung von Begriffen.

3.2 Strukturelemente

Die UDML beschreibt komplexe Abläufe durch eine kleine Menge zentraler Strukturelemente. Diese Elemente bilden das Skelett der Sprache und sind in jedem gültigen Dokument wiederzufinden.

- **BLOCK**: eine abgeschlossene deklarative Einheit mit lokaler Struktur, eigenen Zuständen und ausführbaren Operationen.
- **HEADER**: Metadaten eines Blocks, welche Eigenschaften, Namen, Parameter oder Typzuweisungen enthalten.
- **STATE**: die Gesamtheit der deklarierten Werte eines Blocks; alle veränderbaren Größen müssen hier explizit aufgeführt sein.
- **OPERATOR**: Beschreibung einer deterministisch wirkenden Operation.
- **TRANSITION**: Regel zur Ableitung eines Folgezustands aus einem Vorgängerzustand.
- **HOOK / EVENT**: optionales Strukturelement zur Definition deterministischer Ereignisauslöser, die zusätzliche Übergänge ausführen können.

Alle Strukturelemente sind syntaktisch eindeutig erkennbar und besitzen einen klar abgegrenzten Gültigkeitsbereich.

3.3 Grammatik

Die Syntax der UDML ist auf einen deterministischen, kontextfreien Grammatikstil ausgelegt. Die Grammatik definiert die vollständige formale Struktur der Sprache. Ein vereinfachtes Schema lautet:

```
UDML        ::= BLOCK*

BLOCK       ::= "BLOCK" IDENT "{"
                    HEADER
                    STATE
                    OPERATOR*
                    TRANSITION*
                    HOOK*
                "}"

HEADER      ::= "HEADER" "{" FIELD* "}"
STATE       ::= "STATE" "{" FIELD* "}"
FIELD       ::= IDENT "=" VALUE
```

```
OPERATOR    ::= "OPERATOR" IDENT "{" OP_BODY "}"
OP_BODY     ::= (RULE | ACTION | PARAM)*

TRANSITION  ::= "TRANSITION" IDENT "{"
                    FROM
                    TO
                    CONDITION?
                "}"

HOOK        ::= "HOOK" IDENT "{"
                    EVENT
                    EFFECT
                "}"
```

Diese Darstellung illustriert den Kern der Sprache. Ein vollständiges Grammatikmodell folgt in Kapitel 9 der formalen Spezifikation.

3.4 Syntaktische Regeln

UDML verfügt über eine Reihe grundlegender syntaktischer Regeln, die für jede gültige Struktur zwingend angewendet werden müssen:

1. **Jede deklarative Einheit muss geklammert sein**. Blöcke besitzen immer eine öffnende und schließende Klammer.

2. **Jedes Feld hat genau eine Zuweisung**. Mehrfachzuweisungen derselben Variable sind nicht erlaubt.

3. **Operatoren dürfen keine ungebundenen Parameter enthalten**. Jeder Parameter muss entweder literal oder referenziert sein.

4. **Transitionsregeln sind vollständig**. Ein Übergang muss mindestens einen Ausgangszustand und einen Folgezustand definieren.

5. **Bezeichner müssen eindeutig sein**. Innerhalb eines Blocks dürfen keine Namen doppelt vergeben werden.

6. **Hooks sind optional, aber falls vorhanden vollständig**. Ein Hook benötigt mindestens ein Ereignis und einen deterministischen Effekt.

Diese Regeln gewährleisten strukturelle Klarheit und verhindern Mehrdeutigkeiten beim Parsen.

3.5 Validität und Fehlerzustände

Eine UDML-Struktur ist nur dann gültig, wenn sie:

- lexikalisch korrekt ist,
- syntaktisch der Grammatik entspricht,
- keine ungebundenen Bezeichner oder undefinierten Felder enthält,
- vollständige Zustandsdeklarationen besitzt,
- und alle Übergänge semantisch auflösbar sind.

Ungültige Strukturen können unter anderem entstehen durch:

- fehlende Klammern,
- widersprüchliche Felddeklarationen,
- undefinierte Operatoren,
- Übergänge auf nicht existierende Zustände,
- semantisch leere oder inkonsistente Konstrukte.

Validität ist ein strikt formaler Begriff in UDML: Ein Dokument ist entweder gültig oder ungültig — Zwischenstufen existieren nicht.

4 Semantik

Die Semantik der UDML definiert die formalen Bedeutungen der syntaktischen Konstrukte. Sie legt fest, wie Zustände interpretiert, Operatoren angewendet und Übergänge ausgeführt werden. Während die Syntax lediglich die Form beschreibt, bestimmt die Semantik die Wirkung. Entscheidend ist dabei, dass UDML vollständig deterministisch bleibt: Jede deklarierte Struktur besitzt eine eindeutige Bedeutung, und jede Transformation lässt sich als wohldefinierte Funktion ausdrücken.

Die Semantik folgt einem streng formalen Modell, das Zustände, Operatoren und Übergänge als mathematische Objekte beschreibt. Dadurch wird die Sprache unabhängig von jeder implementierenden Maschine und bleibt in allen Kontexten gleich interpretierbar.

4.1 Zustandsmodell

Der Zustand (`STATE`) ist die zentrale semantische Größe in UDML. Er umfasst alle explizit deklarierten Werte eines Blocks. Ein Zustand ist eine Abbildung der Form:

$$STATE = \{\, x_1 = v_1,\ x_2 = v_2,\ \ldots,\ x_n = v_n \,\}$$

mit folgenden Eigenschaften:

- Jeder Bezeichner x_i ist eindeutig.
- Jeder Wert v_i stammt aus einer wohldefinierten Wertedomäne.
- Der Zustand ist vollständig sichtbar und enthält ausschließlich explizit deklarierte Felder.
- Es existieren keine impliziten oder versteckten Zustandsvariablen.

Das Zustandsmodell ist *geschlossen*. Es darf nur verändert werden durch deterministisch spezifizierte Operationen oder Übergänge. Jede Veränderung ist explizit und auditierbar.

4.2 Zustandsfelder und Domänen

Jedes Zustandsfeld besitzt eine zugeordnete Domäne:

$$x : D$$

wobei D eine der folgenden Klassen sein kann:

- numerische Domänen (Ganzzahlen, reelle Zahlen),
- boolesche Domänen,
- symbolische Domänen (Werte aus endlichen Mengen),
- strukturierte Domänen (Listen, Mappingstrukturen),
- textuelle Domänen (deterministisch interpretierbare Zeichenketten).

UDML verlangt, dass:

1. jede Domäne *endliche oder eindeutig interpretierbare* Werte besitzt,
2. keine impliziten Konvertierungen stattfinden,
3. nur deklarierte Operationen Werte verändern dürfen,
4. jeder Wert wohldefiniert sein muss und keinem mehrdeutigen Typ angehört.

Damit wird verhindert, dass Zustände durch unklare Typregeln oder dynamische Interpretationen instabil werden.

4.3 Operatorsemantik

Ein Operator beschreibt eine deterministische Transformation des Zustands. Semantisch lässt sich jeder Operator als Funktion ausdrücken:

$$OP : STATE \rightarrow STATE$$

Ein Operator besteht aus:

- einer Menge von Eingabeparametern,
- einer formalen Regel, wie die Werte dieser Parameter verarbeitet werden,
- einer exakt definierten Wirkung auf den Zustand.

Die Semantik eines Operators erfüllt folgende Bedingungen:

1. **Totalität**: Für jeden wohldefinierten Eingabezustand existiert ein eindeutiges Ergebnis.
2. **Determinismus**: Gleicher Eingangszustand und gleiche Parameter führen *immer* zum gleichen Ausgangszustand.
3. **Isoliertheit**: Ein Operator darf nur die Felder verändern, die er explizit benennt.
4. **Explizite Wirkung**: Alle Änderungen müssen im resultierenden Zustand sichtbar sein.
5. **Keine Seiteneffekte**: Ein Operator darf nicht über den deklarierten Zustandsraum hinaus wirken.

Damit sind Operatoren reine Funktionsobjekte ohne verborgenes Verhalten.

4.4 Übergangsfunktionen

Eine `TRANSITION` beschreibt die deterministische Ableitung eines Folgezustands unter einer gegebenen Bedingung. Formal:

$$STATE_{n+1} = \begin{cases} OP(STATE_n) & \text{falls Bedingung wahr} \\ STATE_n & \text{sonst} \end{cases}$$

Erweiterung für Transitionen mit Operatorsequenz Ist für eine Transition anstelle eines einzelnen Operators ein Feld

$$\texttt{sequence} = [OP_1, OP_2, \ldots, OP_m]$$

angegeben, so ergibt sich der Folgezustand durch sequentielle Anwendung dieser Operatoren:

$$STATE^{(0)} = STATE_n, \qquad STATE^{(i)} = OP_i\Big(STATE^{(i-1)}\Big) \quad \text{für } i = 1, \ldots, m,$$

$$STATE_{n+1} = STATE^{(m)}.$$

Dies entspricht einer zusammengesetzten Operatorfunktion:

$$OP_{\text{seq}} = OP_m \circ OP_{m-1} \circ \cdots \circ OP_1,$$

mit:

$$STATE_{n+1} = OP_{\text{seq}}\Big(STATE_n\Big).$$

Jeder Operator der Sequenz muss deterministisch sein; die Anwendung erfolgt strikt in deklarierter Reihenfolge.

Die Semantik einer Transition setzt sich aus drei Teilen zusammen:

1. **Ausgangszustand**: Referenz auf den Zustand, aus dem heraus die Transition betrachtet wird.
2. **Operator**: Die durchzuführende Transformation.
3. **Bedingung**: Ein boolescher Ausdruck, der deterministisch ausgewertet werden muss.

Eine Transition ist nur dann gültig, wenn:

- der Operator wohldefiniert ist,
- die Bedingung deterministisch auswertbar ist,
- der resultierende Zustand den Domänenregeln entspricht.

Transitionslogik dient in UDML als sequenzieller Mechanismus, nicht als Kontrollfluss im klassischen Sinn. Sie beschreibt ausschließlich Zustandsveränderungen, nicht Ablauflogik außerhalb des Blocks.

4.5 Determinismusregeln

Die Semantik der UDML verlangt strikt:

1. **Eindeutigkeit**: Für jeden syntaktischen Ausdruck existiert genau eine semantische Bedeutung.
2. **Wiederholbarkeit**: Jeder Ausführungspfad kann beliebig oft reproduziert werden.
3. **Geschlossenheit**: Alle Bedeutungen müssen aus der UDML-Struktur selbst ableitbar sein.
4. **Konfliktfreiheit**: Semantische Widersprüche (z. B. doppelte Zuweisungen) sind unzulässig.
5. **Monotonie**: Operatoren dürfen keine widersprüchlichen oder zirkulären Effekte erzeugen.

Diese Regeln machen die gesamte Sprache mathematisch stabil und ermöglichen maschinenbasierte Formalverifikation.

4.6 Semantische Konsistenz und Konfliktfreiheit

Um semantische Stabilität zu gewährleisten, muss jede UDML-Struktur folgende Konsistenzbedingungen erfüllen:

- Jeder Operator wirkt innerhalb seines deklarierten Gültigkeitsraums.
- Jede Transition verweist nur auf existierende Zustände und Operatoren.
- Jede Bedingung ist verifizierbar und deterministisch auswertbar.
- Unterschiedliche Blöcke teilen keine Zustände, außer wenn dies explizit durch klar definierte Mechanismen erlaubt ist.
- Der Zustandsraum ist zu jedem Zeitpunkt vollständig und widerspruchsfrei.

Semantische Konflikte entstehen unter anderem durch:

- doppelte oder widersprüchliche Zuweisungen,
- undefinierte Werte,
- untypisierte Literale,
- nicht auflösbare Bedingungen,
- Operatoren, die außerhalb ihres Wirkungsbereichs schreiben.

Ein semantisch konsistentes UDML-Dokument ist frei von derartigen Konflikten und kann von jedem deterministischen Interpreten eindeutig ausgeführt werden.

5 Blockmodell

Das Blockmodell bildet den strukturellen Kern der UDML. Ein `BLOCK` ist die kleinste vollständig deklarative Einheit der Sprache. Er vereint in sich:

- einen abgeschlossenen lokalen Zustandsraum,
- eine definierte Menge deterministischer Operatoren,
- semantisch wohldefinierte Übergänge,
- optional deterministische Ereignisbehandlungen (Hooks),
- und eine formale Metastruktur (Header).

Jeder Block ist logisch unabhängig und besitzt eine klar definierte interne Architektur. Blöcke können in größeren Strukturen verknüpft werden, doch bleibt ihre innere Semantik stets geschlossen und von außen unverändert.

Das Blockmodell stellt sicher, dass UDML Abläufe präzise, überprüfbar und vollständig deterministisch beschreibt.

5.1 Aufbau eines UDML-Blocks

Ein gültiger UDML-Block folgt einer festen Struktur:

```
BLOCK <Name> {
    HEADER { ... }
    STATE  { ... }
    OPERATOR { ... }
    OPERATOR { ... }
    TRANSITION { ... }
    HOOK { ... }
}
```

Die Reihenfolge der Sektionen ist verpflichtend und trägt zur Klarheit der Interpretation bei. Jede Sektion erfüllt eine eindeutige Funktion:

1. **HEADER** – strukturelle und metadeklarative Informationen.
2. **STATE** – deklarierte Werte, die der Block verwalten darf.
3. **OPERATOR** – deterministisch wirkende Transformationen.
4. **TRANSITION** – semantische Abläufe in Form von Regeln.
5. **HOOK** – ereignisbezogene Zusatzeffekte (optional).

Ein Block ist *geschlossen*: Er darf nur auf seine eigenen Zustände wirken und keine externen Werte verändern, es sei denn, dies ist explizit modelliert und erlaubt.

5.2 HEADER-Struktur

Der `HEADER` ist der metadeklarative Bereich eines Blocks. Er enthält Angaben, die den Block formal beschreiben, jedoch nicht selbst am Zustand oder Ablauf teilnehmen. Beispiele für Inhalte:

- Typ oder Kategorie des Blocks,
- Version oder interne Kennungen,
- statische Parameter,
- Konfigurationen, die lokale Interpretation betreffen.

Der HEADER besitzt maximal deklarativen Charakter. In ihm vorkommende Werte:

- sind unveränderlich während der Ausführung,
- dürfen nicht semantisch mit STATE-Werten verwechselt werden,
- können Operatoren beeinflussen, aber werden selbst nicht verändert.

Ein HEADER hat die Form:

```
HEADER {
    key1 = value1
    key2 = value2
    ...
}
```

HEADER-Felder sind schreibgeschützt und dienen ausschließlich der Struktur.

5.3 STATE-Struktur

Der `STATE`-Bereich definiert alle dynamischen Werte eines Blocks. Nur hier deklarierte Werte dürfen verändert werden.

Ein Zustand besitzt folgende Eigenschaften:

- Alle Felder müssen explizit deklariert sein.
- Werte müssen wohldefinierten Domänen angehören.
- Kein Feld darf mehrfach definiert werden.
- Keine impliziten oder automatischen Felder existieren.

Der STATE-Bereich hat die Form:

```
STATE {
    var1 = initial_value
    var2 = initial_value
    ...
}
```

Ein STATE ist der einzige zulässige Speicherbereich eines Blocks. Operatoren dürfen ausschließlich auf STATE-Felder zugreifen, nicht auf HEADER.

5.4 Operatoren

Ein `OPERATOR` beschreibt eine deterministisch definierte Transformation von einem Zustand in einen anderen. Jeder Operator besitzt:

- einen Namen,
- eine wohldefinierte Parameterstruktur,
- eine Menge von Regeln oder Aktionen,
- eine präzise definierte Wirkung auf den STATE.

Ein Operator hat die syntaktische Form:

```
OPERATOR <Name> {
    action1 = ...
    action2 = ...
}
```

Semantische Anforderungen:

1. Ein Operator darf nur deklarierte STATE-Felder verändern.
2. Ein Operator darf seine Parameter nicht dynamisch ändern.
3. Ein Operator muss deterministisch sein.
4. Ein Operator erzeugt genau einen Folgezustand.

Operatoren sind der einzige Ort, an dem der Zustand verändert wird.

5.5 Transitionsregeln

Eine TRANSITION definiert die Bedingungen, unter denen ein Operator auf den Zustand angewendet wird. Sie besitzt die Form:

```
TRANSITION <Name> {
    from = state_reference
    to   = operator_name
    when = condition
}
```

Alternativ kann eine Transition eine geordnete Sequenz von Operatoren definieren:

```
TRANSITION <Name> {
  from     = state_reference
  sequence = [ operator_name_1, operator_name_2, ... ]
  when     = condition
  priority = <integer_literal>      // optional
  exclusive = <bool_literal>        // optional
}
```

- sequence gibt eine strikt geordnete Liste von Operatoren an, die nacheinander auf de
- priority legt fest, welche Transition gewählt wird, wenn mehrere Transitionen gleich
- exclusive = true erzwingt, dass in demselben Ausführungsschritt keine weitere Transi

Determinismusauflösung bei mehreren erfüllten Transitionen:

1. Sind mehrere Transitionen gleichzeitig aktiviert, wird die Transition mit dem höchs
2. Haben mehrere aktivierte Transitionen denselben priority-Wert, ist das Dokument ser
3. Ist exclusive = true für die ausgewählte Transition gesetzt, dürfen in diesem Ausfu

Semantik:

- Eine Transition ist ein deterministischer Ablaufknoten.
- Die Bedingung (`when`) muss boolesch und aus dem aktuellen STATE berechenbar sein.
- Die Transition verändert den Zustand ausschließlich durch Anwendung des referenzierten Operators.

- Wenn die Bedingung falsch ist, findet keine Transformation statt.

Transitionslogik dient in UDML als *semantischer Ablaufrahmen*, nicht als imperative Steuerlogik. Sie beschreibt ausschließlich Zustandsübergänge, keine kontrollstrukturellen Programmabläufe.

5.6 Hooks und Ereignisse

Ein `HOOK` ist ein optionales Konstruktelement, das einen definierten Zusatzeffekt beschreibt, der unter bestimmten Ereignissen ausgelöst wird. Hooks dürfen nur deterministische Effekte haben.

Allgemeine Form:

```
HOOK <Name> {
    on     = event_name
    effect = operator_name
}
```

Semantische Anforderungen:

- Ereignisse müssen eindeutig identifizierbar sein.
- Effekte dürfen ausschließlich über existierende Operatoren realisiert werden.
- Hooks müssen deterministisch auswertbar sein.
- Hooks dürfen keine verdeckten Zustandsänderungen verursachen.

Hooks erweitern das Blockmodell um deterministische Reaktionsmechanismen, ohne dessen Grundprinzipien zu verletzen.

6 Ausführungsmodell

Das Ausführungsmodell der UDML beschreibt die formalen Prinzipien, nach denen ein deklarierter Block oder eine Menge von Blöcken deterministisch interpretiert wird. Obwohl UDML selbst keine Implementierung definiert, legt die Sprache exakt fest, wie eine gültige Ausführungsinstanz die deklarativen Strukturen zu lesen, zu bewerten und zu transformieren hat.

Das Modell dient sowohl als theoretische Grundlage für Parser und Interpreter als auch als Garant für Reproduzierbarkeit und Auditierbarkeit. Jede Ausführung muss denselben Regeln folgen und darf keine zusätzlichen, nicht deklarativ abgeleiteten Effekte erzeugen.

6.1 Formale Ausführungspfade

Ein Ausführungspfad ist die deterministische Folge von Zustandsübergängen, die aus einem UDML-Dokument hervorgehen. Formal lässt sich ein Auslaufpfad als Sequenz schreiben:

$$STATE_0 \rightarrow STATE_1 \rightarrow \cdots \rightarrow STATE_n$$

mit der Eigenschaft, dass jeder Übergang durch eine gültige Transition und einen gültig angewendeten Operator zustande kommt.

Ein Ausführungspfad ist vollständig beschrieben durch:

1. den Startzustand $STATE_0$,
2. die Menge der in Frage kommenden Transitionen,
3. die Reihenfolge, in der Bedingungen erfüllt werden,
4. die durch Operatoren erzeugten Zielzustände.

Jeder gültige Pfad ist endlich oder terminiert in einem Zustand, in dem keine Transition mehr erfüllt ist. Zyklen sind nur erlaubt, wenn sie deterministisch auflösbar sind und weder unentscheidbare Schleifen noch semantische Widersprüche entstehen.

6.2 Reproduzierbarkeit

Ein Kernelement des Ausführungsmodells ist vollständige Reproduzierbarkeit. Dies bedeutet:

1. Gleiche Eingabe → Gleicher Zustand → Gleiche Transition → Gleiche Operatorwirkung.
2. Keine versteckten Variablen oder impliziten Zustände.
3. Keine Verwendung externer Zufallsquellen.
4. Keine kontextabhängigen Bedeutungsverschiebungen.

Formal lässt sich Reproduzierbarkeit ausdrücken durch:

$$\forall STATE_n,\ OP: \quad OP(STATE_n) = STATE_m \quad \text{eindeutig}$$

Dies stellt sicher, dass jede Ausführung auf jedem kompatiblen Interpreten denselben Endzustand erzeugt, sofern das UDML-Dokument identisch ist.

Reproduzierbarkeit ist damit nicht nur ein Prinzip, sondern eine mathematische Notwendigkeit für gültige UDML-Ausführungen.

6.3 Validierungslogik

Bevor ein UDML-Dokument ausgeführt werden kann, muss es validiert werden. Die Validierung erfolgt anhand formaler Kriterien:

- syntaktische Korrektheit,
- eindeutige Bezeichner,
- vollständig deklarierte Zustandsfelder,
- wohldefinierte Operatoren,
- gültige und deterministische Bedingungen,
- auflösbare Transitionspfade.

Ein UDML-Dokument ist nur dann ausführbar, wenn *alle* folgenden Bedingungen erfüllt sind:

1. Jeder Block ist syntaktisch vollständig.
2. Keine Struktur enthält widersprüchliche Zuweisungen.
3. Jeder Operator ist wohldefiniert und wirkt nur auf deklarierte Felder.
4. Jede Transition verweist auf existierende Operatoren und Zustände.
5. Jede Bedingung ist deterministisch und auswertbar.

Die Validierungslogik ist streng und lässt keine semantischen Graubereiche zu. Ist eine Struktur nicht eindeutig interpretierbar, gilt das gesamte Dokument als ungültig.

6.4 Auditierbarkeit

Das Ausführungsmodell ist so gestaltet, dass jede Ausführung vollständig prüfbar ist. Dies ergibt sich aus drei Prinzipien:

Transparenz: Jeder Zustandswert ist jederzeit sichtbar und eindeutig bestimmt.

Explizitheit: Jede Veränderung eines Zustands resultiert aus genau einem Operator und ist eindeutig nachvollziehbar.

Deterministische Protokollierbarkeit: Ein Interpertenmodell kann sämtliche Übergänge als sequenzielles Protokoll abbilden, ohne Informationsverlust.

Damit ist jedes Ausführungsergebnis:

- rekonstruierbar,
- vergleichbar,
- überprüfbar,
- formal ableitbar.

Auditierbarkeit ist kein optionaler Zusatz, sondern eine direkte Konsequenz der Semantik und Syntax der UDML.

6.5 Integritätsanforderungen

Validität und Ausführbarkeit setzen voraus, dass die Integrität des Dokuments zu jedem Zeitpunkt gegeben ist. Die UDML definiert hierfür strenge Anforderungen:

1. **Zustandsintegrität:** Es dürfen nur deklarierte Felder existieren und genutzt werden.
2. **Semantische Integrität:** Jeder Ausdruck muss wohldefiniert und deterministisch interpretierbar sein.
3. **Operatorische Integrität:** Operatoren müssen vollständige, widerspruchsfreie und isolierte Wirkungsdefinitionen besitzen.
4. **Transitionsintegrität:** Transitionen müssen eindeutige, nicht widersprechende Pfade erzeugen.
5. **Formale Geschlossenheit:** Ein UDML-Dokument darf keine semantischen Lücken oder unerfüllbaren Abhängigkeiten enthalten.

Diese Integritätsanforderungen sind zwingend für jede Form der Ausführung. Verstöße führen dazu, dass ein Dokument als nicht interpretierbar eingestuft wird.

7 Kompatibilitäts- und Erweiterungsrahmen

Die UDML ist so gestaltet, dass sie als stabile, unveränderliche Beschreibungsebene dienen kann, während darüberliegende technische oder organisatorische Systeme sich weiterentwickeln. Dieses Kapitel beschreibt die regulatorischen und strukturellen Rahmenbedingungen, unter denen UDML erweitert, eingebettet und versioniert werden darf, ohne dass ihre grundlegenden Garantien – Determinismus, Transparenz, Auditierbarkeit – verloren gehen.

Das Ziel des Kapitels ist es, klar abzugrenzen, welche Arten von Erweiterungen zulässig sind, welche strikt ausgeschlossen bleiben müssen und wie UDML in komplexen Architekturen eine unveränderliche, verlässliche Grundlage bildet.

7.1 UDML als unabhängige Basisschicht

UDML fungiert als formal definierte, vollständig deterministische Basisschicht, auf der beliebige übergeordnete Systeme aufsetzen können. Diese Systeme können:

- zusätzliche Regeln interpretieren,
- externe Bewertungen durchführen,
- komplexe Kontrollmechanismen implementieren,
- Abläufe überwachen oder strukturieren.

Doch entscheidend ist:

Die Bedeutung aller UDML-Elemente bleibt unverändert, unabhängig von der Umgebung.

Dadurch bleibt UDML:

- technologisch neutral,
- kontextunabhängig,
- stabil über lange Zeiträume,
- vollständig austauschbar zwischen interpretierenden Systemen.

Jede Ausführungsinstanz darf zusätzliche Logik besitzen, aber sie darf nie die Semantik eines UDML-Dokuments verändern.

7.2 Einbettung in übergeordnete Systeme

UDML kann in eine Vielzahl übergeordneter Systemarchitekturen eingebettet werden. Diese Systeme dürfen:

1. UDML-Blöcke interpretieren,
2. zusätzliche, nicht-UDML-spezifizierte Logik verwenden,
3. Ergebnisse mit externen Kriterien bewerten,
4. Ablaufsteuerung oberhalb der Blockebene implementieren.

Dabei gilt zwingend:

- UDML gibt nur die deklarative Struktur vor, nicht die technische Ausführung.
- Übergeordnete Systeme dürfen Bedeutungen erweitern, aber nicht überschreiben.
- Jede Auswertung muss deterministisch im Rahmen der UDML bleiben; externe Variantenbedingungen dürfen nicht die UDML-Semantik beeinflussen.

Damit bleibt UDML unabhängig von jeder Kontroll-, Regel- oder Ausführungsontologie.

7.3 Erweiterbarkeit des Operatorraums

Der Operatorraum ist prinzipiell erweiterbar. Neue Operatoren dürfen eingeführt werden, sofern sie:

- deterministisch definiert sind,
- ihre Wirkung vollständig explizieren,
- ausschließlich auf deklarierte Zustände wirken,
- keine impliziten Nebeneffekte besitzen.

Ein neuer Operator darf somit niemals:

- semantische Ambiguitäten erzeugen,
- implizite Typecasts einführen,
- sich unterschiedlich verhalten, abhängig von externem Kontext,
- unkontrollierbare Nebenwirkungen produzieren.

Formale Bedingung für neue Operatoren:

$OP_{\text{neu}} : STATE \to STATE$ muss total, deterministisch und wohldefiniert sein.

Erweiterungen des Operatorraums verändern die UDML nicht — sie erweitern lediglich die Menge der innerhalb der Sprache ausdruckbaren Operationen.

7.4 Versionierung

Da UDML als stabile Basisschicht konzipiert ist, muss ihre Versionierung konservativ erfolgen. Versionen unterscheiden sich ausschließlich durch:

- Präzisierungen der bestehenden Regeln,
- Hinzufügen neuer, kompatibler Konstrukte,
- Erweiterung der Grammatik unter Beibehaltung der alten Form,
- Verbesserungen der formalen Spezifikation.

Nicht zulässig sind:

- Änderungen der Semantik bestehender Elemente,

- Umdeutungen von Schlüsselwörtern,
- Entfernen bisher gültiger Konstrukte,
- nicht rückwärtskompatible Brüche.

Versionierung folgt dem Prinzip:

$$\text{Neue Version} = \text{Alte Version} + \text{kompatible Erweiterungen}$$

Somit bleibt jedes gültige UDML-Dokument auch in zukünftigen Versionen gültig.

7.5 Formale Erweiterungsregeln

Um die langfristige Stabilität der Sprache zu gewährleisten, gelten klare Erweiterungsregeln:

1. Erweiterungen dürfen keine semantischen Konflikte verursachen.
2. Erweiterungen müssen rückwärtskompatibel sein.
3. Erweiterungen müssen deterministisch interpretierbar sein.
4. Erweiterungen dürfen nicht gegen bestehende Typ- und Struktursysteme verstoßen.
5. Erweiterungen müssen formal überprüfbar und vollständig definierbar sein.

Jede Erweiterung der Grammatik, Semantik oder Struktur muss durch eine formale, maschinenlesbare Spezifikation ergänzt werden, um die auditierbare Natur der Sprache zu erhalten.

Erweiterungen, die diese Bedingungen nicht erfüllen, sind nicht Teil der UDML.

8 Beispiele

Dieses Kapitel enthält Beispiele für gültige UDML-Strukturen unterschiedlicher Komplexität. Die Beispiele demonstrieren die praktische Anwendung der zuvor definierten syntaktischen und semantischen Regeln. Sie sind bewusst minimalistisch gehalten, um die Kernprinzipien der Sprache klar erkennbar zu machen, zugleich jedoch vollständig und deterministisch ausführbar.

Alle Beispiele folgen strikt dem Blockmodell, verwenden wohldefinierte Zustandsräume und besitzen vollständig transparente Operator- und Transitionslogik.

8.1 Minimaler UDML-Block

Der folgende Block zeigt die minimal notwendige Struktur einer gültigen UDML- Einheit. Er enthält einen Header, einen State, einen Operator und eine Transition.

```
BLOCK MinimalExample {
    HEADER {
        version = 1
        description = "Minimaler gültiger UDML-Block"
    }

    STATE {
        counter = 0
    }

    OPERATOR Increment {
        counter = counter + 1
    }

    TRANSITION IncreaseCounter {
        from = counter
        to   = Increment
        when = (counter < 1)
    }
}
```

Dieser Block erhöht den Zähler von 0 auf 1, solange die Bedingung erfüllt ist. Nach der ersten Anwendung der Transition ist die Bedingung nicht mehr erfüllt und die Ausführung terminiert deterministisch.

8.2 Block mit erweiterten Metadaten

Der folgende Block zeigt die Verwendung komplexerer Header-Daten sowie mehrerer Zustandsfelder.

```
BLOCK MetadataExample {
    HEADER {
        block_type = "calculation"
        version    = 2
        author     = "System"
        category   = "demo"
    }
```

```
    STATE {
        a = 3
        b = 5
        result = 0
    }

    OPERATOR ComputeSum {
        result = a + b
    }

    TRANSITION RunSum {
        from = result
        to   = ComputeSum
        when = (result == 0)
    }
}
```

Hier werden zwei konstante Werte summiert und das Ergebnis in `result` gespeichert. Die Headerinformationen beeinflussen die Semantik nicht; sie dienen lediglich der Metadokumentation.

8.3 Komplexere Zustandsstrukturen

Das nächste Beispiel zeigt eine Struktur mit mehreren Operatoren und einem Zustandsraum, der Zahlen und symbolische Werte kombiniert.

```
BLOCK MultiStepProcess {
    HEADER {
        purpose = "Demonstration eines mehrstufigen Ablaufs"
    }

    STATE {
        stage  = "init"
        value  = 0
        result = 0
    }

    OPERATOR Initialize {
        value = 10
        stage = "initialized"
```

```
    }

    OPERATOR Transform {
        result = value * 2
        stage  = "transformed"
    }

    TRANSITION InitStep {
        from = stage
        to   = Initialize
        when = (stage == "init")
    }

    TRANSITION TransformStep {
        from = stage
        to   = Transform
        when = (stage == "initialized")
    }
}
```

Dieser Block zeigt einen sequentiellen Ablauf:

1. Initialisierung (setzt `value` auf 10),
2. Transformation (verdoppelt den Wert und setzt die Phase fort).

Die Ausführung ist klar determiniert, da jede Transition eine eindeutige Bedingung besitzt.

8.4 Deterministische Übergänge

Dieses Beispiel demonstriert die Verwendung von Bedingungen zur Steuerung deterministischer Abläufe.

```
BLOCK Countdown {
    HEADER {
        description = "Ein einfacher Countdown"
    }

    STATE {
        n = 5
    }
```

```
    OPERATOR Decrement {
        n = n - 1
    }

    TRANSITION Step {
        from = n
        to   = Decrement
        when = (n > 0)
    }
}
```

Die Ausführung reduziert `n` deterministisch von 5 auf 0. Ab `n == 0` findet keine Transition mehr statt, womit der Ablauf terminiert.

8.5 Zusammengesetzte Operatorsequenzen

Im nächsten Beispiel wird gezeigt, wie mehrere Operatoren in Abhängigkeit aufeinanderfolgender Übergänge wirken können.

```
BLOCK SequentialOps {
    HEADER {
        purpose = "Demonstration mehrerer Operationen"
    }

    STATE {
        x = 2
        y = 3
        z = 0
    }

    OPERATOR Multiply {
        z = x * y
    }

    OPERATOR AddOffset {
        z = z + 5
    }

    TRANSITION First {
        from = z
```

```
        to   = Multiply
        when = (z == 0)
    }

    TRANSITION Second {
        from = z
        to   = AddOffset
        when = (z > 0)
    }
}
```

Beispielablauf:

1. `z` wird durch Multiplikation berechnet.
2. Danach wird ein Offset addiert.

Alle Schritte sind vollständig durchsichtig und deterministisch.

8.6 Fehlerhafte Blöcke und ihre Erkennung

Das letzte Beispiel zeigt typische Fehler, die ein UDML-Validator erkennen muss. Diese Blöcke sind **ungültig**.

Beispiel: Undefiniertes Feld

```
BLOCK ErrorUndefined {
    STATE {
        x = 1
    }

    OPERATOR BadOp {
        y = x + 1   // Fehler: 'y' ist nicht deklariert
    }
}
```

Beispiel: Mehrfache Deklaration

```
STATE {
    value = 0
    value = 1   // Fehler: doppelte Definition
}
```

Beispiel: Nicht auflösbare Transition

```
TRANSITION Faulty {
    from = counter    // Fehler: 'counter' existiert nicht
    to   = Increment  // Fehler: 'Increment' existiert nicht
    when = (true)
}
```

Ein gültiger Parser muss solche Dokumente strikt zurückweisen.

9 Formale Spezifikation

Dieses Kapitel beschreibt die UDML in einer formalisierten, maschinenlesbaren Form. Die verwendeten Darstellungsmittel sind BNF und EBNF. Es werden alle Terminals, Non-Terminals, syntaktischen Strukturen und die deterministischen Regeln zur Definition von Zuständen, Operatoren und Übergängen formalisiert.

Die Spezifikation ist so gestaltet, dass jeder kompatible Parser und jeder deterministische Interpreter sie direkt implementieren kann. Alle vorherigen Kapitel beschreiben die Semantik, Syntax und Struktur; dieses Kapitel liefert die formale Grundlage, auf der diese Konzepte eindeutig und frei von Mehrdeutigkeiten definiert sind.

9.1 BNF-Grundgerüst der Sprache

Die folgende BNF beschreibt die oberste Ebene eines UDML-Dokuments:

```
<udml>        ::= <block>*
<block>       ::= "BLOCK" <ident> "{"
                      <header>
                      <state>
                      <operator>*
                      <transition>*
                      <hook>*
                  "}"
```

Jeder Block besteht aus genau einem Header, einem State-Bereich und einer beliebigen Anzahl von Operatoren, Transitionen und optionalen Hooks.

9.2 Terminals und Non-Terminals

Terminals:

- Schlüsselwörter: `BLOCK`, `HEADER`, `STATE`, `OPERATOR`, `TRANSITION`, `HOOK`, `from`, `to`, `when`, `on`, `effect`

- Literale: Zahlen, boolesche Werte, Zeichenketten
- Strukturzeichen: {, }, =

Non-Terminals:

- `<udml>` — Wurzelknoten eines Dokuments
- `<block>` — vollständige deklarative Einheit
- `<header>` — Metadaten des Blocks
- `<state>` — deklarierter Zustandsraum
- `<operator>` — deterministische Operation
- `<transition>` — semantische Regel für Zustandstransformation
- `<hook>` — Ereignisreaktion
- `<field>` — Feldzuweisung
- `<value>` — Literal oder Ausdruck
- `<ident>` — Bezeichner

9.3 Header-Spezifikation

```
<header>      ::= "HEADER" "{" <field>* "}"
<field>       ::= <ident> "=" <value>
```

Regeln:

1. Alle Felder müssen eindeutige Bezeichner besitzen.
2. Werte sind unveränderlich und dienen nur der Dokumentation oder lokalen Auslegung, nicht der Zustandsspeicherung.

9.4 State-Spezifikation

```
<state>       ::= "STATE" "{" <field>* "}"
```

Semantische Bedingungen:

- Jeder Zustand muss initialisiert sein.
- Keine zwei Felder dürfen denselben Bezeichner teilen.
- Der STATE-Bereich bildet den einzigen veränderbaren Raum eines Blocks.

9.5 Operator-Spezifikation

```
<operator>    ::= "OPERATOR" <ident> "{" <op-body>* "}"
<op-body>     ::= <assignment> | <rule>
<assignment>  ::= <ident> "=" <value>
<rule>        ::= <ident> "=" <expression>
```

Operatoren bestehen aus deterministischen Zuweisungen und Regeln. Jede Zuweisung beschreibt eine Zustandsveränderung der Form:

$$STATE' = STATE \cup \{x = f(STATE)\}$$

wobei f ein wohldefinierter Ausdruck ist.

9.6 Transition-Spezifikation

```
<transition> ::= "TRANSITION" <ident> "{"
  "from" "=" <ident>
  ( "to" "=" <ident>
  | "sequence" "=" "[" <ident> ("," <ident>)* "]" )
  "when" "=" <condition>
  [ "priority" "=" <integer_literal> ]
  [ "exclusive" "=" <bool_literal> ]
"}"
```

Formale Anforderungen:

1. Der `from`-Bezeichner muss ein im STATE definierter Wert sein.
2. Der `to`-Bezeichner muss ein existierender Operator des Blocks sein.
3. Die Bedingung muss deterministisch auswertbar sein.

Transitionssemantik:

$$STATE_{n+1} = \begin{cases} OP(STATE_n), & \text{falls Condition}(STATE_n) = \text{true} \\ STATE_n, & \text{sonst} \end{cases}$$

9.7 Hook-Spezifikation

```
<hook>        ::= "HOOK" <ident> "{"
                      "on"     "=" <ident>
                      "effect" "=" <ident>
                  "}"
```

Hooks binden deterministische Reaktionen an Ereignisse. Ein Hook ist gültig, wenn:

- das Ereignis wohldefiniert ist,
- der referenzierte Operator existiert,
- die Ausführung deterministisch bleibt.

9.8 Formale Beschreibung des Operatorraums

Der Operatorraum besteht aus einer Menge von Abbildungen:

$$\mathcal{O} = \{ OP_1, OP_2, \ldots, OP_k \}$$

Jede Abbildung erfüllt:

$$OP_i : STATE \to STATE$$

und muss total, deterministisch und konfliktfrei sein.
Zulässige Ausdrücke in Operatoren:

```
<expression> ::= <value>
               | <ident>
               | <expression> <binop> <expression>
               | "(" <expression> ")"

<binop>      ::= "+" | "-" | "*" | "/" | "==" | "!=" | "<" | ">" | "<=" | ">="
```

Alle Operatorwirkungen müssen sich ohne Kontextwissen rein aus der Definition ableiten lassen.

9.9 Formale Beschreibung der Zustandsmodelle

Ein STATE ist eine endliche Menge von Paaren:

$$STATE = \{(x_1, v_1), \ldots, (x_n, v_n)\}$$

mit:

- eindeutigen Schlüsseln x_i,
- wohldefinierten Werten v_i,
- festgelegten Domänen.

Die formale Update-Regel lautet:

$$STATE_{n+1}(x) = \begin{cases} f(STATE_n), & x = \text{Zielvariable eines Operators,} \\ STATE_n(x), & \text{sonst.} \end{cases}$$

9.10 Referenzstrukturen in BNF

Zur vollständigen Spezifikation einer UDML-Einheit genügt folgende formale Definition:

```
<udml>        ::= <block>*

<block>       ::= "BLOCK" <ident> "{"
                        <header>
                        <state>
                        <operator>*
                        <transition>*
                        <hook>*
                  "}"

<header>      ::= "HEADER" "{" <field>* "}"
<state>       ::= "STATE"  "{" <field>* "}"

<field>       ::= <ident> "=" <value>

<operator>    ::= "OPERATOR" <ident> "{"
                        <assignment>*
                  "}"
<assignment>  ::= <ident> "=" <expression>

<transition>  ::= "TRANSITION" <ident> "{"
                        "from" "=" <ident>
                        "to"   "=" <ident>
                        "when" "=" <condition>
                  "}"

<hook>        ::= "HOOK" <ident> "{"
                        "on"     "=" <ident>
                        "effect" "=" <ident>
                  "}"
```

Diese Grammatik beschreibt die vollständige sprachliche Struktur der UDML sowie die deterministisch interpretierbaren Bestandteile jedes gültigen Dokuments.

10 Sicherheit und Robustheit

Die UDML ist so konzipiert, dass sie in Umgebungen eingesetzt werden kann, in denen formale Präzision, langfristige Stabilität und vollständige Vorhersagbarkeit zwingend erforderlich sind. Sicherheit und Robustheit ergeben sich aus der Kombination von deterministischer Semantik, transparenter Syntaxstruktur und strengen Integritätsregeln.

Dieses Kapitel beschreibt die fundamentalen Prinzipien, die sicherstellen, dass UDML-Dokumente zuverlässig, dauerhaft konsistent und frei von semantischer Drift bleiben. Die definierten Regeln bilden eine feste Grundlage für die Verwendung in kritischen oder auditpflichtigen Anwendungen.

10.1 Semantische Stabilität

Semantische Stabilität bedeutet, dass die Bedeutung aller UDML-Konstrukte über die gesamte Lebensdauer eines Dokuments unverändert bleibt. Dies wird durch folgende Mechanismen garantiert:

- **Explizite Definition aller Elemente:** Nichts besitzt Bedeutung ohne deklarierte Struktur.
- **Kontextunabhängige Semantik:** Die Interpretation hängt ausschließlich vom Dokument ab, nicht von der Umgebung.
- **Fehlen impliziter Mechanismen:** Keine automatische Ableitung, kein implizites Verhalten.
- **Determinismus in jedem Operator:** Jede Ausführung führt zuverlässig zum gleichen Ergebnis.

Damit bleibt die UDML resistent gegenüber Veränderungen in technischen Systemen, Softwareversionen oder Ausführungsmodalitäten.

10.2 Driftprävention (abstrakt)

Semantische Drift entsteht, wenn sich Bedeutungen oder Wirkungen über die Zeit unbemerkt verändern. In UDML ist driftpräventives Verhalten inhärent:

1. **Keine impliziten Bedeutungen** — Bedeutungen können nicht unmerklich wandern, da sie nie implizit waren.
2. **Keine kontextabhängigen Werte** — Die Umgebung kann die Semantik nicht beeinflussen.
3. **Keine dynamische Semantik** — Operatoren und Felder ändern ihre Bedeutung niemals zur Laufzeit.

4. **Strukturale Geschlossenheit** — Der gesamte semantische Raum liegt sichtbar innerhalb des Dokuments.

UDML-Dokumente behalten daher auch bei langfristiger Nutzung eine stabile, unveränderte Bedeutung.

10.3 Interne Konsistenzprüfungen

Ein UDML-Dokument muss intern konsistent sein. Die Sprache erzwingt diese Konsistenz durch mehrere deterministische Anforderungen:

- **Jeder Bezeichner muss eindeutig sein.**
- **Jede Zuweisung darf nur ein einziges Ziel besitzen.**
- **Jede Variable muss vor der Verwendung deklariert sein.**
- **Jeder Operator muss vollständig definierte Wirkungen besitzen.**
- **Jede Transition muss logisch auflösbar sein.**
- **Jede Bedingung muss deterministisch auswertbar sein.**

Sobald eine dieser Bedingungen verletzt wird, gilt das gesamte Dokument als ungenültig. Die UDML kennt kein teilweise gültig“.

Die Konsistenzprüfung folgt dem Grundsatz:

Ist ein Dokument nicht eindeutig interpretierbar, ist es nicht gültig.

10.4 Fehlerklassen und deterministische Behandlung

Die UDML unterscheidet mehrere Kategorien von Fehlern, die während der Validierung auftreten können. Jede Fehlerklasse besitzt klar definierte Eigenschaften:

Lexikalische Fehler Ungültige Zeichen, falsch formatierte Literale, nicht erlaubte Tokens.

Syntaktische Fehler Ungültige Struktur wie fehlende Klammern, falsche Reihenfolgen, unvollständige Blöcke.

Semantische Fehler Widersprüche in den Bedeutungen, z. B.:

- doppelte Deklaration eines Feldes,
- fehlende Operatoren oder Transitionen,
- nicht auflösbare Bedingungen,
- Referenzen auf nicht existierende Bezeichner.

Integritätsfehler Zustände, die gegen Domänen oder strukturelle Regeln verstoßen.

Determinismusverletzungen Operatoren oder Bedingungen, die mehrdeutige Ergebnisse erzeugen könnten.

Alle Fehlerklassen führen deterministisch dazu, dass ein Dokument **nicht ausführbar** ist. Es gibt keine automatische Reparatur, keine implizite Korrektur — jede Anpassung muss explizit durch den Autor erfolgen.

10.5 Formale Grenzen der Sprache

Um Sicherheit und Robustheit dauerhaft zu erhalten, definiert die UDML klare Systemgrenzen. Die Sprache:

- unterstützt keine dynamische Codegenerierung,
- erlaubt keine Selbstmodifikation ihrer eigenen Strukturen,
- besitzt keine nichtdeterministischen Mechanismen,
- integriert keine externen Bedeutungsquellen,
- ermöglicht keine unendlichen Strukturen oder unauflösbare Rekursionen.

Diese Grenzen verhindern:

- unkontrollierbare Ausführungspfade,
- instabile Semantik,
- schwer prüfbare Interpretationen,
- sicherheitskritische Inkonsistenzen.

Die UDML bleibt dadurch eine klar definierte, präzise und vollständig kontrollierbare Sprache.

11 Metaebene der Sprache

Die UDML besitzt eine klar definierte Metastruktur, welche beschreibt, warum die Sprache als deterministische Beschreibungsschicht funktioniert und wie sie sich selbst formalisiert. Die Metaebene umfasst alle Prinzipien, die nicht den operativen Teil eines Dokuments betreffen, sondern die Designidee, die ontologische Konsistenz und die interpretatorische Neutralität der Sprache.

Ein wesentliches Merkmal der UDML ist ihre Fähigkeit, ohne äußere Interpretationsannahmen vollständig beschrieben und analysiert zu werden. Diese Eigenschaft macht sie zu einer stabilen Grundlage für technische, organisatorische und theoretische Systeme, die Transparenz und Nachvollziehbarkeit benötigen.

11.1 Selbstbeschreibungsfähigkeit

Die UDML ist in sich geschlossen aufgebaut. Das bedeutet:

- Alle syntaktischen Elemente sind innerhalb der Sprache definierbar.
- Die Semantik ist vollständig aus der Grammatik und den Regeln ableitbar.
- Jede Struktur kann durch dieselben Mechanismen beschrieben werden, die sie selbst verwendet.

Selbstbeschreibungsfähigkeit bedeutet nicht, dass UDML eine Metasprache für sich selbst erzeugt; vielmehr ist die Sprache so konstruiert, dass keine externen Bedeutungsquellen notwendig sind, um ihre Strukturen zu interpretieren.

Damit wird verhindert, dass:

- Interpretationsdifferenzen zwischen Systemen auftreten,
- verschiedene Implementierungen abweichende Bedeutungen erzeugen,
- Kontextwissen außerhalb der Sprache nötig wird.

Die Sprache beschreibt sich nicht selbst in einem rekursiven Sinn, aber sie enthält alle Regeln, die notwendig sind, um jede UDML-Struktur deterministisch zu interpretieren.

11.2 UDML als Grundlage technischer Transparenz

Ein zentrales Designziel der UDML ist maximale Transparenz. Dies ergibt sich aus mehreren architektonischen Prinzipien:

1. **Explizite Zustandsmodelle:** Alle sichtbaren Daten sind vollständig und unverdeckt definiert.
2. **Strikte Trennung zwischen Deklaration und Ausführung:** Die Beschreibungsebene ist frei von Implementierungsdetails.
3. **Deterministische Abläufe:** Jeder Schritt ist nachvollziehbar, wiederholbar und prüfbar.
4. **Formale Strukturierbarkeit:** Die Grammatik erlaubt vollständige maschinelle Analyse.

Durch diese Eigenschaften eignet sich UDML für Anwendungen, in denen:

- Entscheidungen transparent dokumentiert werden müssen,
- Abläufe unabhängig geprüft werden sollen,
- langfristige Stabilität und Revision notwendig sind.

Transparenz ist somit keine Nebenwirkung, sondern eine fundamentale Konsequenz der Architektur der Sprache.

11.3 Prinzipien der Meta-Reflexion

Meta-Reflexion bezeichnet die Fähigkeit, die Sprache bezüglich ihrer eigenen Struktur zu verstehen, ohne sich selbst zu verändern. UDML unterstützt diese Fähigkeit durch folgende Prinzipien:

- **Strukturelle Einfachheit:** Die Sprache besteht aus wenigen, klar definierten Konstrukten.
- **Formale Einheitlichkeit:** Alle Konstrukte folgen denselben syntaktischen und semantischen Regeln.
- **Abschließende Modellierung:** Jede deklarative Einheit ist geschlossen und vollständig beschreibbar.
- **Konsequente Eliminierung impliziter Bedeutungen:** Nichts geschieht außerhalb der formalen Beschreibungen.

Diese Prinzipien ermöglichen es, UDML-Dokumente nicht nur auszuführen, sondern sie strukturell zu analysieren, zu vergleichen, zu klassifizieren und formal zu verstehen.

Meta-Reflexion ist eine Eigenschaft des Designs — die Sprache erzeugt sie nicht aktiv, sondern macht sie durch ihre Genauigkeit zwangsläufig möglich.

11.4 Motivation der Designentscheidungen

Die Architektur der UDML basiert auf mehreren übergeordneten Zielen:

1. **Determinismus:** Der Kern des Designs besteht darin, jede Form von Mehrdeutigkeit zu eliminieren.
2. **Vorhersagbarkeit:** Systeme sollen zuverlässig und auditierbar bleiben.
3. **Stabilität über Zeit:** Die Bedeutung eines Dokuments darf sich nicht ändern.
4. **Technologische Neutralität:** UDML soll auf beliebigen Systemen interpretierbar sein.
5. **Formale Eleganz:** Die Sprache soll so strukturiert sein, dass sie mathematisch und logisch konsistent bleibt.

Aus diesen Designzielen ergeben sich:

- die reduktionistische Syntax,
- die deterministische Semantik,
- der blockorientierte Aufbau,

- das strikt abgeschlossene Zustandsmodell.

UDML ist damit eine Sprache, die auf Klarheit und Überprüfbarkeit optimiert ist, nicht auf Ausdrucksstärke oder algorithmische Komplexität.

11.5 Strukturelle Konsequenzen

Aus den oben beschriebenen Designprinzipien ergeben sich mehrere fundamentale Konsequenzen für alle Umsetzungen der UDML:

- **Interpretierbarkeit:** Jeder Parser kann ein UDML-Dokument vollständig mechanisch interpretieren.
- **Austauschbarkeit:** Unterschiedliche Ausführungsinstanzen erzeugen denselben Ablauf.
- **Konstanz:** Die Semantik bleibt über alle Versionen hinweg stabil.
- **Komponierbarkeit:** UDML-Blöcke lassen sich beliebig kombinieren, solange die formalen Regeln erfüllt bleiben.
- **Fehlerisolierbarkeit:** Fehler sind lokal auffindbar und betreffen nie das gesamte System außer durch eindeutige strukturelle Verstöße.

Diese strukturellen Konsequenzen machen UDML zu einem zuverlässigen, sicherheitsgeeigneten und langfristig stabilen Modellierungswerkzeug.

12 Anhang

Der Anhang enthält ergänzende Informationen, die für die praktische Verwendung, Implementierung oder Analyse von UDML-Dokumenten hilfreich sind. Er umfasst Begriffserklärungen, tabellarische Übersichten, Referenzinformationen sowie Hinweise zur strukturellen Erweiterung. Diese Inhalte sind nicht Teil der formalen Spezifikation, erleichtern jedoch die Anwendung der Sprache.

12.1 Glossar

BLOCK: Die kleinste vollständige deklarative Einheit in UDML, bestehend aus Header, State, Operatoren, Transitionen und optionalen Hooks.

HEADER: Metadatenbereich eines Blocks, der unveränderliche Informationen enthält.

STATE: Der dynamische Zustandsraum eines Blocks, bestehend aus wohldefinierten, veränderbaren Feldern.

OPERATOR: Eine deterministisch definierte Transformation des Zustands.

TRANSITION: Eine Regel, die bestimmt, wann ein Operator angewendet wird.

HOOK: Ein optionale Ereignisreaktion, die deterministisch einen Operator auslöst.

Zustandsdomäne: Die Menge aller erlaubten Werte für ein Feld im State.

Determinismus: Die Eigenschaft, dass gleiche Eingaben stets zu gleichen Ausgaben führen.

Validität: Die Eigenschaft eines Dokuments, syntaktisch korrekt und semantisch widerspruchsfrei zu sein.

Auditierbarkeit: Die Möglichkeit, jeden Schritt der Ausführung nachvollziehbar und überprüfbar zu rekonstruieren.

12.2 Tabellen der Strukturelemente

Die folgende Tabelle fasst alle primären Strukturelemente der UDML zusammen:

Element	**Kategorie**	**Beschreibung**
BLOCK	Struktur	Übergeordnete Einheit aller Bestandteile
HEADER	Metadaten	Statische Informationen, nicht veränderbar
STATE	Daten	Dynamische Felder eines Blocks
OPERATOR	Logik	Deterministische Transformation eines Zustands
TRANSITION	Ablauf	Regel zur Anwendung eines Operators
HOOK	Ereignis	Reaktion auf definierte Ereignisse

12.3 Operatorreferenzen

Diese Tabelle gibt einen Überblick über die semantische Wirkung der Operator-Konstrukte:

Operatorbestandteil	**Beschreibung**
Zuweisung	Bestimmt explizit die Veränderung eines Zustandsfelds
Parameter	Werte, die als Eingang für Operatorlogik dienen
Regel	Ausdruck, der einen deterministischen Wert erzeugt
Wirkungsbereich	Menge aller STATE-Felder, die ein Operator verändern darf

Die Operatorregeln müssen stets vollständig deterministisch und maschinenverständlich formuliert sein.

12.4 Referenzzustände

Ein UDML-Dokument kann zur Analyse, Testung oder Dokumentation definierte Referenzzustände enthalten. Diese Zustände stellen Beispiele dar, wie ein Block in verschiedenen Ausführungsphasen aussehen kann.

Ein Referenzzustand hat die Form:

```
STATE {
    var1 = <value>
    var2 = <value>
    ...
}
```

Referenzzustände sind nicht Teil der aktiven Semantik eines Dokuments, können aber Strukturprüfungen unterstützen.

12.5 Changelog

Ein Changelog dient der Dokumentation von Weiterentwicklungen ohne Beeinträchtigung der Stabilität bestehender Strukturen. Es folgt üblicherweise dem Muster:

```
Version 1.0:
- Erstveröffentlichung der Spezifikation

Version 1.1:
- Präzisierungen der Operatorregeln
- Ergänzungen der Beispiele

Version 1.2:
- Erweiterung der Grammatik um zusätzliche Ausdrucksformen
```

Die Versionierung folgt dem Prinzip: *Neue Versionen erweitern, aber verändern niemals die Semantik bestehender Konstrukte.*

12.6 Hinweise für Implementierende

Die Implementierung eines UDML-Interpreters oder Validators erfordert besondere Sorgfalt hinsichtlich der deterministischen Natur der Sprache. Folgende Grundprinzipien sollten berücksichtigt werden:

- **Strikte Einhaltung der Grammatik:** Fehlerhafte Dokumente müssen eindeutig abgelehnt werden.

- **Deterministische Evaluierung jeder Expression:** Ausführungslogik darf keine nichtdeterministischen Elemente enthalten.
- **Isolierte Auswertung von Operatoren:** Ein Operator darf nur STATE-Felder beeinflussen.
- **Reihenfolgeunabhängige Interpretation:** Regeln müssen so implementiert werden, dass die Ausführung verschiedener Parser identische Ergebnisse liefert.
- **Klare Fehlerbehandlung:** Fehlerzustände sind strikt zu klassifizieren und deterministisch zu melden.

Implementierende Systeme können erweitert werden, doch die UDML selbst bleibt eine unveränderliche, vollständig deterministische Spezifikation.