Hochschule für Technik und Wirtschaft Dresden (FH)
Fachbereich Informatik/Mathematik

Diplomarbeit

Studiengang Allgemeineine Informatik

Thema: Redesign der Fuzzy-Entwicklungsumgebung FuzzyIDE

eingereicht von: Thomas Kummer, Matr.Nr. 15029
eingereicht am: 27. Februar 2006
Betreuer: Prof. Dr. Heino Iwe

Inhaltsverzeichnis

Danksagung

Hiermit möchte ich mich bei allen bedanken, die zum Gelingen dieser Arbeit beigetragen haben, insbesondere bei Professor Dr. rer. nat. habil. H. Iwe für die Betreuung dieser Arbeit, sowie bei den Mitgliedern des FuzzyIDE Teams.

1. Einleitung

1.1. Thema und Diplomziel

Thema dieser Diplomarbeit ist das Redesign der FuzzyIDE Entwicklungsumgebung. Die Entwicklungsumgebung wurde im Rahmen des Projektseminars von Prof. Dr. Heino Iwe erstellt. Die FuzzyIDE dient dem Erstellen und Testen von Fuzzy-Systemen. Das FuzzyIDE Projekt besteht aus der FuzzyIDE Entwicklungsumgebung und einer Reihe von Plugins. Die Entwicklungsumgebung besitzt eine Mikrokernel Architektur und benötigt Plugins zum Erstellen und Testen eines Fuzzy-Systems. Die FuzzyIDE und die vorhandenen Plugins wurden komplett in Java implementiert, wodurch die FuzzyIDE plattformunabhängig ist. Ziel des FuzzyIDE Projektes ist es, eine Entwicklungsumgebung zu schaffen, welche zu Lehrzwecken kostenfrei eingesetzt werden kann.

Das Projekt soll in dieser Diplomarbeit einem Redesign unterzogen werden, da an einigen Stellen des Projektes Probleme aufgetreten sind. Diese Probleme zu beheben, würde den Rahmen einer Belegarbeit überschreiten. Zudem ist es sinnvoll, in gewissen Bereichen einen neuen Ansatz zur Lösung bestehender Probleme zu wählen. Ziel des Redesigns ist, das Projekt so weit fertig zu stellen, dass es in einer ersten Testversion angewendet werden kann.

1.2. Aufgabenbereiche

Während des Redesigns der FuzzyIDE wird von Mitgliedern des FuzzyIDE Teams an einigen Teilen des Projektes weiter entwickelt. Da aber im Redesign grundlegende Änderungen vorgenommen werden, erfolgt die Anpassung der Ergebnisse der andern FuzzyIDE Teammitglieder erst gegen Ende des Redesigns. Dieses Vorgehen macht es notwendig im Voraus abzuklären welche Teile durch das Redesign betroffen sind.

1.2.1. FuzzyIDE

Da das FuzzyIDE Projekt umfangreich und in einigen Bereichen schon sehr weit fortgeschritten ist, wird das Redesign nicht alle Teile der FuzzyIDE im selben Umfang betreffen. Natürlich sind durch Änderungen in den vom Redesign betroffenen Bereichen auch Anpassungen in den Bereichen notwendig, die eigentlich nicht während des Redesigns betroffen seien sollten. In diesen Abschnitt soll beschrieben werden, welche Teile der FuzzyIDE verändert und welche in der bisherigen Form beibehalten werden.

Betroffene Bereiche

Es wurden für das Redesign die Bereiche ausgewählt, die für die Anwendung der FuzzyIDE unbedingt notwendig sind und die zu Beginn des Redesigns nicht oder nur unvollständig

implementiert waren. Betroffen sind hier zwei der wichtigsten Teile der FuzzyIDE:

- **Datenmodell:** In diesem Teil der FuzzyIDE werden alle Informationen gehalten, die das Fuzzy-System betreffen, welches erstellt und getestet werden soll.
- **Simulator:** Dieser Teil der FuzzyIDE ist dafür zuständig, das im Datenmodell definierte Fuzzy-System in ein funktionierendes System umzusetzen anzuwenden und zu testen.

Durch Änderungen in diesen beiden Teilen sind auch weitergehende Änderungen in folgenden Bereichen zu erwarten: Datenspeicherung, Pluginschnittstelle, Pluginmanager.

Nicht betroffene Bereiche

Nicht oder nur wenig vom Redesign betroffen sind folgende Teile: Grafische Oberfläche, Pluginclient zum Laden von Plugins über das Internet, Pluginserver zum Bereitstellen von Plugins in Internet und das Hilfesystem der FuzzyIDE (siehe Abs. 3.1).

Änderungen durch andere Teammitglieder

Während des Redesigns wird von den FuzzyIDE Teammitgliedern Clemens Altenburger und Romain Schmechta ein Extension Point Mechanismus entwickelt. Dieser soll den bestehenden Pluginmanager der FuzzyIDE ersetzen. Die Ergebnisse dieser Arbeit werden in der Endphase des Redesigns in das FuzzyIDE Projekt eingebunden.

1.2.2. Plugins

Die wichtigsten der bestehenden Plugins werden im Verlauf des Redesigns an die neue Version der FuzzyIDE angepasst. Außerdem wird ein Teil der Funktionalität, die zum Ereichen des Diplomzieles nötig ist, in Form von neuen Plugins realisiert werden.

Parallel zum Redesign der FuzzyIDE arbeiten Teammitglieder des FuzzyIDE Teams noch an einigen Plugins. Die Ergebnisse aus diesen Arbeiten werden nach Beendigung des Redesigns in die FuzzyIDE eingebunden.

1.3. Aufbau

Diese Arbeit besteht aus fünf Teilen:

1. Einführung in die Problematik und allgemeine Informationen über Fuzzylogik sowie den Aufbau des FuzzyIDE Projektes.

2. Ist-Analyse der Teile des FuzzyIDE Projektes, die durch das Redesign betroffen sind. Dieser Teil enthält Informationen über den Aufbau und die Probleme der einzelnen Bereiche. Beschrieben werden das Datenmodell, der Simulator und die vorhandenen Plugins.

3. Anforderungsanalyse zu den Änderungen, die notwendig sind, um das Diplomziel zu erreichen. Die aufgeführten Anforderungen beziehen sich auf die in der Ist-Analyse beschriebenen Teile des FuzzyIDE Projektes sowie die noch zu implementierenden Teile, welche für das Diplomziel benötigt werden.

4. Änderungen und Erweiterungen, die während des Redesigns an dem Projekt vorgenommen wurden. In diesem Teil wird die Umsetzung der einzelnen Vorgaben aus der Anforderungsanalyse beschrieben.

5. Fazit der Arbeit. Dieser Teil enthält die Ergebnisse des Redesigns und die Beschreibung von Funktionen, die noch umzusetzen sind.

Teil I.

Einführung in die Problematik

2. Fuzzy-Set-Theorie und Fuzzy-Systeme

Dieses Kapitel enthält eine kurze Einführung in die Fuzzy-Set-Theorie und ihre Anwendung in wissensbasierten Fuzzy-Systemen. Die Einführung dient dem Verständnis dieser Arbeit und erhebt keinen Anspruch auf Vollständigkeit.

2.1. Grundlagen

2.1.1. Fuzzylogik

Die Fuzzylogik ist eine Verallgemeinerung der klassischen zweiwertigen Logik [Iwe]. In der klassischen Logik kann ein Ausdruck entweder wahr oder falsch sein, Abstufungen zwischen diesen Möglichkeiten sind nicht vorgesehen.

$P(x) \in \{0; 1\}$ Ausdruck P ist wahr oder falsch

Die Fuzzylogik erweitert die klassische Logik um das Element der Unschärfe. Im Gegensatz zur klassischen Logik kann ein fuzzylogischer Ausdruck auch Werte zwischen wahr(=1) und falsch (=0) annehmen. Mit dieser Fähigkeit eignet sich die Fuzzylogik zur Darstellung menschlichen Wissens, das in unscharfen Ausdrücken (ziemlich, sehr, wenig usw.) vorliegt. Die Fuzzylogik kann somit benutzt werden, um unscharfes menschliches Wissen mathematisch zu repräsentieren und in Programme einzubinden.

$P(x) \in [0, 1]$ Ausdruck P(x) liegt im Intervall zwischen 0 und 1

2.1.2. Fuzzy-Menegen

So wie die Fuzzylogik eine Verallgemeinerung der booleschen Logik darstellt, stellen Fuzzy-Mengen eine Verallgemeinerung der klassischen Mengen dar. In der klassischen Mengenlehre wird eine Menge durch das Auswählen von Elementen aus einer Grundgesamtheit(G) gebildet. Die Grundgesamtheit sind dabei alle Elemente, die ausgewählt werden können.

$$A = \{x | Px\} \qquad x \in G$$

A ist die Menge aller Elemente, für welche die Aussage Px wahr ist.

Jede Menge wird über ihre Auswahlfunktion(μ_A) definiert; als Auswahlfunktion werden dabei Ausdrücke der booleschen Logik verwendet. Ein Element kann somit zu einer Menge(A) gehören oder nicht, Abstufungen zwischen diesen Möglichkeiten sind nicht vorgesehen.

$$\mu_A(x) = 1 \quad \Rightarrow x \in A \qquad\qquad \mu_A(x) = 0 \quad \Rightarrow x \notin A$$

Ist der Ausdruck für das Element wahr, gehört es zu Menge. Ist er falsch, gehört das Element nicht zur Menge. Die Menge ist somit eine scharfe Menge, da jedes Element entweder zur Menge gehört oder nicht.

Im Gegensatz zu klassischen Mengen besitzen Fuzzy-Mengen eine fuzzylogische Auswahlfunktion (μ_A), die auch Zugehörigkeitswerte für Elemente zwischen wahr (=1) und falsch (=0) zulässt. Dabei gehören alle Elemente der Grundgesamtheit zu der gebildeten Fuzzy-Menge. Der durch die Zugehörigkeitsfunktion (μ_A) bestimmte Zugehörigkeitsgrad der Elemente schwankt dabei im Intervall zwischen 0 und 1.

$$A = \{(x, \mu_A(x)) | x \in G \wedge \mu_a(x) \in [0, 1]\}$$

Eine Fuzzy-Menge ist somit eine Funktion, welche die Elemente der Grundmenge G auf das Einheitsintervall [0,1] abbildet [Iwe].

2.1.3. Operatoren

Um mit unscharfen Mengen zu rechnen, werden Operatoren benötigt. Die wichtigsten Operationen der klassischen Mengenlehre lassen sich auch für Fuzzy-Mengen umsetzen.

Durchschnitt

Der Durchschnitt zweier Fuzzy-Mengen ist ebenfalls eine Fuzzy-Menge, welche sich wie jede Fuzzy-Menge über ihre Zugehörigkeitsfunktion definiert. Die Zugehörigkeitsfunktion der Durchschnittsmenge ergibt sich aus dem Minimum der Zughörigkeitsfunktionen der Ausgangsmengen für jedes Element.

$$\mathbf{A} \cap \mathbf{B} \qquad \Leftrightarrow \qquad \mu_{A \cap B}(x) = \min(\mu_A(x), \mu_B(x))$$

Vereinigung

Die Zugehörigkeitsfunktion der Vereinigung zweier Fuzzy-Mengen ergibt sich aus dem Maximum der Zugehörigkeitsfunktionen der beiden Ausgangsmengen.

$$\mathbf{A} \cup \mathbf{B} \qquad \Leftrightarrow \qquad \mu_{A \cup B}(x) = \max(\mu_A(x), \mu_B(x))$$

Negation

Die Negation einer Fuzzy-Menge ergibt sich aus dem Komplement der Zugehörigkeitsfunktion der Ausgangsmenge.

$$\overline{A} \qquad \Leftrightarrow \qquad \mu_{\overline{A}}(x) = 1 - \mu_A(x)$$

Zu beachten ist, dass bei der Negation von Fuzzy-Mengen die Komplementgesetze der klassischen Mengenalgebra *keine* Gültigkeit haben.

Prinzip der Widerspruchsfreiheit: $A \cap \overline{A} \neq \emptyset$
Prinzip des ausgeschlossenen Dritten: $A \cup \overline{A} \neq G$

Weitere Operatoren

Neben diesen Grundoperatoren lassen sich in der Fuzzylogik beliebig viele weitere Operatoren definieren, da den Fuzzy-Mengen eine mehrwertige Logik zugrunde liegt. Alle Operatoren für Fuzzy-Mengen lassen sich in zwei Kategorien einteilen: in nichtparametrische und parametrische Operatoren. Parametrische Operatoren besitzen einen oder mehrere Parameter, über die sich das Verhalten des Operators steuern lassen. Beispiele hierfür sind:

Fuzzy-Und: $\mu_{and}(x) = (1-\gamma)\min(\mu_A(X), \mu_B(X)) + \gamma(\frac{1}{2}(\mu_A(X) + \mu_B(X)))$
Fuzzy-Oder: $\mu_{or}(x) = \gamma\max(\mu_A(X), \mu_B(X)) + (1-\gamma)(\frac{1}{2}(\mu_A(X) + \mu_B(X)))$
Min/Max: $\mu_{MinMax}(x) = (1-\gamma)\min(\mu_A(X), \mu_B(X)) + \gamma\max(\mu_A(X), \mu_B(X))$

Über den Parameter γ kann die Arbeitsweise der oben aufgeführten Operatoren verändert werden.

Nicht parametrische Operatoren besitzen im Gegensatz dazu keine Parameter, um ihr Verhalten anzupassen. Beispiele hierfür sind die Grundoperatoren: Vereinigung, Durchschnitt und Negation.

Eine weitere Möglichkeit Operatoren einzuteilen, ist die Klassifizierung nach ihrem Verhalten. Die Operatoren können auf diese Weise eingeteilt werden in:

T-Normen

T-Normen dienen der Bildung von Schnittmengen und entsprechen dem "logischen und". Wichtige Vertreter:

Algebraisches Produkt: $\mu_{A*B}(x) = \mu_A(X) * \mu_B(X)$
Beschränktes Produkt: $\mu_{A\sqcap B}(x) = \min(0, \mu_A(X) + \mu_B(X) - 1)$

S-Normen

S-Normen dienen der Bildung von Vereinigungsmengen und entsprechen dem "logischen inklusive oder".
Wichtige Vertreter:

Algebraische Summe: $\mu_{A+B}(x) = \mu_A(X) + \mu_B(X) - \mu_A(x) * \mu_B(x)$
Beschränkte Summe: $\mu_{A\sqcup B}(x) = \min(1, \mu_A(X) + \mu_B(X))$

Kompensatorische Operatoren

Kompensatorische Operatoren decken den Bereich zwischen Schnittmenge und Vereinigung ab. Wichtige Vertreter:

Arithmetisches Mittel: $\mu_{\frac{1}{2}(A+B)}(x) = \frac{1}{2}(\mu_A(X) + \mu_B(X))$
Geometrisches Mittel: $\mu_{[A*B]\frac{1}{2}}(x) = \sqrt{\mu_A(X) * \mu_B(X)}$

2.1.4. Linguistische Variablen

Linguistische Variablen werden benötigt, um Probleme zu beschreiben, die sich auf Grund ihrer Komplexität schwer mit mathematischen Mitteln beschreiben lassen. Mit Hilfe linguistischer Variablen ist es möglich, menschliches Wissen zu formalisieren und in Fuzzy-Systemen einzubinden. Linguistische Variablen haben im Gegensatz zu konventionellen Variablen keine Zahlen als Werte, sondern Worte oder Ausdrücke einer natürlichen oder künstlichen Sprache. Die Ausprägungen einer linguistischen Variablen werden dabei Terme genannt. Die Menge aller Terme, die eine linguistische Variable annehmen kann, bezeichnet man als linguistische Grundmenge G.

$$L = \{A_1, A_2, \ldots, A_n\}$$
Linguistische Variable L mit A_n Ausprägungen(Termen)

In Fuzzy-Systemen wird jeder Term einer linguistischen Variablen durch eine unscharfe Menge(Fuzzyset) repräsentiert. Alle Terme einer linguistischen Variablen haben dabei die gleiche Basisskala(Grundmenge G). Über die Zuordnung von unscharfen Mengen zu linguistischen Termen wird dem Ausdruck des Terms eine mathematische Größe(Menge) zugewiesen, mit der ein Computer rechnen kann.

Modifikatoren

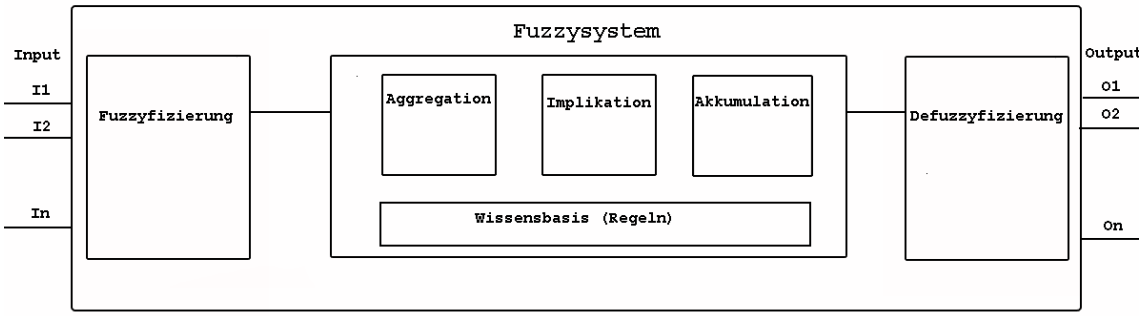
Um bestehende Fuzzy-Mengen anzupassen, welche durch sprachliche Ausdrücke linguistischer Variablen repräsentiert werden, bedient man sich Modifikatoren. Modifikatoren sind Worte oder Wortverbindungen, die einen unären Operator für Fuzzy-Mengen repräsentieren und beschreiben. Zum Beispiel kann dem Modifikator *"sehr"* ein Operator zugeordnet werden, der eine Konzentration der Fuzzy-Menge durchführt. Der Modifikator *"nicht"* dagegen entspricht als Operator einer Negation der Fuzzy-Menge. Die genaue Umsetzung der Modifikatoren in Operatoren ist dabei nicht fest definiert und kann sich unterscheiden.

2.2. Fuzzy-Systeme

Der Aufbau aller Fuzzy-Systeme ist ähnlich und besteht aus drei Teilen: Fuzzyfizierung der Eingangswerte, Inferenz mit Wissensbasis und Defuzzyfizierung der Ausgabemengen.

2.2.1. Fuzzyfizierung

Bevor das Fuzzy-System Berechnungen durchführen kann ist es notwendig, die Eingangsgrößen des Systems von scharfen Werten in Fuzzy-Mengen umzuwandeln. In einem Fuzzy-System wird jede Ein- und Ausgangsgröße durch eine linguistische Variable repräsentiert. Bei der Fuzzyfizierung wird der Übereinstimmungsgrad aller Eingangswerte des Systems mit den Termen(Fuzzy-Mengen) der entsprechenden linguistischen Variablen berechnet. Der



Übereinstimmungsgrad ergibt sich, indem der Eingangswert in die Zugehörigkeitsfunktion der Fuzzy-Mengen der Terme der entsprechenden linguistischen Variablen eingesetzt wird.

2.2.2. Inferenz

Die Inferenz-Einheit des Fuzzy-Systems dient der Berechnung der Output-Fuzzy-Mengen aus den fuzzyfizierten Eingangsgrößen des Systems. Hierzu werden die in der Regelbasis definierten Regeln verwendet.

Regelbasis

In der Regelbasis des Fuzzy-Systems ist das Expertenwissen enthalten, welches durch das Fuzzy-System umgesetzt werden soll. Dazu enthält die Regelbasis *"WENN" - "DANN"* Regeln. Der "WENN"-Teil der Regeln besteht aus einer Reihe von Bedingungen, in welchen die Eingangsgrößen des Fuzzy-Systems mit den Ausprägungen der entsprechenden linguistischen Variablen in Verbindung gesetzt werden. Die einzelnen Bedingungen werden dabei im "WENN"-Teil mit den logischen Operatoren "und" bzw. "oder" verknüpft. Im "DANN"-Teil der Regel wird eine Ausgabegröße mit einem ihrer Terme in Beziehung gesetzt.

Aggregation

In der Aggregation wird der "WENN"-Teil aller Regeln der Regelbasis ausgewertet. Dafür werden die einzelnen Bedingungen des "WENN"-Teils verknüpft und das Gesamtmaß der Übereinstimmung für jede Regel berechnet.

Sicherheitsfaktor

Jede Regel der Regelbasis besitzt einen Vertrauensgrad mit dem das Resultat der Regel in das Gesamtergebnis der Inferenz einfließt. Der Vertrauensgrad liegt im Intervall von [0,1], wobei bei einem Vertrauensgrad von 0 das Ergebnis der Regel nicht in das Endergebnis mit einfließt, während bei einem Vertrauensgrad von 1 das Ergebnis der Regel voll in das Gesamtergebnis eingeht.

Der Sicherheitsfaktor der Regel wird mit dem in der Aggregation bestimmten Erfülltheitsgrad der Regel multipliziert. Das Ergebnis bildet den Grad, in dem der "DANN"-Teil der Regel Anwendung findet.

Implikation

Während des Implikationsschrittes wird der "DANN"-Teil für jede Regel bestimmt. Dazu wird der Anwendungsgrad der Regel mit dem Implikationsoperator verknüpft. Als Implikationsoperatoren werden nichtkompensatorische T-Normen verwendet, da der "DANN"-Teil der Regel nicht richtiger seien kann als der "WENN"-Teil. Das Ergebnis ist eine Fuzzy-Menge, die das Resultat einer Regel repräsentiert.

Akkumulation

Da in Fuzzy-Systemen immer alle Regeln zur Anwendung kommen, werden die im Implikationsschritt berechneten Einzelergebnisse der Regeln während der Akkumulation zum Gesamtergebnis zusammengefasst. Dazu werden für jeden Ausgabewert die im Implikationsschritt berechneten Fuzzy-Mengen der einzelnen Regeln vereinigt. Hierfür werden S-Normen verwendet(z.B. Maximumsoperator, Additionsoperator).

Die beiden in Implikation und Akkumulation verwendeten Operatoren bezeichnet man auch als Inferenz-Strategie. Die gebräuchlichsten Inferenz-Strategien sind: MaxMin, MaxProd, SumProd. Der rechte Operator der Strategie bezieht sich dabei auf den verwendeten Implikationsoperator und der linke Operator auf den benutzten Akkumulationsoperator.

2.2.3. Defuzzyfizierung

Die Defuzzyfizierung ist der letzte Schritt im Fuzzy-System zur Berechnung der Ergebniswerte. Durch die Inferenz-Einheit werden Fuzzy-Mengen für die Outputvariablen gebildet. Die errechneten Fuzzy-Mengen müssen wieder in scharfe Werte umgewandelt werden, um sie außerhalb des Fuzzy-Systems weiter verwenden zu können. Hierzu werden die Outputmengen defuzzyfiziert. Die Defuzzyfizierung kann auf unterschiedliche Weise erfolgen. Am gebräuchlichsten sind dabei die Maximum- und die Momentum-Defuzzyfizierung.

Bei der Maximum-Defuzzyfizierung wird der maximale Zugehörigkeitswert der Outputmenge bestimmt und der zugehörige X-Wert als Ergebnis der Defuzzyfizierung genutzt. Besitzt die Outputmenge mehr als einen maximalen Zugehörigkeitswert, wird je nach Definition entweder der gemittelte, kleinste oder größte X-Wert als Ergebnis verwendet. Diese Methode ist sehr schnell aber auch ungenau.

Bei der Momentum-Defuzzyfizierung(auch COA) wird das Flächenträgheitsmoment der Outputmenge gebildet und als Ergebnis verwendet. Die Berechnungen sind bei dieser Methode umfangreicher, dafür ist das Ergebnis jedoch genauer als bei der Maximum-Defuzzyfizierung.

3. FuzzyIDE

3.1. Architektur

Die FuzzyIDE Entwicklungsumgebung besitzt eine Microkernel-Architektur, ähnlich der Architektur der Entwicklungsumgebung Eclipse. Bei dieser Programmarchitektur wird im eigentlichen Programm nur ein Minimum an Funktionalität implementiert. Funktionen, die über dieses Minimum hinausgehen, werden in Plugins realisiert. Die Plugins werden vom Hauptprogramm geladen und können so zur Laufzeit genutzt werden. Bei dem FuzzyIDE Projekt besitzt das Framework die folgenden Funktionen:

Datenmodell

Das Datenmodell besteht aus einer Reihe von Klassen, deren Aufgabe es ist, die Informationen zu speichern, die das Fuzzy-System definieren. Die Klassen halten Informationen zu den linguistischen Variablen, den Ausprägungen der Variablen und den Regeln des Fuzzy-Systems. Neben der Speicherung der Daten sind die Klassen des Datenmodells dafür verantwortlich, die Informationen in einem konsistenten Zustand zuhalten. Das bedeutet, unzulässiges Verändern der Daten muss unterbunden werden.

Methoden zum Speichern und Laden

Die FuzzyIDE kann ein einmal erstelltes Fuzzy-System sichern und zu einem späteren Zeitpunkt wieder laden. Die Speicherung des Fuzzy-Systems erfolgt in Form von serialisierten Javaobjekten.

Fenstermanagement

Die FuzzyIDE bietet ein Fenstermanagement für die internen Dialoge, welche auf dem Desktop der FuzzyIDE angezeigt werden. Mit Hilfe des Fenstermanagements können mit einem Mausklick alle vorhandenen Dialoge geschlossen oder minimiert werden.

Hilfesystem

Die FuzzyIDE besitzt ein umfangreiches Hilfesystem, welches HTML-Dateien benutzt. Das Hilfesystem sucht beim Start der FuzzyIDE nach einem Unterverzeichnis "help" und lädt alle HTML-Dateien aus diesem Verzeichnis. Beim Laden eines Plugins wird geprüft ob die JAR-Datei des Plugins ein Verzeichnis "help" enthält. Ist dies der Fall, werden alle HTML-Dateien aus diesem Verzeichnis geladen und der Hilfe hinzugefügt.

Pluginclient

Der Pluginclient dient dem Herunterladen von Plugins von einem Server. Der verwendete Server kann in der Datei "fuzzyIDE.properties" im Package de.htwdd.robotic.fuzzyide

festgelegt werden. Der zurzeit verwendete Server zum Verteilen der Plugins läuft auf der iwaps1.informatik.htw-dresden.de Port 5050. Die Quellen des Pluginservers sind im FuzzyIDE Package im Unterverzeichnis "fuzzyServer" vorhanden.

Simulator

Der Simulator der FuzzyIDE dient dem Testen des Fuzzy-Systems. Dazu wird die Umgebung, in der das Fuzzy-System eingesetzt werden soll, entweder simuliert oder eine existierende Umgebung angebunden.

Pluginmanager

Ohne Plugins zum Bearbeiten des Datenmodells oder zum Beschaffen der Inputwerte bzw. zum Auswerten der Outputwerte des Simulators, ist die FuzzyIDE nicht sinnvoll einsetzbar. Die Plugins sind über ein Interface de.htwdd.robotic.fuzzyide.Plugable (Pluginschnittstelle) mit der FuzzyIDE verbunden.

Beim Start der FuzzyIDE, werden die Plugins vom Pluginmanager geladen und in die Menübar der FuzzyIDE eingebunden. Im Normalfall sucht die FuzzyIDE beim Start im Unterverzeichnis "plugins" nach Plugins. Allerdings kann auch beim Start der FuzzyIDE der Parameter "-p" mit einem alternativen Verzeichnis angegeben werden.

Die Plugins werden abhängig von ihren Aufgaben in die Menübar der FuzzyIDE eingebunden. Alle Plugins, die den Simulator der FuzzyIDE nutzen, werden im Menü "Systems & Simulators" eingefügt. Nutzt ein Plugin das Datenmodell der FuzzyIDE, wird es im Menu "Tools" eingeordnet. Plugins, die weder das Datenmodell noch den Simulator nutzen, werden im Menü "Plugins" abgelegt.

3.2. NRC-Bibliothek

Die kanadische NRC-Bibliothek [NRC] "FuzzyJ" bildet den Kern der FuzzyIDE. Mit Hilfe dieser Bibliothek werden alle fuzzylogischen Berechnungen in der FuzzyIDE ausgeführt. Die FuzzyIDE kapselt die Funktionalität der NRC-Bibliothek und bietet die Möglichkeit, mit Hilfe einer grafischen Oberfläche auf die Funktionen der NRC-Bibliothek zuzugreifen.

Die NRC-Bibliothek bietet alle notwendigen Funktionen zum Erstellen und Benutzen eines Fuzzy-Systems. Ein Grossteil der NRC-Klassen repräsentiert ein Element des Fuzzy-Systems. Ein Objekt der NRC-Klasse *nrc.fuzzy.FuzzyVariable* stellt zum Beispiel eine linguistische Variable des Fuzzy-Systems dar. Ebenso existieren Klassen für Regeln und Terme(Ausprägungen). Darüber hinaus gibt es noch Klassen, die für Berechnungen benötigt werden. Die wichtigste dieser Klassen ist die Klasse *nrc.fuzzy.FuzzyValue*. Sie beschreibt eine Fuzzy-Menge. Mit Hilfe dieser Klasse können Fuzzy-Mengen über Operatoren verknüpft werden. Sie bietet zudem Methoden, um die Fuzzy-Menge zu defuzzyfizieren.

Die NRC-Bibliothek zeichnet sich im Vergleich zu anderen Bibliotheken durch ihren großen Funktionsumfang aus. Besonders sind hier die Expressionstrings zu erwähnen. Diese Strings können Ausdrücke enthalten, die aus mehrere Modifikatoren, Fuzzysetnamen und Klammerungen bestehen. Der Ausdruck eines Expressionstrings wird durch die NRC-Bibliothek in eine Fuzzy-Menge umgewandelt, die im Fuzzy-System verwendet werden kann. Mit Hilfe der Expressionstrings können sehr komplexe Bedingungen und Schlüsse in den Regeln des Fuzzy-Systems modelliert werden.

Ein weiterer Unterschied zu den meisten Bibliotheken ist die große Anzahl von unterstützten Fuzzysettypen. Es existieren neben der Klasse "FuzzySet" noch 10 abgeleitete Klassen zum Erstellen spezieller Fuzzysets(S-Fuzzyset, Pi-Fuzzyset usw.).

In Verbindung mit weiteren Funktionen macht dies die NRC-Bibliothek zu einem sehr mächtigen Werkzeug zum Erstellen von Fuzzy-Systemen. Die Bibliothek steht zwar nicht unter einer Open Source Lizenz, ist aber für Lehrzwecke kostenlos erhältlich.

3.3. Projektmanagement

3.3.1. CVS

Das gesamte FuzzyIDE Projekt mit allen zugehörigen Plugins ist in einem CVS-Paket organisiert. Mit Hilfe von CVS ist es möglich, mehrere Entwickler parallel an einem Projekt arbeiten zu lassen. Darüber hinaus bietet CVS die Möglichkeit, alte Entwicklungsstände abzurufen und so den Verlauf der Entwicklung zu verfolgen.
Das gesamte Projekt befindet sich im Paket *fuzzyIDE-package* auf dem Server *iwaps1.informatik.htw-dresden.de*. Die CVS Zugangsdaten zum Abrufen ("checkout") des Projekts sind wie folgt:

Server:	iwaps1.informatik.htw-dresden.de
Paketname:	fuzzyIDE-package
Authentifizierung:	ssh
CVS Root:	/home/cvsroot

Um das Paket über CVS unter Linux oder UNIX abzurufen ist es sinnvoll, die Variablen CVSROOT und CVS_RSH in der *.bachrc* zu setzen.

export CVSROOT=:ext:s@iwaps1.informatik.htw-dresden.de:/home/cvsroot**
export CVS_RSH=ssh

Ist dies getan, kann das Paket einfach über folgenden Befehl auf den eigenen Rechner geholt werden:

cvs checkout fuzzyIDE-package

Wurde der Befehl erfolgreich ausgeführt, existiert danach im aktuellen Verzeichnis ein neues Unterverzeichnis mit dem Namen *fuzzyIDE-package*, welches das gesamte Projekt enthält.

3.3.2. Verzeichnisse

Das Hauptverzeichnis *fuzzyIDE-package* enthält folgende Unterverzeichnisse und Dateien:

- **CVS** Dieses Unterverzeichnis enthält die CVS-Informationen über das aktuelle Verzeichnis. Die Informationen sind für das CVS Versionsmanagement notwendig. Sie haben keinen direkten Bezug zum FuzzyIDE Projekt.
- **dev_framework:** enthält die Quellen der FuzzyIDE Entwicklungsumgebung

- **dev_plugins:** enthält die Quellen aller Plugins, wobei für jedes Plugin ein eigenes Unterverzeichnis vorhanden ist
- **distribution:** In diesem Verzeichnis befindet sich die aktuelle Version der FuzzyIDE Entwicklungsumgebung mit allen funktionierenden Plugins in kompilierter Form.
- **doc:** enthält allgemeine Informationen und Dokumentationen zum Projekt
- **doc_internal:** enthält Vorlagen zum Erstellen einer Dokumentation
- **doc_org:** enthält Dokumente über Planungen des Projektes und einige Sourcetemplets, die von Entwicklern benutzt werden können
- **doc_spec:** enthält die Spezifikationen des Projektes (Architektur, verwendete Java Version usw.)
- **fuzzyJ-Toolkit:** enthält die URL zum Herunterladen der NRC-Klassenbibliothek.
- **fuzzyModel:** enthält Dokumentationsdateien des veralteten XML Datenmodells
- **fuzzyServer:** enthält den Pluginserver, über den die Entwicklungsumgebung Plugins abrufen kann
- **logo:** enthält das Splashscreen Bild und die Icons für die Entwicklungsumgebung
- **web:** enthält Skripte zum Versionsmanagement
- **hosting_informations.txt:** enthält Informationen über den CVS Zugang und Server
- **members.txt:** enthält Informationen zu den Mitgliedern der FuzzyIDE Gruppe (Name, Aufgaben, E-Mail Adresse usw.)
- **readme.txt:** enthält Informationen über die Verzeichnisstruktur

3.3.3. Kompilieren

Im FuzzyIDE Paket existieren eine Reihe von Buildfiles zum Kompilieren der Quellen. Um die Quellen zu übersetzen, wird das Open Source Tool "Ant" benötigt [ANT]. Für die FuzzyIDE und nahezu jedes Plugin existiert eine Datei "build.xml". Diese befindet sich im Wurzelverzeichnis des Plugins bzw. der FuzzyIDE. Um die Quellen der FuzzyIDE oder eines Plugins zu kompilieren, muss in das Verzeichnis gewechselt werden, in dem die "build.xml" Datei liegt und das Kommando *"ant"* aufgerufen werden. Bei einem Plugin Buildfile werden die Quellen des Plugins kompiliert und als Jar-File im Unterordner *dist* des Plugin-Ordners abgelegt.

Das Buildfile der FuzzyIDE übersetzt die Quellen und legt eine Verzeichnisstruktur für die FuzzyIDE an. Bei zusätzlicher Angabe des Ziels *"dist"* werden die Jar-Files aus dem *dev_plulgins* Ordner kopiert und unter *dev_framework/build/dist/plugins* abgelegt. Nach erfolgreichem Abarbeiten des Buildfiles befindet sich eine einsetzbare Version der FuzzyIDE im Ordner *dev_framework/buld/dist*. Diese kann dann mit dem Kommando *"java -jar fuzzyide.jar"* gestartet werden.

4. Fuzzy-Entwicklungsumgebungen

Fuzzy-Systeme werden in immer mehr Bereichen eingesetzt. Aus diesem Grunde werden auch Entwicklungsumgebungen zum Erstellen von Fuzzy-Systemen benötigt. Bestehende Entwicklungsumgebungen sind meist kommerziellen Ursprungs und bieten umfangreiche Funktionen, die über das Erstellen eines Fuzzy-Systems hinausgehen. Folgende Tabelle stellt einen kleinen Überblick über häufig verwendete Fuzzy-Entwicklungsumgebungen dar.

Name	**DataEngine**	**fuzzyTECH**	**CubiCalc**	**mbFuzzIT**
OS	Windows	Windows	Windows	Windows; Unix/Linux
Code-generation	über Modul(ADL); C C++; Dll (Windows);	C ;Assembler; Java; Cobol	über CubiCalc RTC Dll (Windows); C	keine
Plugin-schnittstelle	ja	ja	nein	nein
Fuzzyset-typen	frei definierbar	frei definierbar	frei definierbar keine S,Z oder Pi Fuzzysets; keine Singelton	frei definierbar keine S,Z oder Pi Fuzzysets; keine Singelton
Inferenz-methoden	MaxMin; MaxProd; SumProd ...	MaxMin;MaxProd; SumProd ...	MaxMin;MaxProd; SumProd ...	MaxMin
Defuzzy-fiezirung	Maxum,COA	Maxum,COA	Maxum,COA	Maxum,COA
DDE Anbinding	Nein	Ja	ja	nein
Regel-visualisierung	Ja	Ja	ja	nein
Sonstiges	NeuroFuzzy Systeme; statistische Funktionen ; grafische Makrosprache	speziele Editionen für Zielplattformen; NeuroFuzzy Systeme; optimierte Codeerzeugung für μController		Open Source; kostenlos; mehrfache Regelbasen

Wie in der Tabelle zu erkennen, sind die Entwicklungsumgebungen DataEngine und fuzzyTECH sehr umfangreich und daher auch kostspielig. Die CubiCalc Entwicklungsumgebung ist schon etwas älter(1993) und bietet im Vergleich zu den oben genannten Entwicklungsumgebungen einen geringeren Funktionsumfang. Auf Grund des einfachen Aufbaus der CubiCalc Entwicklungsumgebung [Cub93] eignete sie sich aber in vielen Bereichen als Vorbild für die FuzzyIDE. Auch während des Redesigns wurde an einigen Stellen die CubiCalc Entwicklungsumgebung als Vorlage für Erweiterungen an der FuzzyIDE verwendet.

Die Entwicklungsumgebung *mbFuzzIT* ist eine der wenigen nicht kommerziellen Entwicklungsumgebungen für Fuzzy-Systeme und wurde aus diesem Grunde mit in der Tabelle aufgeführt. Das Programm befindet sich noch in der Entwicklungsphase und eignet sich noch nicht zum Erstellen von Fuzzy-Systemen.

Die FuzzyIDE in diese Tabelle einzuordnen, ist mit dem Ausgangstand des Projektes nicht möglich, da in dieser Version Teile der FuzzyIDE nicht funktionieren. Zu erwähnen ist aber, dass die FuzzyIDE plattformunabhängig ist, Plugins benutzt, keine Codegenerationsmethoden besitzt, eine DDE Anbindung nicht existiert und keine Regeln visualisieren kann.

Teil II.

Ist-Analyse

5. Datenmodell

5.1. Einleitung

Das Datenmodell ist einer der wichtigsten Teile der FuzzyIDE. Hier werden die Daten gehalten, welche das Fuzzy-System definieren. Die Änderungen am Datenmodell bilden die Voraussetzung für das Redesign der anderen Teile der FuzzyIDE, da diese das Datenmodell benutzen. In diesem Kapitel werden der Aufbau und die Probleme des bestehenden Datenmodells beschrieben. Die Beschreibung ist dabei auf den prinzipiellen Aufbau und die daraus resultierenden Probleme begrenzt.

5.2. Aufbau

Die Daten des Fuzzy-Systems werden in Java-Objekten gehalten. Die Objekte sind hierarchisch in Form eines Baumes geordnet. Der genaue Aufbau des Datenmodells ist im UML-Klassendiagramm in Abbildung 5.1 zu sehen.

5.2.1. Klassen

Dieser Abschnitt enthält eine Beschreibung zu den wichtigsten Klassen des Datenmodells. Die Beschreibung beschränkt sich dabei auf solche Klassen, die Daten halten oder dafür von Bedeutung sind.

FuzzyElement:
Ist die abstrakte Basisklasse, von der alle Klassen des Datenmodells abgeleitet sind, welche Daten halten. Die Klasse ist abgeleitet von *java.util.Observable* und bietet somit den von ihr abgeleiteten Klassen die Möglichkeit, sich bei Observern zu registrieren (siehe Abs. 5.2.2). Die einzige Methode, welche die FuzzyElement Klasse zur Verfügung stellt, ist die Methode *"clone"* zum Anlegen einer tiefen Kopie des Objektes.

FuzzySystem:
Die FuzzySystem Klasse bildet die Wurzel des Objektbaums, in dem die Daten des Fuzzy-Systems organisiert sind. Die Klasse enthält den Namen des Fuzzy-Systems sowie ein FuzzyVariableBase Objekt und ein FuzzyRuleBase Objekt. Als Methoden enthält die Klasse nur "get" und "set" Funktionen für den Namen und die beiden Base Objekte.

FuzzyVariableBase:
Diese Klasse hält Referenzen zu allen linguistischen Variablen des Fuzzy-Systems sowie Methoden, um die Variablen zu verwalten (Zugriff, Klonen, Anlegen, Löschen usw.).

FuzzyRuleBase:
Die Klasse hält Referenzen zu allen Regeln des Fuzzy-Systems sowie Methoden, um die

Regeln zu verwalten (Zugriff, Klonen, Anlegen, Löschen usw.).

FuzzyVariable:
Ein Objekt dieser Klasse des Datenmodells hält alle Informationen zu einer linguistischen Variablen. Über die Methoden der Klasse ist es Möglich, der linguistischen Variablen Ausprägungen(FuzzySets) hinzuzufügen und diese zu verwalten (Zugriff, Entfernen, Klonen usw.).

FuzzySet:
Ein Objekt Klasse FuzzySet enthält die Daten einer Ausprägung also auch die Definition der Fuzzy-Menge, welche sie repräsentiert (siehe Abs. 2.1.4). Die Ausprägung ist Teil einer linguistischen Variablen. Dabei kapselt die Klasse alle Ausprägungstypen(Fuzzysettypen), die von der NRC-Bibliothek unterstützt werden. Zudem bietet die Klasse Funktionen, um die einzelnen Punkte der Ausprägung(FuzzySet) zu verändern.

FuzzySetPoint:
Die Klasse FuzzySetPoint repräsentiert einen Punkt eines Fuzzysets und enthält den X- und Y-Wert des Punktes und die Methoden, um auf die beiden Werte zuzugreifen.

FuzzyRule:
Ein FuzzyRule Objekt enthält eine Regel des Fuzzy-Systems. Über die Methoden der Klasse können die Komponenten der Regel verändert werden.

FuzzyValue:
Die Klasse definiert eine Fuzzy-Menge. Die Daten bestehen aus einer linguistischen Variablen (FuzzyVariable) und einer Expression(String).

Die Expression enthält einen Ausdruck, welche aus einer Reihe von Operatoren (and, or), Modifikatoren (siehe Abs. 2.1.4), Klammerungen und Namen von Fuzzysets bestehen kann. Die verwendeten Fuzzysetnamen müssen in der linguistischen Variablen als Ausprägungen vorhanden sein. Der Ausdruck enthält aber mindestens den Namen eines Fuzzysets aus der linguistischen Variablen. Damit ein Ausdruck gültig ist, muss er der Grammatik der NRC-Expressionstrings entsprechen [NRC]. Objekte der Klasse FuzzyValue werden benötigt, um den "WENN"- und "DANN"-Teil der Regeln des Fuzzy-Systems zu modellieren.

5.2.2. Observer Mechanismus

Das Datenmodell besitzt eine Reihe von Observern [Ess02], die benutzt werden, um Änderungen im Datenmodell allen Teilen des Systems bekannt zu geben, die das Datenmodell verwenden. Dies sind der Simulator sowie alle Plugins, die das Interface *de.htwdd.robotic.fuzzyide.plugman.DataModelAccessable* implementieren. Die Observer sind so aufgebaut, dass sie bei einer Änderung an einem Objekt die jeweils nächst höhere Instanz im Objektbaum des Datenmodells benachrichtigen. Wird beispielsweise an einem "FuzzySet" Objekt eine Änderung vorgenommen, so benachrichtigt es das "FuzzyVariable" Objekt, zu dem es gehört.

Das "FuzzyVariable" Objekt wiederum benachrichtigt über einen Observer das "FuzzyVariableBase" Objekt und so weiter. Auf diese Weise wird die Änderung all jenen Teilen der FuzzyIDE bekannt gegeben, die das Datenmodell benutzen.

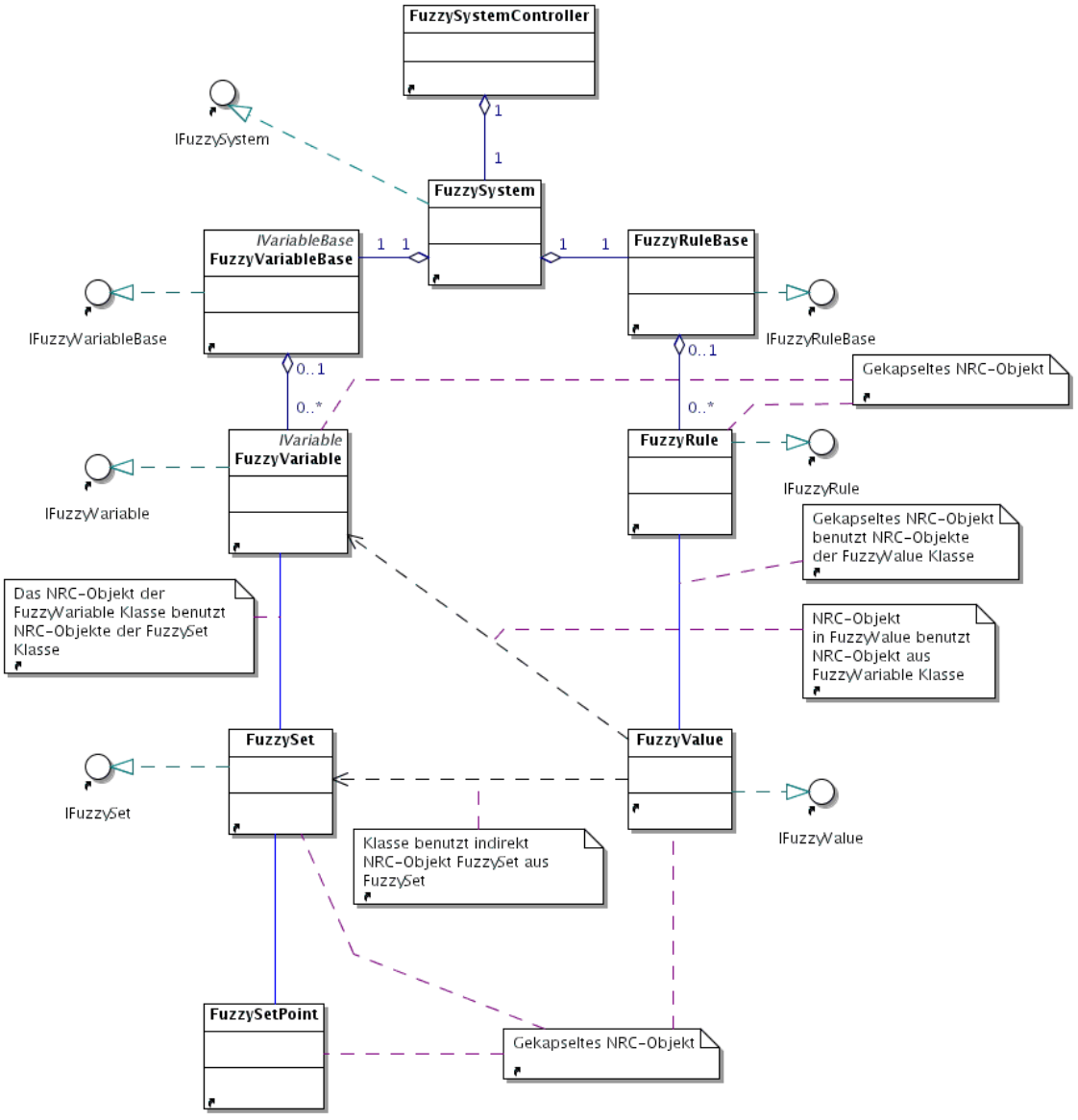


Abbildung 5.1.: Bestehendes Datenmodell

5.2.3. Datenhaltung

Wie im UML-Diagramm 5.2 zu sehen ist, kapselt jede Klasse des Datenmodells, die ein Element des Fuzzy-Systems(linguistische Variable, Regel usw.) repräsentiert, ein Objekt der NRC-Klassenbibliothek. So kapselt zum Beispiel die Klasse *fuzzymodel.FuzzyVariable* ein Objekt der Klasse *nrc.fuzzy.FuzzyVariable*.

Die Klassen des Datenmodells benutzt das privat deklarierte NRC-Objekt [NRC] zur Haltung der meisten Daten.

Die Klassen des Datenmodells nutzen darüber hinaus in ihren Methoden meist die Methoden der NRC-Objekte, um ihre Funktionalität umzusetzen.

Soll zum Beispiel ein Fuzzyset zu einer Fuzzy-Variablen des Datenmodells hinzugefügt werden, muss die Methode *addFuzzySet(de.htwdd.robotic.fuzzyide.fuzzymodel.FuzzySet set, String name)* der FuzzyVariable Klasse des Datenmodells aufgerufen werden. Diese Methode wiederum ruft die Methode *fuzzyVariable.addTerm(String name, nrc.fuzzy.FuzzySet set);* des gekapselten NRC-Objektes auf. Dabei wird ein Teil der Exceptions der NRC-Methode bearbeitet, während andere Exceptions weiter geleitet werden.

Die Methoden der Datenmodellklassen benötigen als Parameter Objekte der Datenmodellklassen. Die Methoden der gekapselten NRC-Objekte benötigen jedoch als Parameter NRC-Objekte. Aus diesem Grund ist es notwendig, die privaten NRC-Objekte anderen Klassen zugänglich zu machen. Dies geschieht über die Methode *getNRC()*, welche alle Klassen implementieren die NRC-Objekte besitzen (siehe Abb. 5.2). Der Quelltext der Methode *addFuzzySet* der Klasse de.htwdd.robotic.fuzzyide.FuzzyVariable verdeutlicht das Vorgehen:

```
public void addFuzzySet(FuzzySet set, String name)
        throws
        FuzzySetOutOfBoundsException,//nicht behandelte Exceptions(nrc)
        FuzzyException,
        InvalidFuzzySetMethodcallException
{
        set.setName(name);
        try {
             fuzzyVariable.addTerm(
             set.getName(),
             set.getNRC());//Zugriff auf gekapseltes NRC-Objekt
             setChanged();
        }
        catch (XValueOutsideUODException e) {//behandelte Exception(nrc)
             e.printStackTrace();
        }
}
```

In dem Quelltext ist zu erkennen, dass die "addFuzzySet" Methode des Datenmodellobjektes die Funktionalität der NRC-Methode "addTerm" kapselt. Da diese Methode aber als Parameter ein Objekt vom Typ nrc.fuzzy.FuzzySet benötigt, muss auf das privat deklarierte NRC-Objekt des Parameters "set" zugegriffen werden. Ähnlich wie in dieser Methode

ist der Aufbau der meisten Methoden der Datenmodellklassen. Die Methoden kapseln die Funktionen ihres NRC-Objektes.

Die Methode "getNRC" ist also zwingend notwendig für alle Klassen, die NRC-Objekte zur Haltung von Daten benutzen. Die Methode stellt dabei eine Verletzung des Klassenkonzeptes dar, da sie direkten Zugriff auf einen privaten Member erlaubt.

5.3. Probleme

Auf Grund der in Abschnitt 5.2.3 beschriebenen Art der Datenhaltung kommt es zu Inkonsistenzen im Datenmodell, die in den folgenden Abschnitten anhand einiger exemplarischer Beispiele beschrieben werden. Die gewählten Beispiele stehen für jeweils eine Art von Fehlern, die im Datenmodell an mehreren Stellen auftreten.

5.3.1. Fehlerhafte Referenzen

Fehlerhafte Referenzen stellen ein schwerwiegendes Problem dar, da durch sie die Eindeutigkeit des Datenmodells nicht mehr gegeben ist. Sie treten im Zusammenhang mit den gekapselten NRC-Objekten des Datenmodells auf (siehe Abs. 5.2.3). Die privat deklarierten NRC-Objekte, in denen die meisten Daten des Datenmodells gespeichert sind, werden von den Methoden anderer NRC-Objekte im Datenmodell benötigt. In vielen Fällen halten die NRC-Objekte Referenzen auf die privat deklarierten NRC-Objekte anderer Datenmodellobjekte. In einigen Klassen des Datenmodells ist es aber notwendig, bei Aufruf bestimmter Methoden, ein neues NRC-Objekt zur Datenhaltung anzulegen. Andere NRC-Objekte halten nach dem Anlegen des neuen Objektes aber noch Referenzen auf das ursprüngliche Objekt, welches nicht mehr aktuell ist. Wegen dieser fehlerhaften Referenzen auf veraltete Objekte sind alle Änderungen an dem neuen Objekt den Objekten unbekannt, die noch eine Referenz auf das ursprüngliche Objekt halten. Folgendes Beispiel verdeutlicht das Problem:

Bei einen Objekt der Datenmodell Klasse *FuzzyVariable* wird die Methode *setName(String name)* aufgerufen. Die linguistische Variable, welche das FuzzyVariable Objekt beschreibt, wird dabei schon in einer Regel des Fuzzy-Systems verwendet. Die Abbildung 5.2 zeigt die Referenzen zwischen den einzelnen Objekten vor (links) und nach (rechts) dem Aufruf der Methode *setName(String name)*. Die Objekte der FuzzyIDE Klassen sind mit "(IDE)" und die der NRC-Bibliothek mit "(NRC)" gekennzeichnet. Wie man auf der rechten Seite sieht, ist nach dem Aufruf der Funktion *setName(String name)* ein neues NRC FuzzyVariable Objekt vorhanden. Das Erzeugen eines neuen NRC-Objektes lässt sich nicht vermeiden, da der Name der Variablen im NRC-Objekt gespeichert wird und dieses keine Methode zum Ändern des Namens besitzt. Sollten nun nach dem Aufruf der *setName(String name)* Methode Änderungen an der Fuzzy-Variablen vorgenommen werden, so sind diese in der Fuzzy-Regel nicht bekannt, da die Regel noch die Referenz auf das altes Objekt hält. Noch problematischer ist es, sollte eine in Regeln verwendete Fuzzy-Variable aus dem Datenmodell gelöscht werden. Das FuzzyVariable Objekt wird aus dem Objektbaum des Datenmodells entfernt. Das gekapselte NRC-Objekt bleibt jedoch vorhanden, da FuzzyRule Objekte noch Referenzen darauf halten.

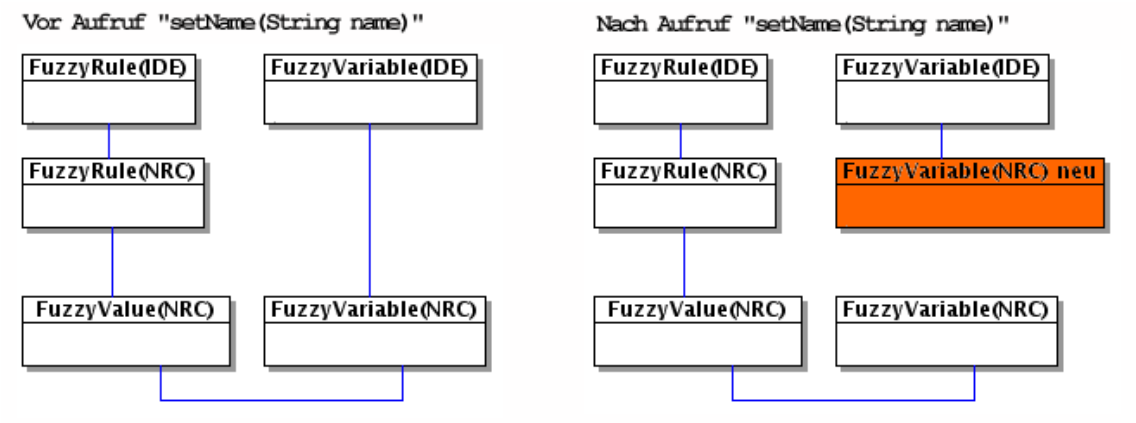


Abbildung 5.2.: Referenzfehler im bestehenden Datenmodell

Die FuzzyRule Objekte benutzen in diesem Fall eine linguistische Variable, die nicht mehr im Datenmodell bekannt ist. Die Regel erwartet also ein Input oder Output, der nicht mehr bekannt ist.

5.3.2. Doppelte Datenhaltung

In einigen Teilen des Datenmodells kommt es zu doppelter Datenhaltung, da nicht alle Informationen in NRC-Objekten gehalten werden können. So besitzt beispielsweise die Klasse *de.htwdd.robotic.fuzzyide.fuzzymodel.FuzzySet* einen privat deklarierten Member nrc.fuzzy.FuzzySet, der die Informationen des Sets hält und es gibt ein ebenfalls privat deklariertes String Objekt, welches den Namen des Sets hält. Dies ist notwendig, da der Name nicht im NRC-Objekt gespeichert werden kann. Sollte das Set aber einer Variablen hinzugefügt werden, übergibt der Konstruktor der Datenmodellklasse die beiden Member dem Konstruktor des NRC-Objektes der Variablen. Wird nach Hinzufügen des Sets zur Variablen der Name des Fuzzysets(im Datenmodell FuzzySet Objekt) geändert, ist der neue Name des Sets der Fuzzy-Variablen nicht bekannt (siehe Abb. 5.3).

5.3.3. FuzzySet Klasse

Im Datenmodell werden die Informationen zu den Fuzzysets in einer Klasse (FuzzySet) gehalten, unabhängig vom Typ des Fuzzysets. Dies kann zu einer Reihe von Problemen führen. So besitzt die Klasse beispielsweise eine Methode "removePoint(int i)", um einen Punkt aus dem FuzzySet Objekt zu entfernen. Wird diese Methode bei einem Fuzzyset vom Typ "TriangleFuzzySet" aufgerufen und ein Punkt entfernt, so handelt es sich bei diesem Fuzzyset nicht mehr um ein gültiges Fuzzyset vom Typ "TriangleFuzzySet" mehr, da dieser Typ zwingend drei Punkte benötigt(siehe Abb. D). Das gleiche Problem tritt bei den Methoden "setPoints(Vector points)" und "appendPoint(FuzzySetPoint point)" auf.

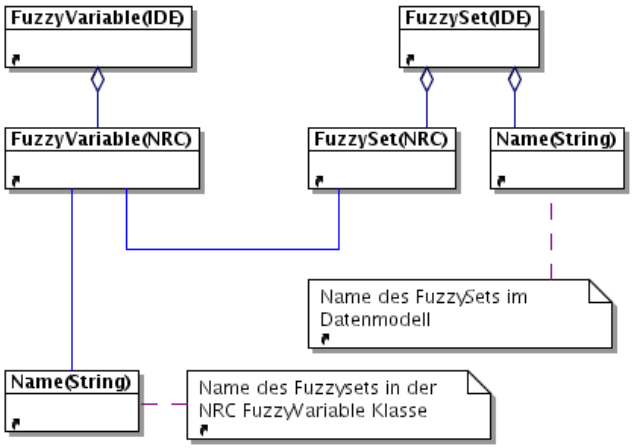


Abbildung 5.3.: Doppelte Datenhaltung im bestehenden Datenmodell

5.4. Ausstehende Implementierung

Einige Teile des Datenmodells sind noch nicht vollständig implementiert. So existieren zwar die Methoden zum Setzen und Abfragen des Sicherheitsfaktors(Certainty) in der Klasse *FuzzyRule*, sie sind jedoch noch nicht mit der nötigen Funktionalität versehen. Darüber hinaus unterstützt das Datenmodell noch nicht alle Fuzzysettypen der NRC-Bibliothek vollständig.

6. Simulator

Der Simulator der FuzzyIDE wird benötigt, um ein angelegtes Fuzzy-System zu testen. Ein Fuzzy-System bildet gewöhnlich nur einen Teil eines Programms. Aus diesem Grunde ist es notwendig, die Umgebung zu simulieren, in der das Fuzzy-System genutzt werden soll, um das System zu testen.

Der Simulator der FuzzyIDE kann entweder ein existierendes System ansteuern, welches das Fuzzy-System mit Inputwerten versorgt und die Outputwerte weiterverarbeitet oder er kann ein solches System simulierten. Der bestehende Simulator der FuzzyIDE befindet sich im Package "calculation". Er baut auf dem bestehenden Datenmodell der FuzzyIDE auf und wird in diesem Kapitel näher beschrieben.

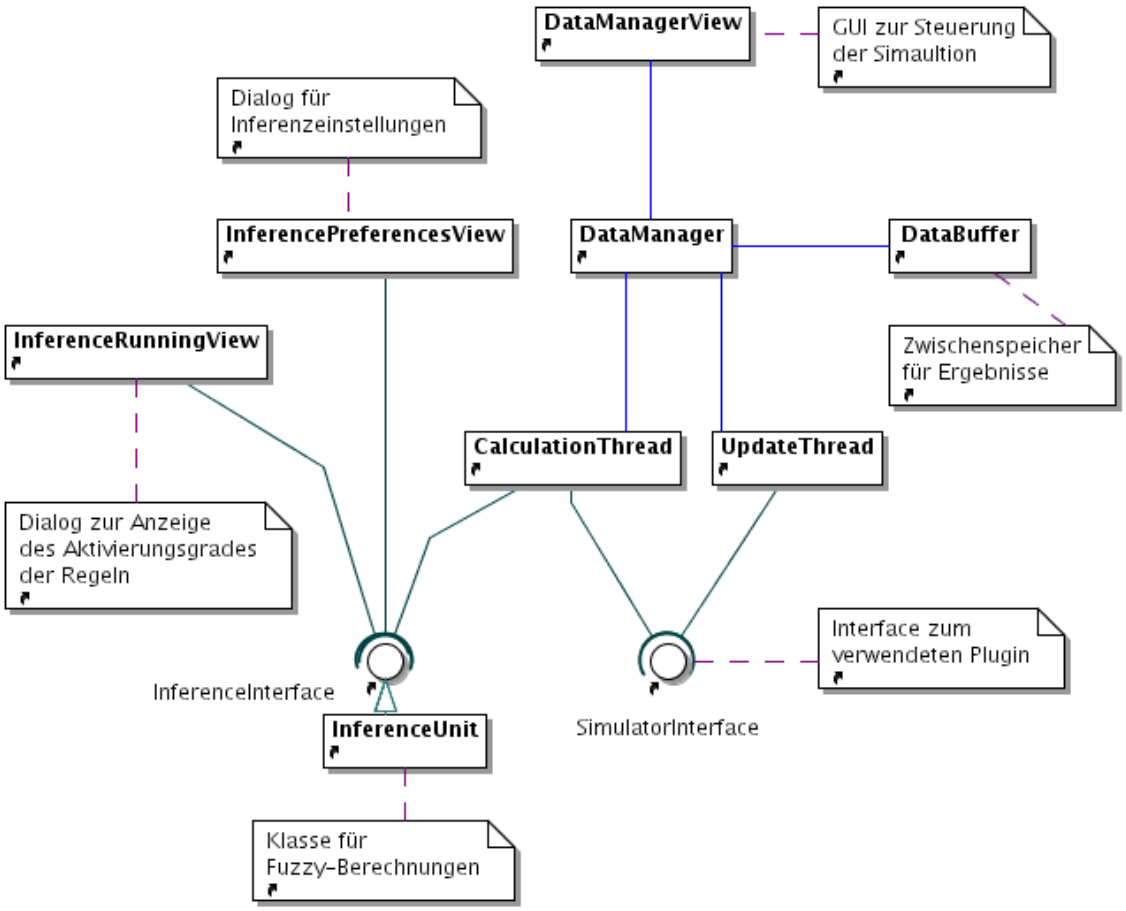


Abbildung 6.1.: Klassendiagramm des bestehenden Simulators

6.1. Aufbau

6.1.1. Pluginanbindung

Der Simulator benutzt Plugins, um dem Fuzzy-System eine Programmumgebung zu simulieren. Die Plugins werden zum Bereitstellen des Inputs und Auswerten des Outputs des Fuzzy-Systems verwendet. Der Simulator benutzt während einer Simulation immer nur *ein* Plugin. Dieses ist für die Beschaffung der Inputwerte und für das Auswerten der Outputwerte zuständig. Während einer Simulation ist es nicht möglich, mehrere Plugins zu nutzen und die anfallenden Aufgaben auf die Plugins zu verteilen. Das verwendete Plugin ist über das Interface *"SimulatorInterface"* mit dem Simulator verbunden. Alle Plugins, die dieses Interface implementieren, werden als Simulatorplugins erkannt und beim Laden im Menüeintrag "Systems & Simulators" eingeordnet.
Das Interface besitzt vier Methoden:

- **public void startSimulation();**
  Diese Methode des Plugins wird beim Start der Simulation vom Simulator gerufen. Hier kann das Plugin Initialisierungsarbeiten durchführen.
- **public void stopSimulation();**
  Die Funktion bietet dem Plugin die Möglichkeit, Aufräumarbeiten durchzuführen. Sie wird vom Simulator beim Beenden der Simulation aufgerufen.
- **public void calculateNextState(final long milliseconds, final double [] lastInferenceResults);**
  Diese Methode des Plugins wird durch den Simulator gerufen und übergibt als Parameter die Ergebnisse des letzten Berechnungsschrittes.
- **public double [] returnCurrentState();**
  Diese Methode gibt den vom Plugin berechneten Input an das Fuzzy-System im Simulator.

Die Übergabe von Inputwerten und die Rückgabe von Ergebnissen erfolgt in Form von "Double" Vektoren.

6.1.2. Threads

Der Simulator benutzt während der Simulation zwei Threads. Einen Thread zum Berechnen der Werte des Fuzzy-Systems und einen anderen Thread zur Aktualisierung der Ausgaben des Plugins. Der Thread zur Berechnung der Ergebnisse des Fuzzy-System ruft nach den Berechnungen die "calculateNextState" Methode des Plugins auf. Als Parameter werden die Ergebnisse des Fuzzy-Systems und die Zeit seit Beginn der Simulation übergeben. Der andere Thread des Simulators ruft die "returnCurrentState" Methode des Plugins auf. Diese gibt die Ergebnisse des Plugins als Input an das Fuzzy-System des Simulators zurück. Der Simulator bietet die Möglichkeit für beide Threads, den Zeitraum zwischen den Berechnungsschritten festzulegen. Um die "calculateNextState" und die "returnCurrentState" Methode synchron zu nutzen, gibt es die Möglichkeit, beide Methoden(calculateNextState, returnCurrentState) nacheinander in einen Thread zu rufen. Dazu muss im "Simulation Control" Dialog die Checkbox "synchronized" aktiviert werden. Ist dies geschehen, werden beide Methoden im Calculationsthread aufgerufen.

6.1.3. Datenmodell Anbindung

Für die Berechnungen im Simulator wird ein Fuzzy-System aus NRC-Objekten benötigt. Dieses wird aus dem Datenmodell der FuzzyIDE erstellt. Die benötigten NRC-Objekte bestehen bereits im Datenmodell der FuzzyIDE. Sie sind dort als private Member in den Klassen des Datenmodells(siehe Kap. 5) vorhanden. Um die NRC-Objekte des Datenmodells in eine anwendbare Form zu bringen, werden flache Kopien angelegt und in Vektoren geordnet. Um die Vektoren im Simulator mit dem Datenmodell in Übereinstimmung zu halten, wird ein Observer genutzt. Der Observer benachrichtigt den Simulator bei Änderungen am Datenmodell. Der Simulator aktualisiert daraufhin die Vektoren. Während der Simulation werden über die Vektoren die Methoden der NRC-Objekte für die Berechnung aufgerufen.

6.2. Oberfläche

6.2.1. Simulation Control Dialog

Mit Hilfe dieses Dialoges kann die Simulation gestartet und gestoppt werden. Er bietet die Möglichkeit, die Zeiträume zwischen den Berechnungsschritten und den Updateschritten festzulegen. Über die Checkbox "synchronized" des Dialoges kann festgelegt werden, dass Berechnungsschritt und Updateschritt in der Simulation synchron erfolgen. Auf diese Weise kann sichergestellt werden, dass bei jedem Berechnungsschritt neue Inputwerte für das Fuzzy-System bereitstehen und für jeden Simulationsschritt neue Outputwerte für das benutzte Plugin zur Verfügung stehen.

6.2.2. Rule Fire Dialog

Mit Hilfe des "Rule Fire" Dialogs kann der Aktivierungsgrad der Regeln während der Simulation ausgegeben werden. Der Aktivierungsgrad der Regeln wird als "double" Vektor von der "InferenceUnit" an den Dialog "Rule Fire" gegeben (siehe Abb. 6.1). Der Aktivierungsgrad der Regeln wird farblich dargestellt.

6.2.3. Settings Dialog

Über den "Settings" Dialog kann die Inferenzstrategie des Simulators festgelegt werden. Es können die Operatoren für Aggregation, Implikation, Akkumulation und Defuzzyfizierung festgelegt werden(siehe Abs. 2.2).

7. Plugins

Wie in Abschnitt 3.1 beschrieben, besitzt die FuzzyIDE Entwicklungsumgebung eine Microkernel-Architektur, die nur ein Minimum an Funktionalität bietet. Der Großteil der Funktionen wird in Plugins realisiert. Zu Beginn des Redesigns existierten bereits mehrere Plugins. Dieses Kapitel stellt die vorhandenen Plugins mit ihren Funktionen vor.

7.1. Essentielle Plugins

Die in diesem Abschnitt beschriebenen Plugins sind notwendig, um mit der FuzzyIDE arbeiten zu können.

Ruleeditor-Plugin

Das Ruleeditor-Plugin dient dem Anlegen und Bearbeiten der Regeln im Fuzzy-System. Das Plugin ist nicht an das aktuelle Datenmodell der FuzzyIDE angepasst und kann daher nicht verwendet werden. Das Plugin wird nicht weiter entwickelt, da es durch ein anderes Plugin(Ruleeditor2) mit größerem Funktionsumfang ersetzt wird.

Variableneditor-Plugin

Das Plugin dient dem Anlegen und Bearbeiten von Fuzzy-Variablen und den zugehörigen Fuzzysets. Das Plugin funktioniert, ist aber noch nicht vollständig fertig gestellt. Während des Redesigns wird weiter an diesem Plugin gearbeitet.

7.2. Tool Plugins

Die Plugins in den folgenden Abschnitten erleichtern die Arbeit mit der FuzzyIDE, sind aber nicht unbedingt notwendig.

Console

Das Console-Plugin leitet die Standart- und Fehlerausgabe in einen Dialog der FuzzyIDE um. Es bietet die Möglichkeit, die Ausgaben zu löschen und zu scrollen. Das Plugin ist voll funktionsfähig und es sind keine Probleme bekannt. Da das Plugin weder das Datenmodell noch den Simulator benutzt, sind während des Redesigns keine Änderungen am Plugin zu erwarten.

Fuzzytree-Plugin

Das Fuzzytree-Plugin stellt die Daten des aktuellen Fuzzy-Systems dar. Dafür werden Fuzzy-Variablen und Regeln des Fuzzy-Systems in einer Baumstruktur mit ihren Komponenten aufgelistet. Die Abbildung des Datenmodells wird über die Observer des Daten-

modells auf dem aktuellen Stand gehalten. Es ist nicht funktionsfähig, da es eine ältere Version des Datenmodells benötigt. Im Verlauf des Redesigns muss das Plugin an die Änderungen im Datenmodell angepasst werden.

7.3. Beispiel Plugins

Der Abschnitt beschreibt die vorhandenen Plugins, mit deren Hilfe ein bestimmter Sachverhalt oder eine Funktion der FuzzyIDE demonstriert wird.

Test-Plugin

Dieses Plugin dient zur Demonstration der Pluginschnittstelle. Das Plugin ist ebenfalls nicht funktionsfähig, da es auf einer alten Version des Datenmodells beruht.

Example-Plugin

Dieses Plugin dient ebenfalls zur Demonstration der Pluginschnittstelle. Es fügt einen Menüeintrag dem Menü der FuzzyIDE hinzu. Über den Menüpunkt wird ein leerer Dialog dargestellt. Von Programmierern eines neuen Plugins kann der Quelltext als Vorlage für das eigene Plugin verwendet werden. Es ist nicht funktionstüchtig, da es eine alte Version des Datenmodells benötigt.

SimpleSimulator

SimpleSimulator ist ein zu Demonstrationszwecken geschriebenes Plugin, welches die Arbeitsweise sowie die Schnittstellen des Simulators demonstriert. Es kann mit der Version, die zu Beginn des Redesigns vorliegt, nicht genutzt werden, da es auf dem veralteten XML-Datenmodell basiert.

7.4. Simulator Plugins

Die hier beschrieben Plugins dienen der Simulation eines Sachverhaltes.

Kugel-Plugin

Das Plugin dient der Simulation einer Eisenkugel, die im Magnetfeld eines Elektromagneten in der Schwebe gehalten wird. Das Fuzzy-System wird dabei benutzt, den Elektromagneten so zu steuern, dass er die Kugel in der Schwebe hält. Das Plugin ist nicht vollständig implementiert. Eine Anbindung an das FuzzyIDE Datenmodell fehlt. Es kann aus diesem Grund nicht benutzt werden.

Falter2-Plugin

Das Plugin simuliert den Flug eines Nachtfalters, der eine Lichtquelle umkreist. Das Plugin kann mit der vorliegenden Version der FuzzyIDE nicht genutzt werden, da es eine ältere Version des Datenmodells benötigt.

Falter-Plugin

Genau wie das Falter2-Plugin simuliert dieses Plugin einen Nachtfalter, der eine Lichtquelle umkreist. Das Plugin ist voll funktionsfähig und kann eingesetzt werden. Es nutzt die FuzzyIDE nur zum Darstellen der Ausgaben in einem Dialog. Das Plugin enthält ein eigenes Fuzzy-System und einen eigenen Simulator. Somit ist es vom Simulator und dem Datenmodell der FuzzyIDE unabhängig.

Propeller-Plugin

Das Propeller-Plugin dient dem Betreiben einer Propellerwippe, welche über die COM-Schnittstelle des Rechners angesteuert wird. Es nutzt für die Anbindung der Propellerwippe einen DDE-Server. Es ist mit der vorliegenden Version des Datenmodells nicht lauffähig.

7.5. Probleme

Wie in den vorherigen Abschnitten beschrieben, ist das häufigste Problem die Verwendung eines veralteten Datenmodells. Nahezu alle Plugins benutzen noch das aufgegebne, auf XML beruhende Datenmodell. Eine Anpassung der Plugins an die vorliegende Version des Datenmodells ist nicht sinnvoll, da sich das Datenmodell im Verlauf des Redesigns nochmals ändern wird.

Ein weiteres, in den Plugins auftretendes Problem ist die Vermischung von NRC-Klassen und Klassen des Datenmodells. Die Klassen des Datenmodells kapseln die Funktionalität ihrer NRC-Objekte nicht vollständig (siehe Abs. 5.2.3). Aus diesem Grund werden sowohl NRC-Objekte als auch Datenmodellobjekte in den Plugins benötigt. Da die NRC-Klassen und die entsprechenden Datenmodellklassen denselben Namen tragen, ist es in den Plugins nötig, den Packagenamen zusätzlich anzugeben, um die namensgleichen Klassen zu unterscheiden. Dies macht den Programmcode unübersichtlich und führt leicht zu Verwechslungen.

Teil III.

Anforderungsanalyse

8. Datenmodell

Aus der Ist-Analyse des Datenmodells in Kapitel 5 lässt sich eine Reihe von Grundsätzen für das Redesign des Datenmodells ableiten.

- Vermeidung doppelter Datenhaltung.
- Beheben der Referenzfehler.
- Schaffen eindeutiger Schnittstellen.
- Weitgehendes Beibehalten der Schnittstellen des Datenmodells.
- Vervollständigen der fehlenden Funktionen.

In diesem Kapitel wird in Grundzügen beschrieben, wie diese Änderungen im Datenmodell ereicht werden sollen. Die genaue Umsetzung der Änderungen werden im Kapitel 11 der Arbeit erläutert.

8.1. Änderungen zur Fehlerbehebung

8.1.1. NRC-Objekte

Als wichtigste und umfangreichste Änderung ist geplant, alle NRC-Member Objekte aus den Klassen des Datenmodells zu entfernen. Dies ist notwendig, da sich die NRC-Objekte nicht zur konsistenten Datenhaltung eignen (siehe Abs. 5.3.1 und 5.3.2). Als Ersatz für die NRC-Objekte ist eine Speicherung der Daten in primitiven Datentypen wie Integer, Double oder in einfachen Java Klassen (z.B. String, ArrayList) vorgesehen. Das Entfernen der NRC-Objekte bietet zudem den Vorteil, dass die Namensgleichheit einiger Klassen (z.B. FuzzyVariable usw.) im Package de.htwdd.robotic.fuzzyide.fuzzymodel mit den NRC-Klassen im Package nrc.fuzzy nicht mehr zu Verwechslungen führt.

Da die NRC-Objekte auch zum Grossteil für die Fälle der doppelten Datenhaltung verantwortlich sind, entfällt auch diese Fehlerquelle mit dem Entfernen der NRC-Objekte.

8.1.2. Konsistentes Datenmodell

Um das Datenmodell konsistent zu halten, ist es notwendig, einen Mechanismus zu implementieren, der das Löschen oder unzulässige Verändern von Datenmodellobjekten verhindert. Es muss beispielsweise verhindert werden, dass Fuzzysets(Terme) und Variablen gelöscht werden, die noch in Regeln des Fuzzy-Systems benutzt werden. Auch das Ändern des Fuzzy-Variablentyps (Input, Output) muss verhindert werden, solange die Fuzzy-Variable noch in Regeln verwendet wird. Es ist also eine Art Schreibschutz zu implementieren, der das Ändern von Objekten nur zulässt, wenn dies keine negativen Auswirkungen auf die Konsistenz des Datenmodells hat.

Die im vorherigen Abschnitt beschriebene Art der Datenhaltung in primitiven Datentypen bringt einige Probleme mit, die wieder korrigiert werden müssen. Ein Beispiel dafür sind die linguistischen Expressions, die als String der FuzzyValue Klasse übergeben werden. Die Expressions werden im bestehenden Datenmodell durch das NRC-Objekt(Klasse nrc.fuzzy.FuzzyValue) geparst und auf ihre Gültigkeit geprüft. Ohne die NRC-Objekte muss ein geeigneter Parser geschrieben werden, um die Expressions auf ihre Gültigkeit zu prüfen.

Eine weitere Funktion, die mit dem Entfernen der NRC-Objekte ersetzt werden muss, ist das Generieren von Punkten. Im bestehenden Datenmodell generiert das NRC-Objekt bei einigen Fuzzysettypen die Punkte des Fuzzyssets.

Beispielsweise bei den Pi-Fuzzysets (siehe Abb. D. Bei diesem Set werden nur zwei X-Werte dem Konstruktor übergeben und alle Punkte des Sets vom NRC-Objekt auf Basis einer mathematischen Funktion berechnet. Mit dem Entfernen der NRC-Objekte ist es nicht mehr möglich, die Punkte des Fuzzysets zu generieren. Es entfällt damit auch die Möglichkeit, das Fuzzyset graphisch darzustellen. Es muss daher ein Weg gefunden werden, die Punkte des Sets auf andere Weise zu berechnen.

8.2. Erweiterungen des Datenmodells

8.2.1. FuzzySet Klasse

Die Klasse *de.htwdd.robotic.fuzzyide.fuzzymodel.FuzzySet*, welche die Daten zu einen Fuzzyset hält, soll während des Redesigns durch eine abstrakte Klasse "FuzzySet" ersetzt werden. Von dieser abstrakten Klasse wird dann für jeden Fuzzysettyp (LFuzzySet usw. siehe Abb. D) eine eigene Kasse abgeleitet. Diese Änderung ist wichtig, da bei vielen Fuzzysettypen Methoden mit ungültigen Parametern aufgerufen werden können. So erwartet beispielsweise der Fuzzysettyp Singleton genau einen Punkt, der den Y-Wert(Zugehörigkeitswert) von 1 besitzt. Bei Aufruf der Methode "setPoints(Vector points)" muss also geprüft werden, dass der Vektor "points" nur einen Punkt mit dem Y-Wert 1 besitzt. Ähnliche Prüfungen sind für die 10 anderen Fuzzysettypen notwendig. Um diese Prüfungen zu umgehen und eine eindeutigere Schnittstelle zu schaffen, wird die Methode "setPoints(Vector points)" entfernt und in den abgeleiteten Klassen durch eine Methode "setParameter" ersetzt, mit der die Parameter des entsprechenden Fuzzysettyps gesetzt werden können.

8.2.2. Temporäre Variablen

Wie in der Anforderungsanalyse des Simulators(siehe Kap. 9) zu erkennen ist, werden temporäre Variablen zum Speichern von Zwischenergebnissen während der Simulation benötigt. Um diese von den Fuzzy-Variablen unterscheiden zu können, müssen sie getrennt von den Fuzzy-Variablen im Datenmodell gehalten werden. Andererseits gibt es einige Plugins, für die nicht von Bedeutung ist, ob eine verwendete Variable eine temporäre Variable oder eine Fuzzy-Variable ist. Aus diesem Grund ist ein gemeinsames Interface zu definieren. Über dieses Interface kann dann auf beide Variablentypen zugegriffen werden.

8.3. Fehlerbehandlung

Die Fehlerbehandlung, die zurzeit teilweise von den NRC-Klassen geregelt ist, wird durch eigene Exceptionklassen ersetzt. Dies ist notwendig, da die NRC Elemente aus dem Datenmodell entfernt werden. Als weitere Änderung werden alle Exceptionklassen des Datenmodells von der Klasse "FuzzyException" abgeleitet, um die Exceptions in einer sinnvollen Hierarchie zu ordnen.

9. Simulator

Wie in der Ist-Analyse des bestehenden Simulators beschrieben, beruht dieser auf dem alten Datenmodell der FuzzyIDE. Aufgrund des im letzten Kapitel beschriebenen Redesign des Datenmodells, entfallen die NRC-Objekte aus dem Datenmodell. Diese Objekte benutzt jedoch der bestehende Simulator. Wegen dieser Änderungen im Datenmodell und den geplanten Änderungen am Simulator, ist eine Erweiterung des bestehenden Simulators nicht sinnvoll. Im Verlauf des Redesigns wird der existierende Simulator entfernt und durch einen neuen ersetzt. Das Design des neuen Simulators orientiert sich dabei am Aufbau des Simulators der CubiCalc Fuzzy-Entwicklungsumgebung [Cub93].

9.1. Anforderungen

Im FuzzyIDE Paket gibt es eine Reihe von Plugins, die jeweils nur einen Sachverhalt simulieren. Zum Beispiel die Plugins "Falter-Plugin", "Kugel" "Propeller" und "Falter2-Plugin" (siehe Kap. 7). Für jede Simulation ein eigenes Plugin zu programmieren, ist sehr zeitaufwändig und umständlich. Jedes Plugins nutzt eigene Klassen zur Eingabe, Visualisierung und Steuerung der Simulation. Der bestehende Simulator ist auch keine entscheidende Vereinfachung, da er für jede Simulation nur ein Plugin benutzen kann. Dieses Plugin muss die Inputwerte beschaffen und die Outputwerte des Simulators verarbeiten. Also ist es auch mit dem bestehenden Simulator notwendig, für jedes simulierte System ein eigenes Plugin zu schreiben.

Mit dem geplanten Simulator sollen diese einseitigen und eingeschränkten Lösungen vermieden werden. Ziel ist es, einen Simulator zu implementieren, der mehrere Plugins verwenden kann. Die Aufgaben des Simulators beschränken sich dabei auf das Steuern der Simulation und das Aufrufen der Plugins. Mit einem Simulator, der mehrerer Plugins in einer Simulation nutzen kann, ist es möglich, wieder verwendbare Plugins für spezielle Aufgaben zu implementieren. Auf diese Weise muss zum Beispiel nur ein Plugin zur visuellen Ausgabe von Ergebnissen implementiert werden. Dieses Plugin kann dann in allen simulierten Systemen wieder verwendet werden.
Anforderungen, die der neue Simulator erfüllen muss, sind:

- **Anpassbarkeit:** Er muss die Möglichkeit besitzen, die Plugins auszuwählen, die verwendet werden sollen. Darüber hinaus soll auch die Reihenfolge frei wählbar sein, in der die Plugins gerufen werden.

- **Laufzeitverhalten:** Der Simulator soll Schnittstellen bereitstellen, über welche die verwendeten Plugins nicht nur auf die In- und Outputwerte des Fuzzy-Systems Zugriff haben, sondern auch auf die Einstellungen des Fuzzy-Systems. (z.B.: Aktivierungsgrad der Regeln, Sicherheitsfaktoren usw.)

- **Fehlerbehandlung:**Um nicht in jedem vom Simulator verwendeten Plugin, eine Fehlerbehandlung aufbauen zu müssen, soll der Simulator die Exceptions der Plugins

behandeln. Der Nutzer soll dabei entscheiden können, welche Fehler zum Abbruch der Simulation führen und welche Fehler ignoriert werden.

- **Variablen-Zugriff:** Das Setzen der Inputs und das Auswerten der Outputs sollte über die entsprechenden Variablen des Datenmodells möglich sein.
- **Temporäre Variablen:** Nutzen von temporären Variablen, um Werte zwischen den Simulatorplugins auszutauschen.

9.2. Aufbau

9.2.1. Anbindung der NRC-Bibliothek

Der bestehende Simulator benutzt die NRC-Objekte des Datenmodells für die Berechnungen des Fuzzy-Systems. Da geplant ist, ein Datenmodell ohne NRC-Objekte zu schaffen, diese Objekte jedoch für die Fuzzy- Berechnungen im Simulator benötigt werden, muss vor Beginn der Simulation ein NRC-Modell erstellt werden. Die Informationen des Datenmodells werden genutzt, um die Konstruktoren der NRC-Klassen aufzurufen und ein "lauffähiges" NRC-Modell für die Berechnungen zu erstellen. Wichtig ist hierbei, dass die Objekte des Datenmodells und die des neu erstellten NRC-Modells in Verbindung gesetzt werden. So ist ein Zugriff auf die NRC-Objekte über die Datenmodellobjekte möglich.

9.2.2. Inferenz Einstellung

Die NRC-Bibliothek beherrscht eine Reihe von Inferenz-Strategien zum Berechnen der Outputwerte des Fuzzy-Systems. So ist es möglich, die Operatoren für die Aggregation, Implikation, Akkumulation sowie die Defuzzyfizierung festzulegen. Um die Fähigkeiten der NRC-Bibliothek auszuschöpfen, sollte auch der Simulator alle möglichen Inferenz-Strategien der NRC-Bibliothek unterstützen. Dabei muss die Strategie vor Beginn der Simulation festgelegt werden, da auf der Basis der Inferenz-Strategie ein entsprechendes NRC-Modell beim Start der Simulation angelegt werden muss.

9.3. Pluginanbindung

Der Simulator soll Plugins benutzen. Welche Plugins der Simulator nutzt, soll dabei vom Nutzer festgelegt werden können. Damit der Simulator Plugins nutzen kann, muss ein Interface definiert werden, über das er mit den Plugins kommuniziert. Geplant ist, ein Interface *ISimulatorPlugin* zu definieren, das alle Simulatorplugins implementieren müssen. Dies ist auch notwendig, um die Simulatorplugins von den anderen Plugins zu unterscheiden.

Die einfachste Möglichkeit, Plugins an den Simulator anzubinden wäre, im ISimuator-Plugin Interface eine Methode zu definieren, die von allen Simulatorplugins implementiert wird. Diese Methode der Plugins wird während der Simulation durch den Simulator aufgerufen und die Ergebnisse der Fuzzy-Berechnungen werden als Parameter übergeben. Das Problem hierbei ist, dass die Methode der Plugins nach den Fuzzy-Berechnungen gerufen wird. Das führt dazu, dass im ersten Simulationsschritt die Fuzzy-Berechnungen durchgeführt werden, ohne dass Inputwerte durch die Plugins gesetzt werden können. Einen

Ausweg bringt es auch nicht, die Plugins vor den Fuzzy-Berechnungen aufzurufen, da in diesem Fall die Plugins im ersten Simulationsschritt keine gültigen Outputwerte vom Fuzzy-System erhalten. Es ist also sinnvoll Sinnvoll wäre es, die Plugins, welche den Input des Fuzzy-Systems bereitstellen, vor den Fuzzy-Berechnungen und die Ausgabeplugins danach aufzurufen.

In der Fuzzy-Entwicklungsumgebung CubiCalc benutzt der Simulator keine Plugins. Es gibt jedoch vier Textboxen, in die Programmcode eingegeben werden kann. Dieser wird während der Simulation zu unterschiedlichen Zeitpunkten ausgeführt [Cub93].
Diese Zeitpunkt sind:

Initialisierung:
Der hier eingegebene Programmcode wird einmal zu Beginn der Simulation ausgeführt.

Preprocessing:
Dieser Code wird bei jedem Simulationsschritt ausgeführt, bevor die Berechnungen des Fuzzy-Systems durchgeführt werden.

Simulation:
Dieser Code wird bei jedem Simulationsschritt ausgeführt, nachdem die Fuzzy-Berechnungen durchgeführt wurden, jedoch vor der Aktualisierung der Anzeigen.

Postprocessing
Dieser Code wird bei jedem Simulationsschritt ausgeführt nachdem die Fuzzy-Berechnungen durchgeführt und die Anzeigen aktualisiert wurden.

Ähnlich sollen die Schnittstellen zwischen den Simulatorplugins und dem neuen Simulator der FuzzyIDE aufgebaut sein. Statt zu den vier oben beschriebenen Zeitpunkten Programmcode auszuführen, sollen an diesen Zeitpunkten die Simulatorplugins aufgerufen werden. Da der neue Simulator Plugins für visuelle Ausgaben verwenden soll, macht die Postprocessing Schnittstelle keinen Sinn. Dafür würde es Sinn machen, noch eine Schnittstelle einzuführen, die beim Beenden der Simulation aufgerufen wird. Deshalb wird die Postprocessing Schnittstelle wegfallen und durch eine Destructor Schnittstelle ersetzt, über die beim Beenden der Simulation Plugins aufgerufen werden.

9.3.1. Pluginhaltung

Um dem Nutzer die Möglichkeit zu geben, das Verhalten des Simulators anzupassen, muss es möglich sein, die Simulatorplugins auszuwählen, die während der Simulation verwendet werden sollen. Die Einstellung, ob ein Plugin verwendet wird oder nicht, soll an einer zentralen Stelle möglich sein. Dafür müssen in einer Klasse des Simulators Referenzen auf alle geladenen Simulatorplugins gehalten werden. Zu jeder Referenz eines Plugins muss zudem noch die Information gespeichert werden, ob es während der Simulation verwendet werden soll oder nicht. Um Referenzen der Simulatorplugins an einer zentralen Stelle zu sichern, muss der Pluginmanager der FuzzyIDE angepasst werden. Er muss nun beim Laden eines Plugins prüfen, ob es sich um ein Simulatorplugin handelt und gegebenenfalls eine Referenz dem Simulator übergeben.

9.3.2. Änderungen zur Laufzeit

Da während der Simulation das Verhalten des Fuzzy-Systems getestet wird, ist es sinnvoll, einige wichtige Änderungen am Fuzzy-System während der Simulation vornehmen zu können. Um den Simulator möglichst schlank zu halten, sollten auch für diese Änderungen Plugins genutzt werden. Es muss die Schnittstelle zwischen Plugins daher so definiert werden, dass ein Plugin die Möglichkeit bekommt, Änderungen an der Inferenz-Einheit vorzunehmen.

9.4. Fehlerbehandlung

Während der Simulation kann es zu Fehlern bei der Berechnung des Fuzzy-Systems oder beim Aufrufen der Plugins kommen. Um nicht in jedem Plugin eine Fehlerbehandlung implementieren zu müssen, soll der neue Simulator die Fehler der Plugins und des Fuzzy-Systems an einer Stelle behandeln. Die Plugins und das Fuzzy-System werfen Exceptions, die dann an zentraler Stelle ausgewertet werden. Mit dieser Methode kann der Nutzer vor Beginn der Simulation festlegen, ob ein bestimmter Fehler zum Abbruch der Simulation führen soll oder nicht. Es sollte beispielsweise möglich sein festzulegen, ob die Simulation abgebrochen wird, wenn der Wert für eine Fuzzy-Variable außerhalb der Basisskala liegt.

10. Plugins

10.1. Benötigte Plugins

Wie im Abschnitt 3.1 beschrieben, besitzt die FuzzyIDE einen Microkernel, der nur ein Minimum an Funktionen realisiert. Der Grossteil der Funktionalität der FuzzyIDE wird durch Plugins bereitgestellt. Um mit der FuzzyIDE sinnvoll arbeiten zu können, sind Plugins notwendig, die folgende Funktionen umsetzen:

- Variableneditor zum Anlegen und Bearbeiten von Fuzzy-Variablen und den zugehörigen Fuzzysets.
- Ruleeditor, zum Anlegen und Bearbeiten der Regeln des Fuzzy-Systems.
- Es wird ein Inputplugin benötigt, welches dem Simulator der FuzzyIDE Eingabewerte für das Fuzzy-System zu Verfügung stellt.
- Es wird ein Outputplugin benötigt, welches die Ergebnisse des Fuzzy-Systems darstellt.
- Codegeneration Plugin, welches dazu benötigt wird, das Fuzzy-System als Quellcode auszugeben.

Plugins, welche die ersten beiden Aufgaben erfüllen, sind bereits für die FuzzyIDE vorhanden. Diese Plugins werden während des Redesigns von ihren Entwicklern angepasst, so dass sie auch weiterhin genutzt werden können. Zu den restlichen drei genanten Punkten sind noch keine Plugins vorhanden. Da diese Funktionen aber nötig sind, um die FuzzyIDE einzusetzen, werden entsprechende Plugins im Verlauf des Redesigns erstellt. In diesem Abschnitt sollen die Anforderungen an die benötigten Plugins beschrieben werden.

10.1.1. Simulator Inputplugin

Die Anforderung an ein Simulator Inputplugin ist, die Inputvariablen des Fuzzy-Systems während der Simulation zu setzen. Dies könnte zum Beispiel durch das Lesen von Werten aus einer Datei geschehen. Das Plugin müsste die Werte lesen und den Inputvariablen des Fuzzy-Systems zuordnen. Ein solches Plugin ist durchaus sinnvoll, jedoch nicht universell einsetzbar, da es immer auf Inputwerte aus einer Datei angewiesen ist. Da noch keine Plugins für den Simulatorinput existieren, macht es Sinn, zuerst ein Plugin zu schreiben, welches einen größeren Funktionsumfang bietet und nicht auf Vorgaben, wie das Vorhandensein von Inputwerten in Dateien, angewiesen ist. Um dies zu ereichen, ist geplant, ein Plugin zu implementieren, welches Quellcode interpretiert und während der Simulation ausführt. Das Plugin soll dabei ähnliche Funktionalität wie die Skriptsprache der Fuzzy-Entwicklungsumgebung CubiCalc aufweisen [Cub93] und sich leicht um weitere Funktionen erweitern lassen.

10.1.2. Simulator Outputplugin

Zur Ausgabe von Simulatorwerten ist das auch das oben beschriebene Interpreter-Plugin geeignet. Um die Ergebnisse eines Fuzzy-Systems anzuzeigen und das Funktionieren zu prüfen, ist ein Plugin, welches die Ausgaben visuell darstellt, besser geeignet. Es ist geplant, ein Plugin zu entwickeln, welches die Werte zweier beliebiger Variablen des Datenmodells als XY-Diagramm darstellt. Es soll dabei auch möglich sein, mehrere Diagramme während einer Simulation anzuzeigen.

10.1.3. Codegeneration Plugin

Ein Plugin zum Generieren von Quellcode ist zwar nicht zwingend notwendig, um mit der FuzzyIDE zu arbeiten. Es wird jedoch dazu benötigt, das erstellte Fuzzy-System in eigene Anwendungen einzubinden. Da die FuzzyIDE für die Fuzzy-Berechnungen auf die NRC-Bibliothek angewiesen ist, wird auch der erzeugte Quellcode auf diese Bibliothek angewiesen sein. Für die Nutzung eines Fuzzy-Systems auf einem Mikrocontroller sind die NRC-Bibliothek und die Programmiersprache Java ungeeignet. Aus diesem Grund wird versucht, auch eine Ausgabe als C Quellcode zu implementieren, die nicht auf die NRC-Bibliothek angewiesen ist.

10.2. Vorhandene Plugins

Das Redesign betrifft viele zentrale Bereiche der FuzzyIDE. So kommt es zu Rückwirkungen auf die bereits vorhandenen Plugins. Besonders betrifft dies die Plugins, die das Datenmodell der FuzzyIDE nutzen. An ihnen müssen Anpassungen vorgenommen werden, um ein Weiterfunktionieren gewährleisten zu können. Während des Redesigns werden von den Mitgliedern der FuzzyIDE Gruppe einige Plugins durch neue Versionen mit mehr Funktionalität ersetzt. Das macht das Anpassen der alten Versionen an das neue Datenmodell überflüssig. Die alten Versionen werden aufgegeben und nicht weiter entwickelt. Hier soll erläutert werden, wie mit den vorhandenen Plugins der FuzzyIDE verfahren wird.

10.2.1. Anzupassende Plugins

Während des Redesigns wird versucht, folgende Plugins an die neue Version der FuzzyIDE anzupassen. Diese Plugins wurden ausgewählt, weil entweder eine Anpassung einfach möglich ist oder sie für die Arbeit mit der Entwicklungsumgebung benötigt werden.

- Fuzzytree Plugin
- Variableneditor
- Console Plugin
- ExamplePlugin
- Falter Plugin

Das Variableneditor-Plugin wird noch entwickelt (siehe Anh. C). Das Plugin wird von seinem Entwickler an die Änderungen während des Redesigns angepasst. Die restlichen oben aufgeführten Plugins werden im Rahmen dieser Diplomarbeit angepasst.

10.2.2. Aufgegebene Plugins

Die folgenden Plugins werden nicht im Zuge des Redesigns an die neue Version der FuzzyIDE angepasst, da sie nicht mehr benötigt werden oder eine Anpassung zu aufwändig ist.

- **Falter2-Plugin:** Das Plugin verwendet den bestehenden Simulator der FuzzyIDE, der im Rahmen des Redesigns durch ein neuen Simulator ersetzt wird.
- **Simplesimulator:** Das Plugin dient der Demonstration des Simulators der FuzzyIDE. Da der Simulator ersetzt wird, ist auch dieses Plugin nicht mehr notwendig.
- **Kugel-Plugin:** Das Plugin besitzt noch kein Datenmodell und muss noch fertig gestellt werden.
- **Ruleeditor-Plugin:** Das Plugin wird durch das neue Ruleeditor2-Plugin ersetzt(siehe Anh. C).

10.3. Benötigte Schnittstellen

Auf Grund der geplanten Erweiterungen der FuzzyIDE wird es notwendig werden, neue Interfaces zu definieren, um Informationen zwischen dem Systemkern und den Plugins auszutauschen. Wie in den Abschnitten 10.1.1 und 10.1.2 zu erkennen ist, wird es notwendwendig, dass einige Plugins Daten halten: zum Beispiel den Code einer Skriptsprache oder die Information, welche Werte in einem Plot ausgegeben werden sollen. Bei der Speicherung des Projektes gehen derzeit diese Daten verloren. Um nicht jedes Mal den Interpreter-Code oder die Plots neu von Hand anlegen zu müssen, sollten diese Plugins über Methoden zum Laden und Speichern ihrer Daten verfügen. Aus diesem Grund ist es sinnvoll, ein neues Plugin Interface zu definieren, über das die Plugins ihre zu sichernden Informationen an die FuzzyIDE weitergeben können. Auf diesem Wege können die Informationen der Plugins in das Savefile der FuzzyIDE einfließen. Beim Laden eines FuzzyIDE Savefiles werden die Daten dem entsprechenden Plugin wieder übergeben. Dabei muss in Betracht gezogen werden, dass beim Laden des Savefiles möglicherweise nicht alle Plugins vorhanden sind, welche Daten gesichert haben.

Teil IV.

Implementierung

11. Datenmodell

Während des Redesigns wurden die in der Anforderungsanalyse(siehe Kap. 8) beschriebenen Anforderungen umgesetzt. In diesem Kapitel wird die Umsetzung der wichtigsten Punkte beschrieben.

11.1. Eindeutigkeit

11.1.1. Benutzt-Konzept

Um ein eindeutiges Datenmodell zu schaffen, war es nötig, einen Mechanismus zu implementieren, welcher das unzulässige Löschen oder Verändern von Datenmodellelementen verhindert. Zu diesem Zweck wurde ein Verfahren entwickelt, mit welchem Datenmodellelemente als "benutzt" gekennzeichnet werden können. Ein Objekt wird als "benutzt" gekennzeichnet, indem es eine Referenz auf das Objekt speichert, von dem es benutzt wird. Wird das Objekt nicht mehr von einen bestimmten Objekt benutzt, wird die Referenz auf das benutzende Objekt wieder gelöscht. Um dies zu realisieren, wurde die abstrakte Basisklasse *de.htw.robotic.fuzzyide.fuzzymodel.FuzzyElement* erweitert. Von dieser Klasse sind alle Datenmodellklassen abgeleitet, die Daten halten. Die FuzzyElement Klasse enthält nun 4 zusätzliche Methoden.

Methode: public void use(FuzzyElement fe)

Mit Aufruf dieser Methode wird dem Objekt bekannt gegeben, dass es vom FuzzyElement Objekt "fe" benutzt wird. Dazu wird eine Referenz des Objektes "fe" in einer Liste gespeichert. Existiert bereits eine Referenz zu dem Objekt "fe", wird keine neue Referenz gespeichert.

Methode: public void unuse(FuzzyElement fe)

Mit dieser Methode kann einem FuzzyElement Objekt angezeigt werden, dass es nicht mehr durch das im Parameter "fe" übergebene FuzzyElement Objekt benutzt wird. Dazu wird die Referenz von "fe" aus der Liste gelöscht.

Methode: public boolean isUsed()

Gibt *"true"* zurück, falls das FuzzyElement Objekt durch ein anderes Element benutzt wird. Sollte das Element nirgends benutzt werden, ist der Rückgabewert *"false"*.

Methode: public String isUsedFrome()

Diese Methode gibt an, von welchen Elementen das FuzzyElement benutzt wird. Dazu wird bei jeden Element in der Liste der Elemente, die das Objekt benutzen, die Methode

"toString()" aufgerufen. Die einzelnen Strings werden dann zu einem String zusammengefügt, den die Methode zurückgibt. Sollte das Element nirgends verwendet worden sein, wird ein leerer String zurückgegeben.

Diese Methoden allein verhindern natürlich nicht das Löschen oder Verändern von benutzten Datenmodellelementen. Um dies zu ereichen, müssen alle Methoden, die ein Element unzulässigerweise verändern können, prüfen, ob dieses Element benutzt wird. Zudem ist es notwendig, wenn ein Element ein anderes benutzt die *use* Methode des benutzten Elementes aufzurufen. Für einige Klassen war es notwendig, die *use* und *unuse* Methoden zu überschreiben. So zu Beispiele bei der Klasse *FuzzyValue* um den Fuzzysets direkt bekant zugeben von welcher Regel sie benutzt werden und nicht erst den Umweg über die FuzzyValue Klasse zugehen. Für den Fall, dass versucht wird, ein benutztes Element zu löschen oder unzulässig zu verändern, gibt es eine neue Exception. Diese gibt dem System bekannt, dass die Änderung nicht möglich ist, da ein Element benutzt wird.

11.1.2. Linguistikexpression Parser

Die Klasse *de.htwdd.robotic.fuzzyide.fuzzymodel.FuzzyValue* enthält eine linguistische Expression, die in den Regeln des Fuzzy-Systems benötigt wird. In der ursprünglichen Version des Datenmodells wurde die Expression an ein NRC-Objekt vom Typ
nrc.fuzzy.FuzzyValue übergeben. Diese prüfte die Expression auf ihre Gültigkeit und erstellte aus ihr eine Fuzzy-Menge. Da während des Redesigns alle NRC-Objekte aus dem Datenmodell entfernt wurden, war es notwendig, die Ausdrücke auf anderem Wege auf ihre Gültigkeit zu prüfen.

Um den Expression String zu prüfen, wurde mit Hilfe des Jay [JAY] Parsergenerators ein Parser generiert. Er zerlegt den Expression String in seine Bestandteile. Der vom Parser benötigte Lexer wurde dabei mit dem Tool JLex[JLE] generiert. Während der Zerlegung des Strings durch den Parser wird überprüft, ob die Modifikatoren dem System bekannt sind und der gesamte String der Grammatik der NRC-Expressions entspricht. Zusätzlich werden die im String vorhandenen Namen der Fuzzysets mit den Namen der Fuzzysets aus der verwendeten Variablen verglichen. Wird das FuzzyValue Objekt dem "WENN"- oder "DANN"-Teil eines FuzzyRule Objekts hinzugefügt, werden die Fuzzysets der Expression als "benutzt" durch das FuzzyRule Objekt gekennzeichnet. Dafür ruft das FuzzyRule Objekt die überschriebene *use* Methode der FuzzyValue Klasse auf, welche wiederum die *use* Methoden der benutzten Fuzzysets aufruft. Beim Entfernen eines FuzzyValue Objektes aus einem FuzzyRule Objekt wird auf dieselbe Weise die Methode *unuse* aufgerufen. Dieses Vorgehen verhindert das Löschen von Variablen oder Fuzzysets, die noch in Regeln des Fuzzy-Systems verwendet werden.

11.2. Neuerungen

11.2.1. Basisklasse FuzzyElement

Um den Nutzer der FuzzyIDE die Möglichkeit zu geben, Notizen zu Datenmodellelementen zu speichern, wurde die Basisklasse FuzzyElement wie folgt erweitert:

- *public void setDescription(String des)*
  Diese Methode weist dem FuzzyElement Objekt eine Beschreibung zu.
- *public String getDescription()*
  Die Methode gibt die Beschreibung des FuzzyElementes zurück.

Um die Methoden anwenden zu können, wurde das Plugin Fuzzytree so erweitert, dass zu den Datenmodellelementen (Fuzzysets usw.) eine Beschreibung eingegeben werden kann. Da die Beschreibungen Teil des Datenmodells sind, fließen sie auch mit ins Savefile des Projektes ein und werden beim Laden eines Savefiles wieder hergestellt.

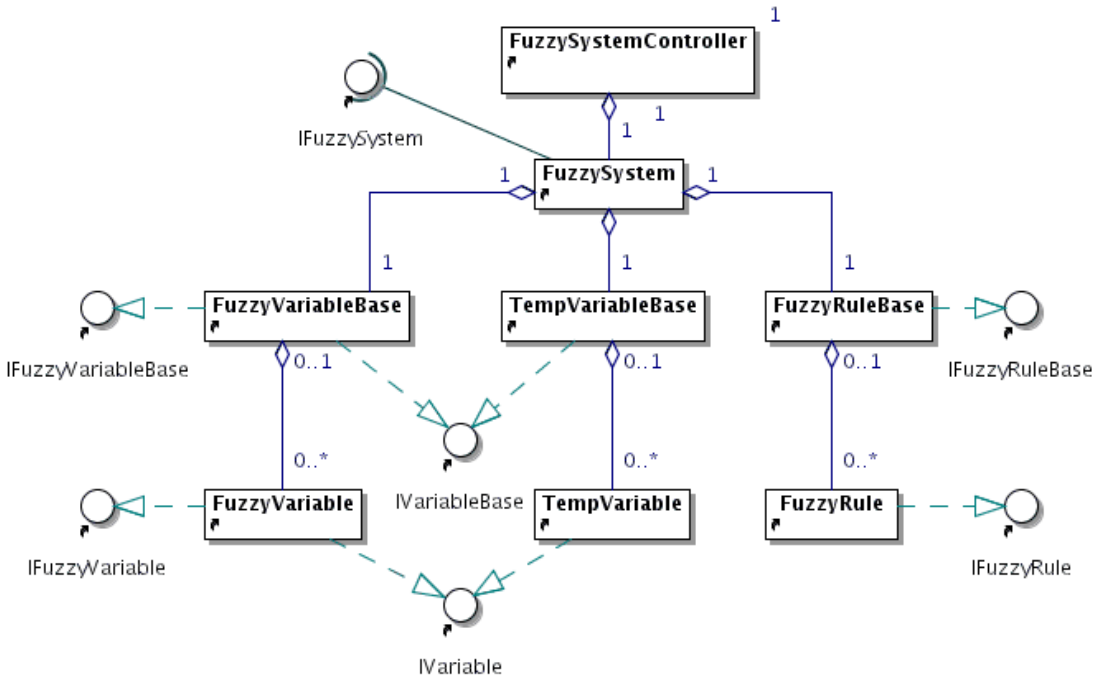


Abbildung 11.1.: Änderungen am Datenmodell

11.2.2. Temporäre Variablen

Mit den Änderungen am Simulator war es notwendig, temporäre Variable in das Datenmodell einzuführen. Diese werden benutzt, Werte zwischen den einzelnen Plugins des Simulators auszutauschen.

Für die temporären Variablen wurden zwei neue Klassen im Datenmodell angelegt (siehe Abb. 11.1). Die Klasse TempVariable, welche Informationen zu einer temporären Variablen hält und die Klasse TempVariableBase, welche alle TempVariable Objekte des Datenmodells speichert. Die temporären Variablen besitzen im Gegensatz zu den Fuzzy-Variablen keine Basisskala und keine Ausprägungen. Sie besitzen nur einen Namen und einen Wert.

Da es in einigen Fällen unbedeutend ist, ob es sich bei einer Variablen um eine Fuzzy- oder TempVariable handelt, wird das Datenmodell zusätzlich um zwei Interfaces erweitert: das IVariable Interface, welches von den Klassen TempVariable und FuzzyVariable

implementiert wird und das IVariableBase Interface, welches von den Klassen FuzzyVariableBase und TempVariableBase implementiert wird. Über diese Interfaces kann auf Variablen zugegriffen werden, unabhängig davon, ob es sich dabei um Fuzzy-Variablen oder temporäre Variablen handelt.

11.2.3. Methoden

Im Codegeneration-Plugin wurden einige Werte für Fuzzysets benötigt, die bisher nicht notwendig waren. Da diese Werte aber auch für andere zukünftige Plugins von Nutzen sein können, wurden die Funktionen zur Berechnung nicht im Plugin sondern im Datenmodell implementiert.

Methode: double getArea()

Die Methode gibt den Flächeninhalt des Fuzzysets zurück. Zu beachten ist bei dieser Funktion, dass sich bei den Fuzzysettypen L-, R-, Z- und S-Fuzzyset der Flächeninhalt ändert, sobald das Set einer Variablen zugeordnet wird. Der Grund hierfür ist, dass die Settypen als Punkt entweder den oberen oder unteren Rand der Basisskala (UoD) enthalten. Diese Punkte sind aber nicht im Fuzzyset, sondern nur in der Variablen bekannt. Somit kann für diese Typen der Flächeninhalt nur korrekt berechnet werden, wenn sie einer Variablen zugewiesen wurden.

Methode: double getCOA()

Diese Funktion gibt das Flächenträgheitsmoment(COA) zurück. Es gelten bei dieser Methode für die L-, R-, S- und Z-Fuzzysets dieselben Einschränkungen wie bei der *getArea()* Methode. Da das Flächenträgheitsmoment ein wichtiges Merkmal einer Fuzzy-Menge ist, wurde die Ausgabe der Methode auch in das Plugin Fuzzytree übernommen.

Methode: double getMight()

Die Funktion gibt das gewichtete Mittel der Y-Werte über die gesamte Basisskala der Variablen zurück. Diese ist Methode nur anwendbar, wenn das Fuzzyset einer Fuzzy-Variablen zugewiesen wurde, da sonst die Basisskala unbekannt ist. Sollte die Methode aufgerufen werden, bevor das Fuzzyset einer Variablen zugewiesen wurde, ist der Rückgabewert der Funktion -1.

Die FuzzyVariable Klasse wurde außerdem um die Methoden *setValue(doubel value)* und *getValue()* erweitert. Sie werden benötigt, um der Variablen einen Wert zuzuweisen bzw. diesen Wert zu lesen. Sie werden während der Simulation benutzt, um Inputwerte an das Fuzzy-System zu geben und die Outputwerte des Systems zu erhalten. Sollte versucht werden mit der Methode *setValue(doubel value)* Werte zu setzen, die außerhalb der Basisskala der Fuzzy-Variablen liegen, wird eine Exception durch die Methode geworfen.

11.2.4. Mathematical Expression Parser

Mit dem Entfernen der NRC-Objekte aus den Datenmodell kam es zu einem negativen Effekt: die S-, Z-, und Pi-Fuzzysets ließen sich nicht mehr grafisch im Variableneditor Plugin

darstellen. Der Grund hierfür war, dass ein Teil der Punkte durch NRC-Objekte generiert wurde. Somit ließen sich diese Punkte nach dem Entfernen der NRC-Objekte nicht mehr darstellen. Die Punkte werden in der NRC-Bibliothek durch die Klassen RFunction, LFunction sowie für die S- und Z-Fuzzysets durch die abgeleiteten Klassen SFunction und ZFunction erzeugt. Um im der FuzzyIDE trotzdem S-, Z- und Pi-Fuzzysets grafisch darstellen zu können und die Funktionalität der NRC-Bibliothek zu erweitern, wurde das Datenmodell der FuzzyIDE um die Klassen *de.htwdd.robotik.fuzzyide.fuzzymodel.RFuction* und *LFunction* ergänzt. Diese beiden Klassen werden in den Pi-, S-, Z-, L-, R- und LR Fuzzysetklassen dazu benutzt, Punkte zu generieren. Hierzu wird einem Objekt der *RFunction* bzw. *LFunction* Klasse eine mathematische Funktion übergeben. Mit Hilfe der Funktion werden die Werte der Fuzzysetpunkte berechnet.

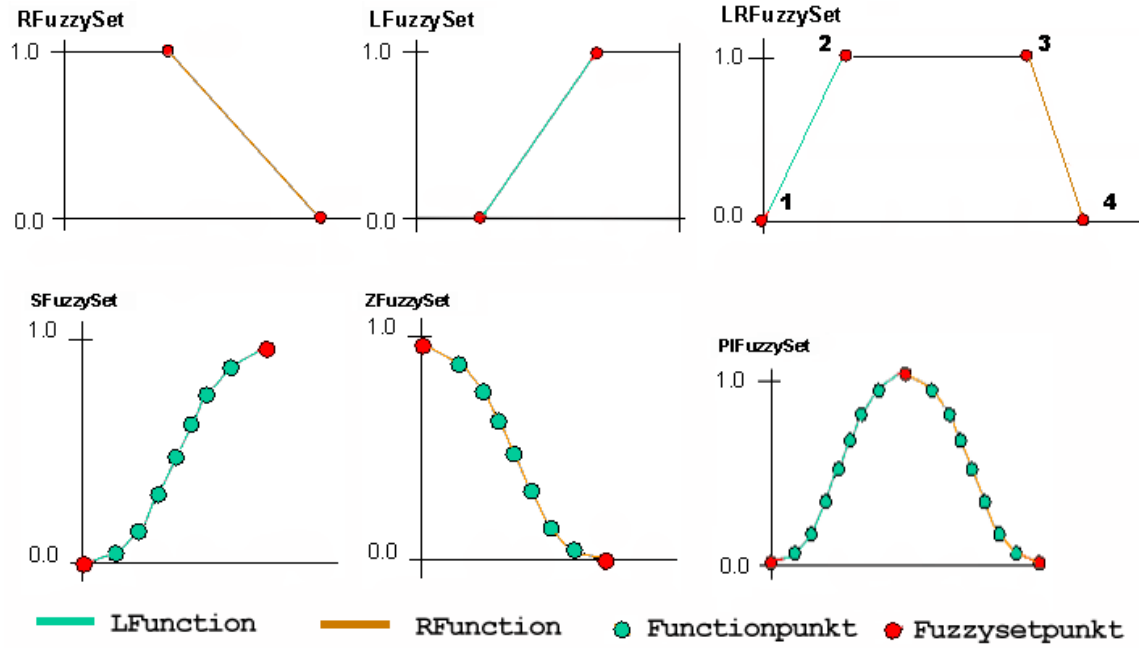


Abbildung 11.2.: Fuzzysets mit Funktionen

In den Pi-, S- und Z-Fuzzyset Klassen sind die "Function" Objekte fest implementiert und die mathematische Funktion zum Erzeugen der Punkte kann nicht verändert werden. Es lässt sich über den Konstruktor der entsprechenden Fuzzysetklasse nur festlegen, wie viele Punkte erzeugt werden sollen. Den L- und R-Fuzzysetklassen kann ein RFunction bzw. LFunction Objekt im Konstruktor übergeben werden. Wird kein "Function" Objekt übergeben, wird eine lineare Funktion verwendet. Dem Konstruktor der LR-Fuzzyset Klasse muss je ein LFunction und ein RFunction Objekt übergeben werden. Für den Fall, dass ein LR-Fuzzyset mit linearen Funktionen angelegt werden soll, ist ein Fuzzyset vom Typ Trapezoid-Fuzzyset zu verwenden(siehe Abb. D).

Generieren der Punkte

Um die Punkte zu erzeugen, benutzt die *RFunction* bzw. *LFunction* Klasse die freie Bibliothek *jep-2.3.0.jar* [JEP]. Diese Bibliothek besitzt die Fähigkeit, eine mathematische

Funktion, welche in einen String enthalten ist, zu parsen und zu berechnen. Die geparsten Funktion kann unbekannte Variable enthalten. Für diese Variable können Werte übergeben und der Funktionswert des gesamten Ausdrucks berechnet werden. Die Fähigkeit der Bibliothek wird dazu benutzt, die Y-Werte der Fuzzysetpunkte zu berechnen. Dem Konstruktor *RFunction* bzw. *LFunction* Klasse werden dabei 4 Parameter übergeben:

- String **expresion**: Dieser String enthält einen mathematischen Ausdruck mit einer unbekannten Variablen "x".
- Double **min**: Dieser Wert gibt an, ab welchem X-Wert die in "expresion" übergebene Funktion genutzt wird.
- Double **max**: Dieser Wert gibt an, bis zu welchem X-Wert die in "expresion" übergebene Funktion genutzt wird.
- Integer **numpoints**: Dieser Wert gibt an, wie viele Punkte generiert werden sollen.

Der Konstruktor der "Function" Klasse parst mit Hilfe der *jep-2.3.0.jar* Bibliothek den im Parameter "expresion" übergebenen String und prüft, ob er eine gültige mathematische Funktion mit einer unbekannten Variablen "x" enthält. Der Bereich zwischen "min" und "max" wird in "numpoints" Teilstrecken unterteilt. Zu jedem X-Wert, der das Ende einer Teilstrecke markiert, wird der Y-Wert der Funktion berechnet. Die X- und Y-Werte werden dem Fuzzyset übergeben, in welchem das "Function" Objekt genutzt wird. Die Werte der übergebenen Punkte werden im Fuzzyset so transformiert, dass sie an der Stelle liegen, die durch eine Funktion definiert werden soll (siehe Abb. 11.2).

Beispiel:
Soll beispielsweise ein LFuzzySet Objekt erzeugt werden, welches statt eines linearen einen quadratischen Anstieg besitzt, muss dem Konstruktor des LFuzzySets ein LFunction Objekt übergeben werden. Das LFunction Objekt berechtet die Punkte mit quadratischem Anstieg. Beim Anlegen des LFunction Objektes muss dem Konstruktor der LFunction eine quadratische Funktion(z.B.: "x$\hat{2}$") übergeben werden. Der Wert "min" kann auf 0 und der Wert "max" zum Beispiel auf 1 gelegt werden. Wichtig ist dabei, dass bei der Funktion ("x$\hat{2}$") der "min" Wert nicht kleiner als 0 ist, da das LFunction Objekt erwartet, dass das lokale Minimum des Y-Wertes im Bereich vom "min" und "max" bei "min" liegt (und das lokale Maximum bei "max"). Bei der RFunction Klasse ist dieser Sachverhalt genau entgegengesetzt. Das LFunction Objekt zerlegt nun die Strecke zwischen min und max in gleiche Teile, wobei sich die Anzahl der Teile nach dem Parameter "numpoints" richtet. Zu diesen X-Werten werden mit Hilfe der *jep-2.3.0.jar* Bibliothek nun die Y-Werte der Funktion ("x$\hat{2}$") berechnet. Da die vom LFunction Objekt berechneten Koordinaten nicht mit den Koordinaten des LFuzzySet Objekts übereinstimmen, werden die X-Werte transformiert und die Y-Werte auf den Bereich zwischen 0 und 1 normiert. Dies ist notwendig, damit sich die Punkte des LFunction Objektes im L-Fuzzyset an der richtigen Stelle befinden.

11.3. Exceptions

Im ursprünglichen Datenmodell wurde die Fehlerbehandlung zum Teil von den NRC-Klassen vorgenommen. Ohne die NRC-Objekte wurde es notwendig, die Fehlerbehandlung im Datenmodell neu zu implementieren und eigene Exceptionklassen anzulegen. Alle Exceptions des Datenmodells sind nun von der Klasse FuzzyException abgeleitet.

12. Simulator

Ziel war es, einen Simulator zu implementieren, der für möglichst viele Aufgaben genutzt werden kann. Der Simulator ist Teil der FuzzyIDE und wurde nicht als Plugin realisiert. Der Hauptgrund hierfür war die Schaffung eines eindeutigen Standards, auf den die Programmierer von Plugins aufbauen können. Um den Simulator zu nutzen, werden Plugins benötigt. Diese werden zur Generierung des Inputs, Visualisierung sowie zur Auswertung der Ergebnisse des Fuzzy-Systems benötigt. Der Simulator bietet lediglich die Fähigkeiten, Fuzzy-Berechnungen durchzuführen und die Plugins aufzurufen. Mit Hilfe der Plugins ist es möglich, die Umgebung zu simulieren, in der das Fuzzy-System eingesetzt werden soll oder es kann mit Hilfe der Plugins ein existierendes System angesteuert werden.

12.1. Achitektur

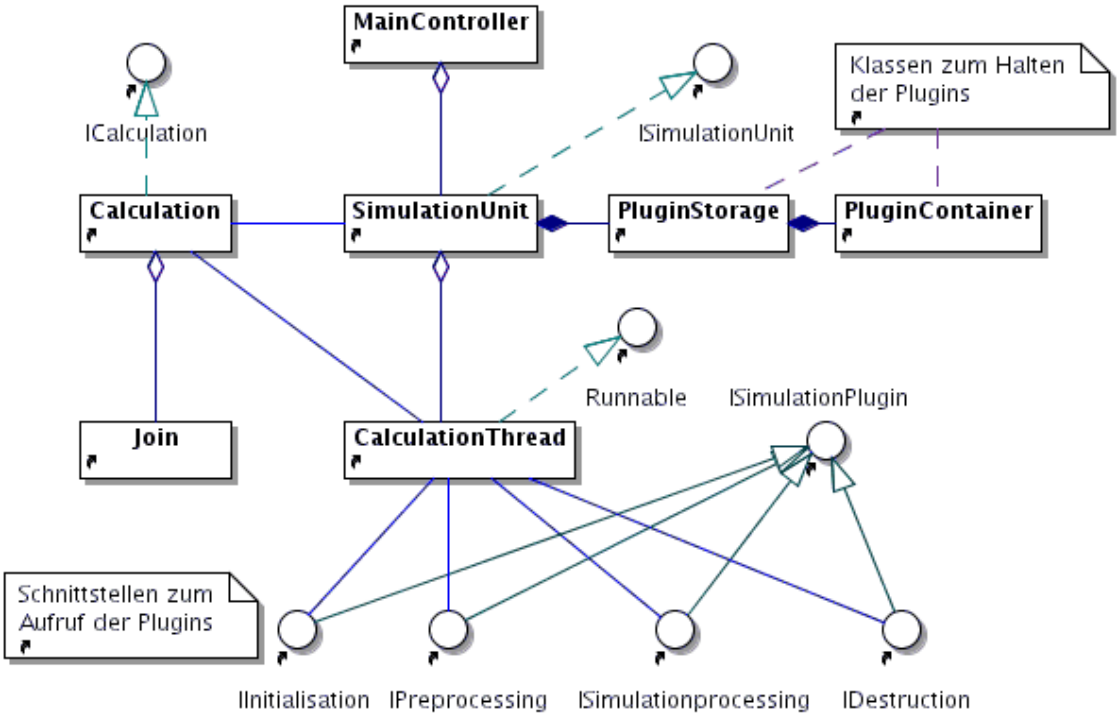


Abbildung 12.1.: Klassendiagramm Simulator

12.1.1. Wichtige Klassen

SimulationUnit
Die SimulationUnit stellt die zentrale Klasse des Simulators dar. Über die Methoden dieser Klasse kann die Simulation gestartet und gesteuert werden. Außerdem wird über diese Klasse die Fehlerverwaltung des Simulators geregelt (siehe Abs. 12.3).

PluginStorage
Diese Klasse enthält Informationen zu den Plugins, die der Simulator verwenden kann. Der Pluginmanager prüft beim Laden der Kasse, ob sie das Interface ISimulatorPlugin besitzt. Ist dies der Fall, wird eine Referenz des Plugin Objektes an den PluginStorage Member der SumulationsUnit übergeben. Damit ist das Plugin dem Simulator bekannt und kann verwendet werden.

PluginContainer
Die Klasse PluginStorage besitzt vier Objekte der Klasse PluginContainer. Ein PluginContainer Objekt enthält jeweils alle Referenzen zu einer Art von Pluginschnittstelle (IInitialisation usw., siehe Abs. 12.2.1) und die Information darüber, ob diese Schnittstelle des Plugins in der Simulation benutzt werden soll.

Calculation
Ein Objekt dieser Klasse wird beim Start der Simulation erstellt. Der Konstruktor der Klasse erhält als Parameter eine Referenz auf das aktuelle Datenmodell. Er erstellt aus dem aktuellen FuzzyIDE Datenmodell das NRC-Modell, welches für die Berechnungen benötigt wird. Die gesamten Fuzzy-Berechnungen des Simulators werden von dieser Klasse ausgeführt. Die wichtigste Methode der Klasse ist die "calculateStep" Methode, welche einen Schritt des Fuzzy-System berechnet. Dies ist die einzige Klasse der FuzzyIDE, die NRC-Objekte verwendet.

Join
Die Klasse Join ist eine Hilfsklasse für die Klasse Calculation. Sie dient dem Zuordnen von NRC-Objekten zu den entsprechenden FuzzyIDE Datenmodellobjekten. In dieser Klasse werden zwei Objektvektoren in Beziehung gesetzt. Ein Vektor enthält die Datenmodell Objekte, der andere die entsprechenden NRC-Objekte. Es ist somit möglich, über ein Datenmodellobjekt auf das entsprechende NRC-Objekt zuzugreifen und umgekehrt.

CalculationThread
Die Klasse implementiert das Interface Runnable mit der Methode "run()" und kann somit von einen Objekt vom Typ *java.lang.Thread* in einem Thread aufgerufen werden. Diese Klasse ruft in einen Thread die Plugins und die "calculateStep" Methode der Calculation Klasse auf.

12.2. Pluginanbindung

12.2.1. Interfaces

Um die Plugins an den Simulator anbinden zu können, wurde eine Reihe von neuen Interfaces definiert, welche von den Plugins implementiert werden müssen, damit der Simulator diese nutzen kann. Die Interfaces werden hier kurz beschrieben.

Interface ISimulationPlugin

Diese Schnittstelle bildet die Voraussetzung, dass die FuzzyIDE das Plugin als Simulatorplugin erkennt. Der Pluginmanager (oder Extension Point Manager) prüft beim Laden des Plugins, ob es die ISimulationPlugin Schnittstelle besitzt. Ist dies der Fall, erkennt es der Pluginmanager als ein Simulatorplugin und der Menüeintrag des Plugins erscheint im Menüpunkt "Simulator Plugins" der FuzzyIDE. Eine Referenz des Plugins wird an den "PluginStorage" Member der "SimulationsUnit" übergeben. Somit kann der Simulator das Plugin benutzen.
Das Interface stellt dem Simulator drei Methoden bereit. Die Methode *" initPlugin"* wird vom Simulator vor Beginn der Simulation aufgerufen, um im Plugin Initialisierungen durchzuführen. Die zweite Methode ist die *"destroyPlugin "* Methode. Sie wird nach dem Beenden der Simulation aufgerufen. In dieser Methode kann das Plugin die anfallenden Aufräumarbeiten erledigen. Die letzte Methode des Interfaces ist die Methode *"multiActivation"*. Der Rückgabewert der Funktion gibt an, wie die weiteren Schnittstellen des Plugins verwendet werden (siehe Abs. 12.2.2).

Die vier Interfaces IInitialisation, IPreprocessing, ISimulationprocessing und IDestruction sind von ISimulationPlugin abgeleitet. Die Interfaces stellen jeweils eine Funktionen zur Verfügung. Diese Funktion wird durch den Simulator zu einem gewissen Zeitpunkt aufgerufen. Der Zeitpunkt, an dem die Funktionen der einzelnen Schnittstellen aufgerufen werden, ist in Abbildung 12.2 zu sehen. In den Methoden der vier Schnittstellen können die Plugins die notwendigen Aufgaben während der Simulation verrichten.

Ein Plugin kann eine oder mehrere dieser Schnittstellen implementieren. Ob der Simulator die Schnittstellen eines Plugins einzeln oder nur gemeinsam verwenden kann, hängt vom Rückgabewert der Methode "multiActivation" der ISimulationPlugin Schnittstelle ab. Gibt das Plugin in dieser Methode *true* zurück, können die Schnittstellen des Plugins nur gemeinsam aktiviert oder deaktiviert werden (siehe Abs. 12.2.2).Ist der Rückgabewert *false*, können die Schnittstellen des Plugins einzeln genutzt werden.

Interface ICalculation

Den Methoden der vier Schnittstellen (IInitialisation, IPreprocessing, ISimulationprocessing, IDestruction), die vom Simulator während der Simulation gerufen werden, wird ein Objekt mit der Schnittstelle ICalculation übergeben. Das übergebene Objekt ist das Calculation Objekt, welches die Fuzzy-Berechnungen ausführt. Über die ICalculation Schnittstelle haben die Plugins die Möglichkeit, das Verhalten des Fuzzy-Systems zu beeinflussen und Informationen zum letzten Berechnungsschritt zu erhalten. Mit Hilfe der Methoden des Interfaces können die Regeln des Fuzzy-Systems an- und abgeschaltet und der Aktivierungsgrad der Regeln beim letzten Berechnungsschritt des Fuzzy-Systems ausgegeben werden.

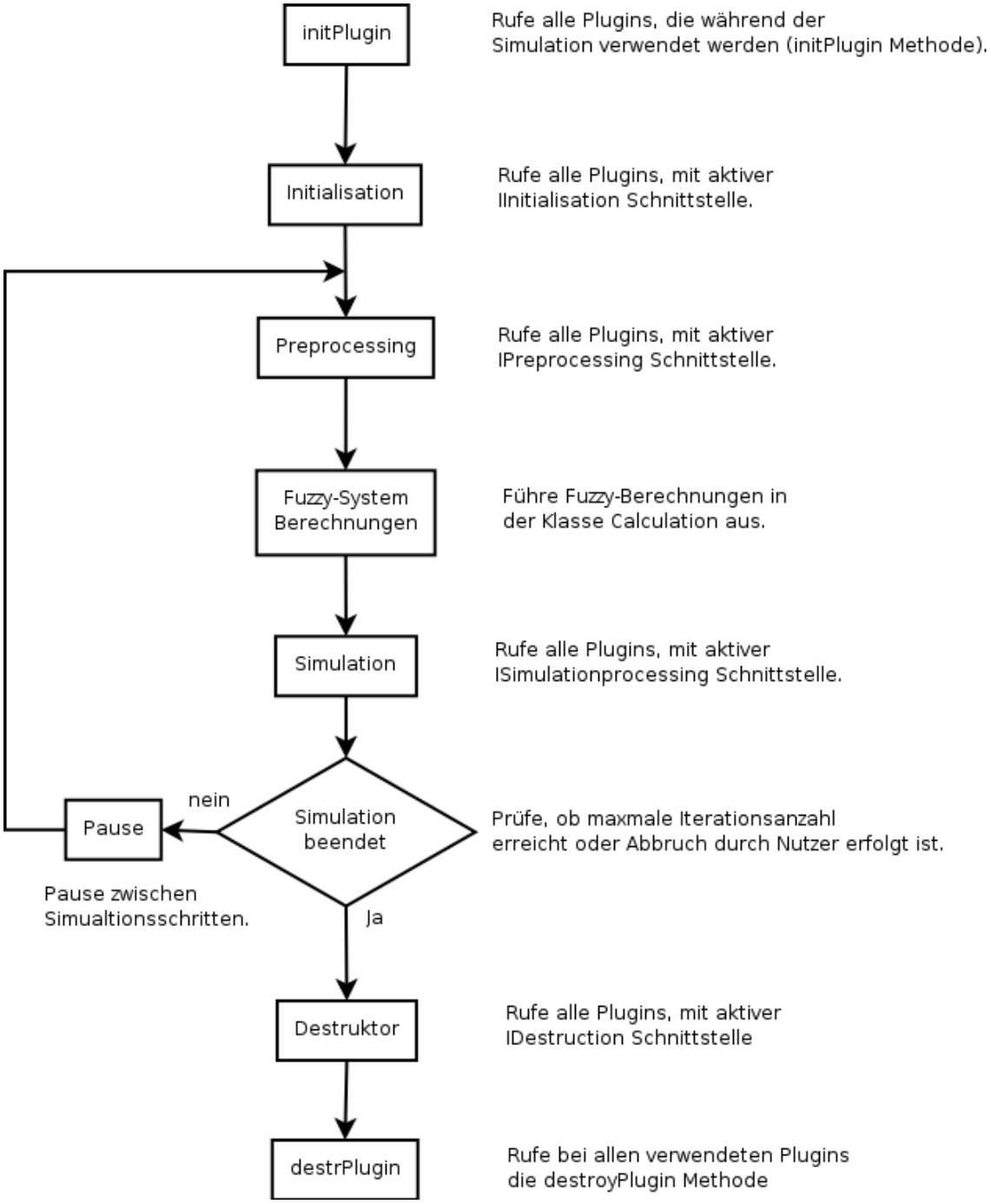


Abbildung 12.2.: Ablauf der Simulation

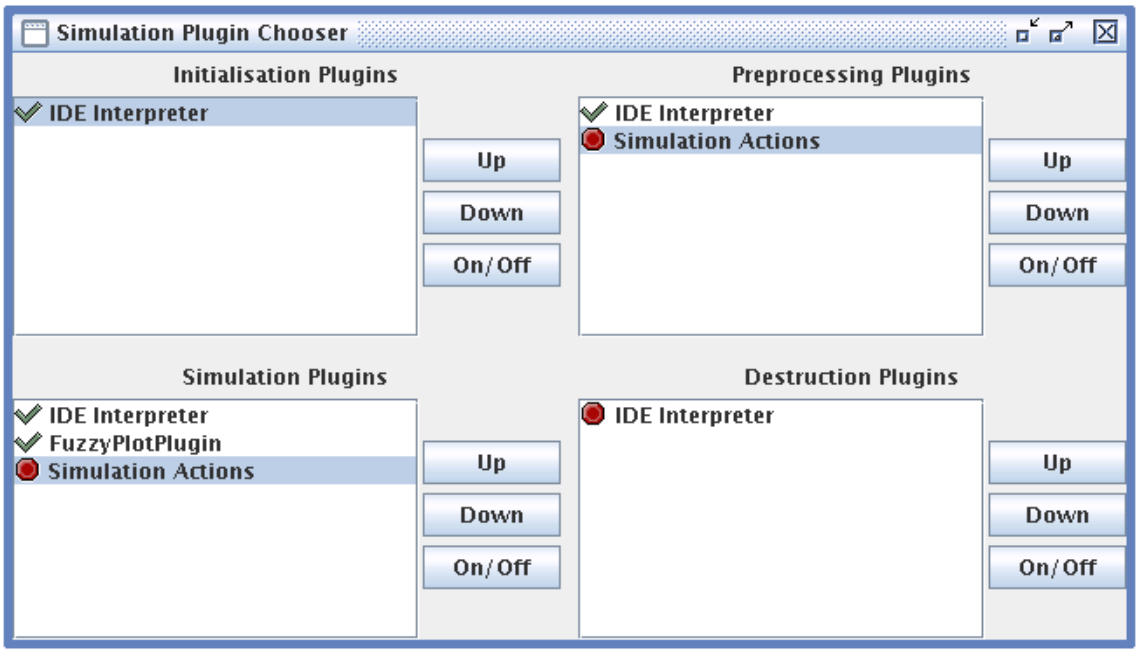


Abbildung 12.3.: Plugin Chooser Dialog

12.2.2. Plugin Chooser Dialog

Wie im Abschnitt 12.2.1 beschrieben, wird beim Laden der Plugins geprüft, ob ein Plugin die ISimulationPlugin Schnittstelle besitzt. Plugins, bei denen dies der Fall ist, werden dem "PluginStorage" der SimulationsUnit übergeben. Dieser prüft, welche der vier von ISimulationPlugin abgeleiteten Schnittstellen(IInitialisation usw.) das Plugin besitzt und ordnet Referenzen der einzelnen Schnittstellen des Plugins in die entsprechenden PluginContainer. Sollte also ein Plugin alle vier Schnittstellen implementieren, werden im Simulator vier Referenzen auf das Plugin Objekt gehalten, wobei sich die Referenzen im Typ unterscheiden(IInitialisation usw.).

Nachdem das Plugin geladen wurde, erscheinen die Schnittstellen des Plugins im "Plugin Chooser" Dialog (siehe Abb. 12.3). Der "Plugin Chooser" Dialog dient dem Auswählen der Plugins oder genauer gesagt, dem Auswählen der Schnittstellen, die während der Simulation benutzt Werden sollen. Der Dialog zeigt ein Plugin, sobald es mindestens eine der vier Schnittstellen (IInitialisation usw.) besitzt. Die Einordnung in die Listboxen ist abhängig von den Schnittstellen, die das Plugin implementiert. Das IDEInterpreter Plugin besitzt beispielsweise alle vier Schnittstellen und befindet sich daher auch in allen vier Listboxen. Über die Buttons "Up" und "Down" kann die Reihenfolge innerhalb einer Listbox verändert werden und somit auch die Reihenfolge, in der die Plugins durch den Simulator während der Simulation gerufen werden. Mit Hilfe des "On/Off" Buttons kann festgelegt werden, ob die entsprechende Schnittstelle des Plugins durch den Simulator verwendet wird.

So werden zum Beispiel in Abbildung 12.3 die IInitialisation, IPreprocessing und ISimulationprocessing Schnittstellen des IDEInterpreter-Plugins verwendet, die IDestruction Schnittstelle jedoch nicht. Für den Fall, dass es bei einem Plugin keinen Sinn macht, die Schnittstellen einzeln zu aktivieren, gibt es eine Möglichkeit, dies zu verhindern. Der

Rückgabewert der Methode "boolean multiActivation()", die jedes Plugin mit dem ISimulationPlugin Interface implementieren muss, gibt an, ob die Schnittstellen des Plugins einzeln aktiviert werden können oder nicht. Der Ablauf der Simulation mit den Einstellungen aus dem Screenshoot 12.3 ist im Sequenzdiagramm in Abbildung 12.7 zu sehen.

12.2.3. Simulator In- und Output

Da die Plugins dafür zuständig sind, den Input für den Simulator bereitzustellen und den Output des Simulators weiterzuverarbeiten, muss es den Plugins möglich sein, die Werte der Variablen zu lesen und zu setzen. Um den Zugriff möglichst einfach zu gestalten, können die Werte mit Hilfe der Datenmodellobjekte gesetzt und gelesen werden. Vor jedem Berechnungsschritt des Fuzzy-Systems, den die Klasse Calculation durchführt, werden die Werte aus den Variablen des Datenmodells ausgelesen und an die Objekte des NRC-Modells übergeben.

12.2.4. Plugin Observer

Während des Redesigns wurde von Mitgliedern des FuzzyIDE Team ein Extension Point Mechanismus zum Laden von Plugins implementiert. Er ersetzt den alten Pluginmanager der FuzzyIDE. Mit Hilfe des Extension Point Mechanismus ist es möglich, Plugins zur Laufzeit der FuzzyIDE zu laden und zu beenden. Dies brachte das Problem mit sich, dass die Anzeigen im Pluginchooser Dialog nicht aktuell blieben. Um dieses Problem zu lösen, wurde ein Observer Mechanismus für die Plugins implementiert, der den Pluginchooser Dialog auf aktuellem Stand hält, falls ein Plugin geladen oder beendet wird.

12.3. Fehlerbehandlung

Um die Fehler, die während der Simulation auftreten können, an einer Stelle zu behandeln, wurde eine zentrale Fehlerbehandlung für den Simulator implementiert. Eine zentrale Fehlerbehandlung gibt dem Nutzer die Möglichkeit festzulegen, welche Fehler zum Abbruch der Simulation führen und welche Fehler ignoriert werden sollen. Der Simulator kennt zwei unterschiedliche Arten von Fehlern (Exceptions):

1. Schwerwiegende Fehler, die immer zum Beenden der Simulation führen.
2. Fehler, die durch Einstellungen in der FuzzyIDE geblockt werden können.

12.3.1. Schwerwiegende Fehler

Bei schwerwiegenden Fehlern handelt es sich solche Fehler, die ein Weiterführen der Simulation unmöglich oder sinnlos machen. In Frage kommen dabei folgende Fehler:

- **Threadfehler:** Sie treten auf, wenn Start, Betrieb oder Beenden des Simulationsthreads nicht erfolgreich waren.
- **NRC-Modellfehler:** Sie treten auf, wenn aus dem Datenmodell der FuzzyIDE kein NRC-Modell erstellt werden kann.

- **Initialisierungsfehler:** Der Fehler tritt auf, wenn beim Initialisieren eines Plugins in der Methode "initPlugin()" eine Exception geworfen wird.
- **Destruktorfehler:** Beim Beenden der Simulation in der Methode "destroyPlugin()" tritt eine Exception auf.

Sollte einer dieser Fehler auftreten, wird der Simulationsthread beendet und die entsprechende Exception an die Klasse "SimulationControlFrame" des Packages *gui* weitergegeben. In dieser Klasse wird der Text der Exception dann in einem Dialog ausgegeben.

12.3.2. Behandelbare Fehler

Bei behandelbaren Fehlern kann der Nutzer festlegen, ob sie zu Abbruch der Simulation führen sollen. Dies geschieht mit Hilfe des "Simulation Errorhandling" Dialoges (siehe Abb. 12.4), in dem die behandelbaren Fehler ausgewählt werden können, welche zum Abbruch der Simulation führen sollen.

Über den Dialog "Simulation Errorhandling" (siehe Abb. 12.4) können die Exceptions ausgewählt werden, bei welchen die Simulation beendet wird. Der Dialog bietet auch die Möglichkeit, auf ganze Fehlergruppen zu reagieren. Zum Beispiel auf alle Fehler des Fuzzy-Systems oder auf solche, die in den Plugins auftreten können.

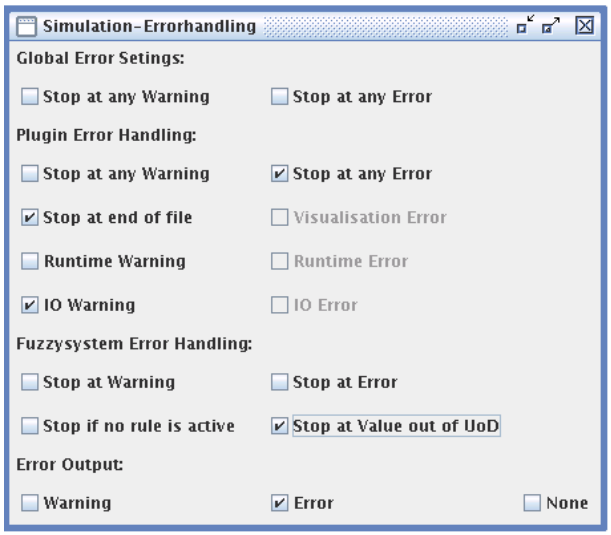


Abbildung 12.4.: Errorhandling Dialog

Zusätzlich kann in diesem Dialog auch der Umgang mit den Fehlern und Warnungen festgelegt werden, die nicht zum Abbruch der Simulation führen. Zur Auswahl stehen die Möglichkeit, die Fehler und Warnungen über "System.err" auszugeben oder die Ausgaben vollständig zu unterbinden.

Geblockt werden können die Exceptions der Klassen SimulationPluginException, SimulationFuzzyException. Exceptions der Klasse SimulationPluginException werden durch die Plugins an den Simulator gegeben. Diese Klasse enthält einen Parameter mit dem Typ des Fehlers. Erzeugt ein Plugin während der Simulation eine SimulationPluginException, kann es festlegen, welchen Fehlertyp diese Exception haben soll. Der Simulator prüft dann,

ob der Fehlertyp der Exception zum Abbruch der Simulation führt oder nicht. Solle der Fehler nicht zum Abbruch der Simulation führen, wird je nach Einstellung der Text der Exception ausgegeben oder nicht. Danach wird die Simulation fortgesetzt.

Auf die gleiche Weise erfolgt die Behandlung der Exceptions vom Typ "Simulation-FuzzyException". Einziger Unterschied ist hierbei, dass durch diese Exceptionklasse die Fehler des Fuzzy-System behandelt werden. Die Exception wird also von dem "Calcualtion" Objekt ausgelöst, in welchem die Fuzzy-Berechnungen durchgeführt werden. Für den Fehlertyp-Parameter stehen bei dieser Exception andere Fehlertypen zur Auswahl.

12.4. Inferenz-Einstellungen

In den Inferenz-Einstellungen kann festgelegt werden, welche Operatoren das Fuzzy-System für die Berechnungen benutzen soll. Die Inferenz-Einstellungen können über den Dialog "Inference Unit - Settings"(siehe Abb. 12.5) verändert werden.

Über diesen Dialog können die Operatoren für Aggregation, Implikation, Akkumulation und Defuzzyfizierung (siehe Abs. 2.2) des Fuzzy-Systems festgelegt werden. Der Aggregations-Operator bezieht sich auf die Verknüpfung der einzelnen Bedingungen, die als Objekte der Klasse FuzzyValue vorliegen, jedoch nicht auf die Bedingungen innerhalb eines FuzzyValue Objektes. Die Einstellungen müssen vor Beginn der Simulation festgelegt werden und lassen sich während einer Simulation nicht ändern.

Die Inferenz-Einstellungen des Fuzzy-Systems sind Teil des Datenmodells und in der Klasse FuzzySystem gespeichert. Auf der Basis dieser Einstellungen wird vom Simulator das NRC-Modell erzeugt, welches die Berechnungen durchführt.

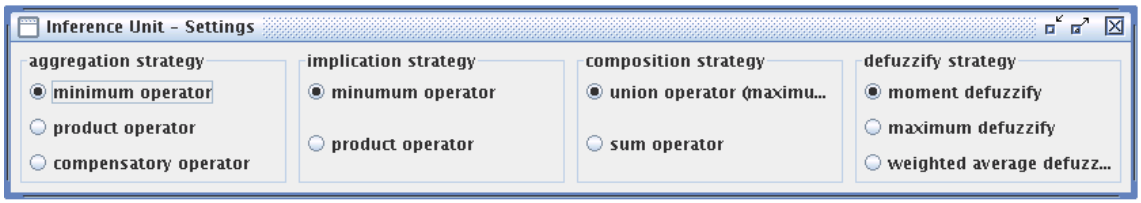


Abbildung 12.5.: Screenshoot Inferenz-Einstellungen

12.5. Simulationssteuerung

Vor dem Start der Simulation sollten das Fehlerverhalten(siehe Abs. 12.3), die verwendeten Plugins(siehe Abs. 12.2.2) sowie die Inferenz-Einstellungen(siehe Abs. 12.4) festgelegt werden. Zum Steuern der Simulation dient der "Simulation Control" Dialog(siehe Abb. 12.6). Er bietet dem Nutzer eine Reihe von Einstellungen, das Verhalten des Simulators festzulegen. Über den Schieberegler oder die linke Textbox lässt sich die Zeit zwischen zwei Simulationsschritten festlegen. Die Angabe der Zeit zwischen den Schritten erfolgt in Millisekunden. Mit Hilfe der rechten Textbox lässt sich die Anzahl der Iterationsschritte festlegen, welche der Simulator durchführen soll bevor die Simulation angehalten wird. Wird eine 0 eingegeben, wird die Simulation unbegrenzt fortgeführt.

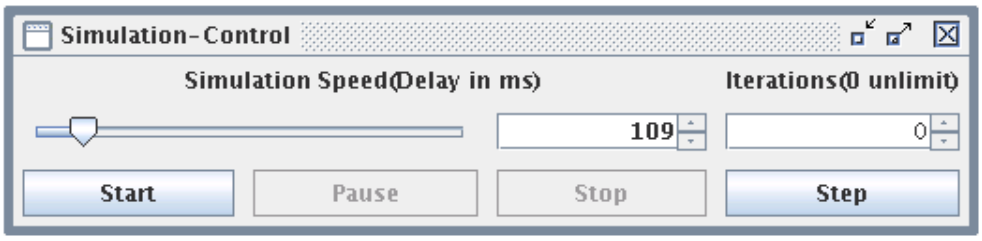


Abbildung 12.6.: Screenshoot Simulation Control Dialog

Mit Hilfe des "Start" Buttons kann der Simulationsthread gestartet werden. Der "Step" Buttons links führt einen einzelnen Simulationsschritt aus. Der Button "Pause" dient dem Unterbrechen der Simulation, ohne sie zu beenden. Sollte die Simulation unterbrochen sein, kann sie mit Hilfe desselben Buttons fortgesetzt werden.

12.5.1. Startabläufe

Der Start der Simulation beginnt mit dem Betätigen des "Start" Buttons im "Simulation Control" Dialog. Ist das geschehen, werden folgende Schritte ausgeführt, die zum Start der Simulation führen:

1. Aufruf der Methode "start()" in der Simulationsunit.
2. Erzeugen eines neuen Calculation Objektes. Hierbei wird aus dem aktuellen Datenmodell das NRC-Modell erstellt.
3. Erzeugen eines CalculationThread Objektes.
4. Zusammenstellen aller aktivierten Pluginschnittstellen aus dem PluginStorage und Übergabe dieser an das CalculationThread Objekt.
5. Erzeugen eines java.lang.Thread Objektes, welches das CalculationThread Objekt benutzt.
6. Starten des Simulationsthreads (java.lang.Thread Objekt).

12.5.2. Simulationsabläufe

Bevor die Simulation beginnt, wird bei jedem Plugin, welches der Simulator in der aktuellen Einstellung verwendet (siehe Abs. 12.2.1), die Methode "initPlugin" aufgerufen. Falls in einem der Plugins dabei eine Exception auftritt, wird die Simulation beendet.

Im ersten Schritt der Simulation ruft der Simulationsthread die Plugins mit den aktiven IInitialisation-Schnittstellen auf. Ist dies geschehen, fährt der Simulator wie bei jedem der folgenden Simulationsschritte fort und führt folgende Aufgaben aus:

1. Der Simulationsthread ruft die Plugins mit den aktiven IPreprocessing Schnittstellen auf(in der festgelegten Reihenfolge siehe Abs. 12.2.1).
2. Der Simulationsthread ruft die "calculateStep" Methode der Calculation Klasse auf, um einen Schritt des Fuzzy-Systems zu berechnen.

3. Der Simulationsthread ruft die Plugins mit den aktiven ISimulationprocessing Schnittstellen auf(in der festgelegten Reihenfolge siehe Abs. 12.2.1).

12.5.3. Ende der Simulation

Zum Abbruch der Simulation kann es aus mehreren Gründen kommen. Erstens kann der Nutzer die Simulation über den "Simualtion Control" Dialog beenden. Zweitens, die eingestellte maximale Anzahl der Iterationen wird ereicht. Drittens kann während der Simulation ein Fehler auftreten, der zum Abbruch führt. Im ersten Fall werden beim Beenden der Simulation folgende Aufgaben umgesetzt:

1. Der Simulationsthread ruft die Plugins mit den aktiven IDestruction Schnittstellen auf(in der festgelegten Reihenfolge siehe Abs. 12.2.1).
2. Die Simulationsunit ruft die "destroyPlugin()" Methode aller Plugins auf, die während der Simulation verwendet wurden.
3. Die Observer des Simulators werden über das Beenden der Simulation informiert.
4. Der Thread wird beendet.

Im Fall des Erreichens der eingestellten maximalen Zahl der Iterationen werden nur die Punkte 1, 3 und 4 ausgeführt. Dies ist sinnvoll, weil es dem Nutzer die Möglichkeit gibt, eine neue maximale Iterationsanzahl einzustellen und die Simulation fortzusetzen. Im Fall eines Simulationsabbruches wegen eines Fehlers werden die Schritte 2,3 und 4 ausgeführt. Das Aufrufen der Plugins mit einer aktivierten IDestruction-Schnittstelle macht keinen Sinn, da bereits ein Fehler aufgetreten ist. Aus diesem Grund kann das korrekte Aufrufen der Plugins nicht mehr gewährleistet werden.

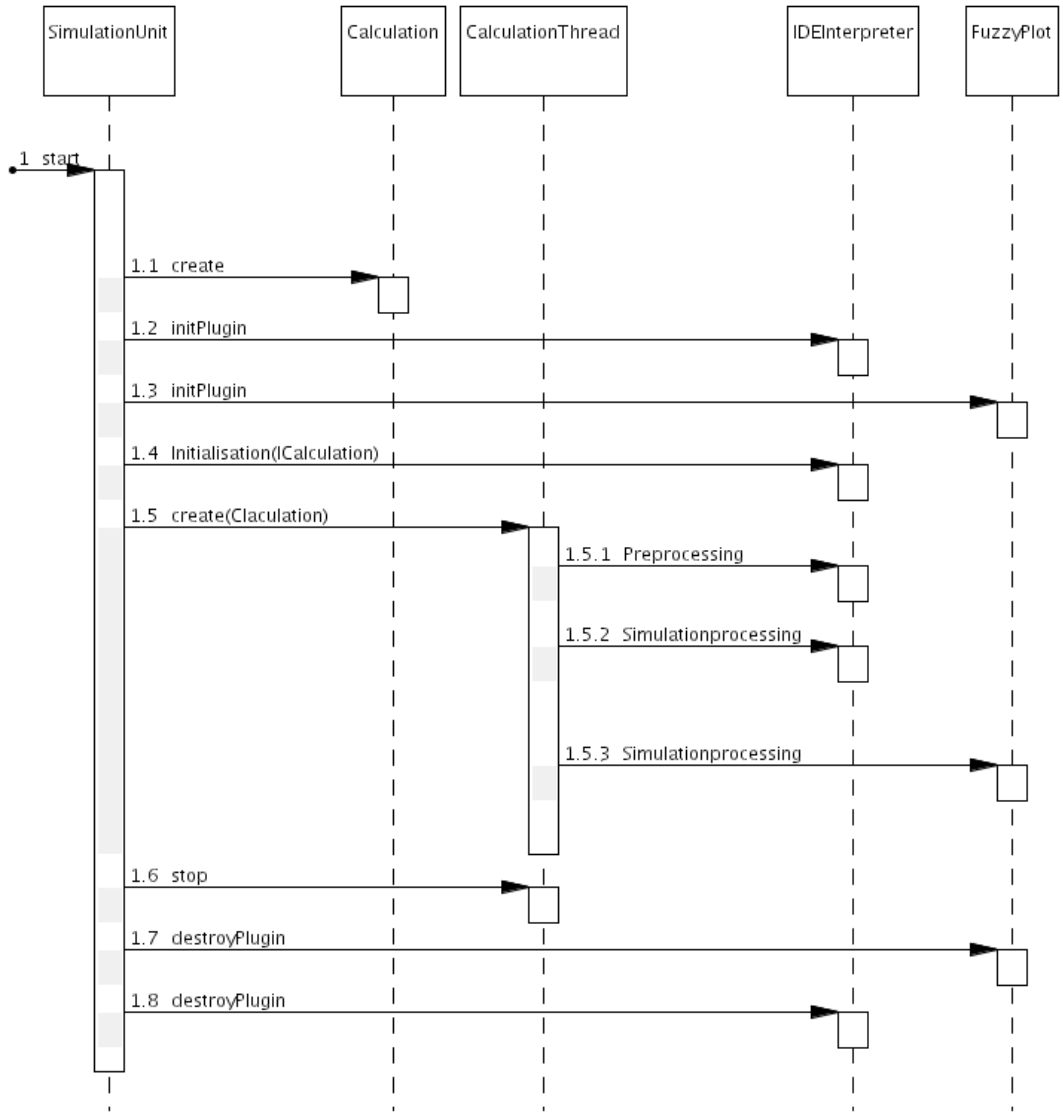


Abbildung 12.7.: Sequenzdiagramm Simulator

13. IDEInterpreter-Plugin

13.1. Einleitung

Um den Simulator der FuzzyIDE Entwicklungsumgebung möglichst effektiv nutzen zu können, werden Möglichkeiten benötigt, die Inputvariablen des Fuzzy-Systems zu setzen und den Output des Systems weiter zu bearbeiten. Da ein Fuzzy-System im Normalfall in ein Programm eingebunden ist, welches die Inputs setzt und die Outputs weiterverarbeitet, sollte dazu auch in der FuzzyIDE eine Möglichkeit geschaffen werden, Programmcode auszuführen. Ziel ist es, dem Nutzer eine Skriptsprache zur Verfügung zu stellen, welche die Werte des Fuzzy-Systems zur Laufzeit der Simulation setzen und ausgeben kann.

13.1.1. Anforderungen

Aus dem oben genannten Vorgaben ergeben sich folgende Anforderungen, welche das Plugin erfüllen muss:

- Unterstützung aller Schnittstellen des Simulators (siehe Abs. 12.2.1). Dies ist notwendig, damit das IDEInterpreter-Plugin Ausgangswerte setzen, Berechnungen durchführen und Ergebnisse auswerten kann.
- Lesen und Setzen der Werte des Datenmodells. Dies gilt für Fuzzy In- und Outputvariablen, sowie für die temporären Variablen des Datenmodells.
- Deklarierung lokaler Variablen. Da die Berechnungen häufig das Speichern von Zwischenwerten erfordert, ist es sinnvoll, diese lokal im Plugin zu deklarieren.
- Das Plugin muss die wichtigsten Sprachelemente einer Programmiersprache unterstützen: Bedingte Sprünge (if,if else), Schleifen (for, while, do while), Aufrufen von vordefinierten Funktionen (sin, cos usw.).
- Die Funktionen des Plugins sollen einfach erweiterbar sein, ohne Änderungen an der Skriptsprache vornehmen zu müssen.
- Das Plugin soll für jede der vier Schnittstellen des Simulators ein eigenes Programm erzeugen. So können die Schnittstellen auch einzeln genutzt werden.
- Der Quellcode soll während der Laufzeit der FuzzyIDE Entwicklungsumgebung geändert und angewendet werden können.

13.1.2. Vorüberlegungen

Um den in der Skriptsprache verfassten Quellcode in ausführbaren Programmcode umzusetzen, wird ein Compiler benötigt. Gewöhnlich erzeugt ein Compiler Maschinencode zum Ausführen oder Zwischencode für eine hypothetische Maschine, die von einem Interpreter

realisiert wird. So erzeugt beispielsweise der Javacompiler *javac* Programmcode für eine virtuelle Maschine, die mit Hilfe des Javainterpreters realisiert wird. Da der Skriptsprachen Quellcode zur Laufzeit der FuzzyIDE angewendet werden soll, muss er vom Javainterpreter ausgeführt werden können. Eine Möglichkeit wäre es, den in der Skriptsprache verfassten Quellcode in Javacode umzusetzen und diesen mit Hilfe des Java Compilers in Java Programmcode umzuwandeln. Der Java Programmcode ließe sich dann zur Laufzeit über den Java Reflexion Mechanismus [Ess02] laden und ausführen. Dieses Vorgehen bringt jedoch das Problem mit sich, dass die Klassen erst auf die Festplatte geschrieben und danach über einen Aufruf des Javacompilers übersetzt werden müssen. Dieser muss aber nicht auf jeden Rechner vorhanden sein.

Eine bessere Alternative ist die Umsetzung des Skriptsprachenprogramms direkt in in Javaobjekte. Hierbei erzeugt der Compiler einen Baum von Objekten, welcher das gesamte Programm repräsentiert. Zum Ausführen des Programms wird bei dem Objekt, welches an der Wurzel des Baumes steht, die Methode *execute()* aufgerufen. Diese wiederum ruft die *execute()* Methode der Objekte der nächsten Schicht auf und so weiter. Auf diese Weise ist es problemlos und schnell möglich, zur Laufzeit der FuzzyIDE ausführbaren Code zu erzeugen und diesen anzuwenden. Der Nachteil besteht darin, dass für jedes Sprachkonstrukt der Skriptsprache eine eigene Javaklasse vorhanden sein muss, um von ihr Objekte anzulegen. So müssen beispielsweise Klassen für einen *Block* oder für eine *IF* Anweisung implementiert werden. Dies führt schnell zu einer großen Anzahl von Javaklassen. Trotz der Nachteile, ist diese Methode der oben beschriebenen vorzuziehen.

13.2. Anwendung

Um das Plugin während der Simulation zu benutzen, muss vor Beginn der Simulation der Programmcode eingegeben werden. Die Codeeingabe erfolgt im "IDEInterpreter" Dialog (siehe Abb. 13.1). Der Dialog besitzt vier Textfelder, auf die über Kartenreiter zugegriffen werden kann. In diesen Textfeldern kann der Programmcode eingegeben werden, der während der Simulation ausgeführt werden soll. Jedes der Textfelder entspricht dabei einer Schnittstelle des Plugins, die vom Simulator während der Simulation aufgerufen wird. Das Textfeld im Reiter "Initialisation Code" enthält somit den Programmcode, der über die Methode der IInitialisation Schnittstelle aufgerufen wird und so fort. Soll das Plugin in der Simulation genutzt werden, muss es im "Plugin Chooser" Dialog (siehe Abs. 12.2.2) aktiviert werden. Es ist dabei möglich, die vier Schnittstellen des Plugins einzeln zu aktivieren. So kann beispielsweise festgelegt werden, dass nur der "Inititialisation Code" und der "Simulation Code" ausgeführt werden. Wurde eine Schnittstelle aktiviert, ruft der Simulator beim Start der Simulation die Methode "initPlugin()" auf. In dieser Methode wird der Quellcode aller vier Textfelder kompiliert und jeweils ein Objektbaum(Programm) erzeugt, der während der Simulation ausgeführt wird. Sollte beim Kompilieren ein Fehler auftreten, wird eine Exception geworfen und der Start der Simulation abgebrochen.

13.3. Die Sprache

Die für das Plugin entworfene Skriptsprache wurde speziell an die FuzzyIDE angepasst. Um den Nutzer eine längere Einarbeitung zu ersparen, wurde die Syntax an die Programmiersprache C angelehnt.

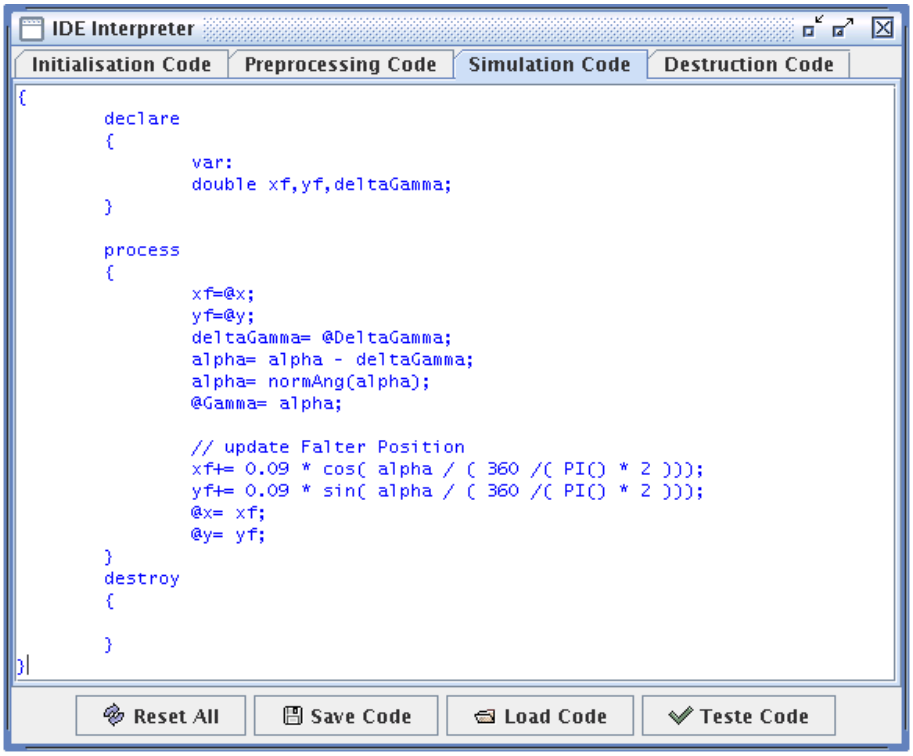


Abbildung 13.1.: Screenshot IDEInterpreter

13.3.1. Aufbau

In diesem Abschnitt soll der Aufbau des Quellcodes erklärt werden. Die Erläuterung beschränkt sich hierbei auf den prinzipiellen Aufbau. Die gesamte Grammatik der Sprache ist in der BNF-Notation im Anhang A.1 nachzulesen.

Datentypen

Die Skriptsprache unterstützt drei Arten von Datentypen: *double, int, File*. Variablen dieser Typen können in den Sektionen *global* und *declair* angelegt werden. Die "int" Variablen werden vom Compiler intern in Variablen des primitiven Datentyps "long" umgesetzt. Sollte ein Double-Wert einer Integer-Variablen zugewiesen werden, wird er wie in normalen Javaprogrammen konvertiert. Der Variablentyp "File" bietet die Möglichkeit, über das Skriptprogramm Dateien zu lesen und zu schreiben. Der Typ wird zwar in der Sprache unterstützt, die Funktionen zum Lesen und Schreiben von Dateien sind jedoch noch nicht vollständig implementiert.

Alle Variablen, die im Quelltext deklariert werden, sind ausschließlich im Plugin bekannt. Die Variablen des Plugins sind unabhängig von den Variablen des Datenmodells.

Quellcode

Ein Quellcode der Skriptsprache besteht aus vier Blöcken. Nicht jeder Block muss vorhanden oder ausgefüllt sein. Wichtig ist jedoch, dass die Reihenfolge der Blöcke eingehalten

wird.

1. **global:** In diesem Block können globale Variablen vom Typ *int*, *double* oder *File* angelegt werden. Die hier definierten Variablen sind auch in den Quelltexten der nachfolgenden Schnittstellen bekannt. Globale Variablen sollten im Quelltext der *InitialisationCode* Schnittstelle deklariert werden, da sie dann allen Schnittstellen bekannt sind. Wird eine globale Variable beispielsweise im Quelltext für die Schnittstelle *Preprocessing* deklariert, ist sie im Quelltext *InitialisationCode* unbekannt.

   Den Variablen kann bereits während der Definition ein Wert zugewiesen werden. Dies ist mit einer Zahl oder einer Formel möglich, solange diese keine unbekannten Variablen enthält. Sollte der Wert einer Variablen nicht festgelegt werden, wird sie mit dem Wert 0 initialisiert. Achtung! Der Compiler gibt keine Fehlermeldung aus, sollte aus einer nicht initialisierten Variable gelesen werden!

2. **declair:** In diesem Block können lokale Variablen vom Typ *int*, *double* oder *File* angelegt werden. Sie sind nur dem Programm der betreffenden Schnittstelle bekannt. Sollten eine lokale und eine globale Variable gleichen Namens definiert sein, überdeckt die lokale Variable in ihrem Bereich die globale Variable. Den lokalen Variablen kann ebenso wie den globalen Variablen während ihrer Definition ein Wert zugewiesen werden.

3. **process:** Dieser Block enthält den Programmcode, der während der Simulation über eine der Schnittstellen des Plugins vom Simulator aufgerufen wird.

4. **destroy:** Dieser Block enthält den Programmcode, der nach dem Beenden der Simulation ausgeführt wird. Wie im Abschnitt 12.5.3 beschrieben, wird vom Simulator nach dem Beenden der Simulation bei allen verwendeten Plugins die Methode "destroyPlugin()" aufgerufen. In dieser Methode wird der Programmcode des "destroy" Blocks ausgeführt. Der Block kann genutzt werden, die verwendeten Dateien zu schließen oder die Werte von Variablen auszugeben.

Sprachkonstrukte

Die Skriptsprache unterstützt folgende Sprachelemente in ihrer Grammatik:

- **if(** Bedingung **)** Anweisung
- **if(** Bedingung **)** Anweisung **else** Anweisung
- **do** Anweisung **while(** Bedingung **);**
- **while(** Bedingung **)** Anweisung
- **for(** Anweisung **;** Bedingung **;** Anweisung **)** Anweisung
- **return;**

Die Anweisungen können aus einem einzelnen Befehl oder einem Block bestehen, welcher wiederum eine Gruppe von Anweisungen enthält. Die Bedingungen können mit den logischen Opertoren "&&" (logisches und) und "||" (logisches inklusives oder) miteinander verknüpft werden. Es ist es möglich, Zahlen und Funktionen als Bedingungen zu nutzen,

ähnlich wie in der Sprache C. Eine Bedingung gilt als nicht erfüllt, wenn die Zahl oder der Rückgabewert einer Funktion gleich Null ist. In allen anderen Fällen gilt die Bedingung als erfüllt.

Operatoren

Die Sprache kennt folgende binäre Operatoren: +, -, *, / und %, die booleschen Operatoren &&, ||. Die Sprache kennt zusätzlich die für C typischen Zuweisungsoperatoren +=, -=, *=, /=, %=, welche das Ergebnis des Ausdrucks der Variablen auf der linken Seite das Operators zuweisen. Als unäre Operatoren können die Operatoren ++(Inkrement) und – (Dekrement) in der Sprache verwendet werden, um Variablen einfach als Zähler benutzen zu können.

Datenmodellanbindung

Der Zugriff auf die Variablen des Datenmodells im Quelltext der Skriptsprache ist über den Zusatz "@" möglich. Soll beispielsweise auf die Fuzzy-Variable "Gamma" des Datenmodells zugegriffen werden, muss im Quelltext der Name "@Gamma" für die Variable angegeben werden. Der Zugriff auf die temporären Variablen erfolgt ebenso. Probleme können hierbei nicht auftreten, da alle Variablen des Datenmodells eindeutige Namen besitzen. Der Zusatz "@" ist notwendig, um die Variablen des FuzzyIDE Datenmodells von den Variablen im IDEInterpreter-Plugin zu unterscheiden. Abgesehen von dem "@" Zusatz, gleicht das Arbeiten mit den Datenmodellvariablen dem Arbeiten mit Variablen des IDEInterpreters. Zu beachten ist allerdings, dass es zu einer Exception kommt, sollte einer Fuzzy-Variablen ein Wert zugewiesen werden, der außerhalb ihrer Basisskala (UoD) liegt. Geschieht dies, gibt der IDEInterpreter eine SimulationPluginException an den Simulator zurück. Falls diese Exception nicht geblockt wurde (siehe Abs. 12.3), wird die Simulation beendet.

13.4. Compiler

Der Quellcode muss zur Anwendung in eine ausführbare Form gebracht werden. Hierzu wird ein Compiler benötigt. Er liest den Quellcode und übersetzt ihn in eine ausführbare Form [Beca]. Der IDEInterpreter-Compiler benötigt folgende Teile:

13.4.1. Lexikalischer Scanner

Die Aufgabe der lexikalischen Analyse ist es, den Quelltext einzulesen und in eine weiterverarbeitbare Form zu bringen. Hierzu wird der Quelltext in Morpheme (auch Atome, Lexeme) zerlegt. Dabei werden nicht benötigte Zeichen wie Leerzeichen, Zeilenumbrüche oder Kommentare überlesen. Was ein Morphem ist, hängt von der Sprache ab, die der Compiler umsetzen soll. Morpheme können aus einzelnen Zeichen oder Zeichenketten bestehen. Einige Morpheme, wie beispielsweise Schüsselwörter oder Zahlen, werden während der lexikalischen Analyse in eine einfacher verarbeitbare Form gebracht. Schüsselwörter werden von Strings in Integerwerte konvertiert. Diese kann der Parser einfacher verarbeiten.

Möglichkeiten, einen Lexer in Java zu implementieren:

- Es ist möglich, einen Lexer in Java mit der Klasse StringTokenizer aus dem Package java.util zu realisieren. Die Klasse bietet Funktionen, einen String in Morpheme zu zerlegen. Für eine lexikalische Analyse einer Programmiersprache müssten die Funktionen der Klasse erweitert werden.

- Für Java existiert eine Reimplementierung des Tools *lex* [Her99]. Es möglich, mit Hilfe von "lex" einen Lexer als C Quelltext zu generieren. Die Reimplementierung für Java mit dem Namen JLex erzeugt den Java Quelltext einer Klasse zur lexikalischen Analyse [JLE].

Für die Erzeugung eines Lexers wurde das Tool JLex verwendet. JLex liest eine Eingabedatei mit der Erweiterung "lex" und setzt den Inhalt in eine Javaklasse um, die zur lexikalischen Analyse benutzt werden kann.

Die mit JLex erzeugte Klasse besitzt einige von JLex generierte Methoden. Die wichtigste Methode ist dabei *yylex()*. Sie veranlasst den Lexer, das nächste Morphem zu verarbeiten. Diese Methode wird durch den Parser aufgerufen, wenn dieser ein neues Morphem benötigt. Über den Rückgabewert der von JLex generierten Methode "value()" erhält der Parser das aktuelle Morphem in einer verarbeitbaren Form.

13.4.2. Parser

Der Parser ist im Compiler dafür verantwortlich, den Eingabetext auf die Grammatik der Programmiersprache zu prüfen und die Funktionen zur Codegenerierung aufzurufen. Der im Compiler benötigte Parser wurde mit dem Parsergenerator Jay [JAY] generiert. Dieses Tool ist eine Reimplementierung des Parsergenerators Yacc [Her99]. Mit Yacc können Parser in der Programmiersprache C ausgegeben werden. Im Gegensatz zu Yacc gibt Jay den generierten Parser als Java Klasse aus. Der Einsatz Jay und JLex empfiehlt sich, weil ein mit Jay generierter Parser einen mit JLex erzeugten Lexer über ein Interface aufrufen kann und keine Änderungen am generierten Parser notwendig sind.

Im Fall des IDEInterpreter Compilers wird in den Codegenerationsmethoden ein Objektbaum erzeugt, der während der Simulation ausgeführt wird. Ein Beispiel dazu befindet sich im Anhang A.3.

13.4.3. Spracherweiterungen

Neben den in Abschnitt 13.3 beschriebenen Fähigkeiten der Sprache, können in der Skriptsprache Funktionen aufgerufen und für Berechnungen genutzt werden. Dieses Verfahren ermöglicht es, die Fähigkeiten der Sprache zu erweitern, ohne am Compiler Änderungen vornehmen zu müssen.

Jede in der Skriptsprache genutzte Funktion muss in einer eigenen Klasse im Package "functions" des Plugins implementiert sein. Der Parser des Compilers erkennt eine Funktion an folgendem Aufbau im Quelltext: "Funktionsname(Parameterliste)".
Wurde eine Funktion im Quelltext erkannt, versucht der Compiler die entsprechende Klasse zu laden, in der die Funktion implementiert ist. Hierzu wird der Reflexion Mechanismus von Java genutzt [Ess02]. Wird zum Beispiel die Funktion *sin* im Quelltext erkannt, versucht der Compiler, die Javaklasse *sin* im Package "functions" zu laden. Anhand der Parameterliste der Funktion wird nach einem passenden Konstruktor der Klasse gesucht und

dieser aufgerufen. Wird die Klasse nicht gefunden oder ist kein entsprechender Konstruktor vorhanden, wird der Compilerlauf mit einer entsprechenden Fehlermeldung abgebrochen.

13.4.4. Funktionsarten

Es existieren zwei unterschiedliche Arten von Funktionen, die geladen werden können.

Expressionen

Die von der Klasse Expression(siehe Anh. A.4) abgeleiteten Funktionen haben einen Rückgabewert und können nur an solchen Stellen im Quelltext aufgerufen werden, an welchen auch eine Zahl stehen könnte.

Beispiel: d= **sin**(49);
In der Zuweisung wird die Funktion *sin* mit dem Parameter 49 aufgerufen.

Um eine eigene, von Expression abgeleitete Funktion zu implementieren, müssen folgende Methoden der Expression Klasse überschrieben werden:

1. Methode: **public double doubleValue()**
   Diese Methode gibt den Wert der Expression als "double" Wert zurück. Sie wird zum Beispiel genutzt, einer Variablen vom Typ "double" den Funktionswert zuzuweisen.

2. Methode: **public long longValue()**
   Diese Methode gibt den Wert der Expression als "long" Wert zurück. Sie wird beispielsweise genutzt, einer Variablen vom Typ "int" den Funktionswert zuzuweisen.

3. Methode: **public boolean compare()**
   Diese Methode gibt *true* zurück, wenn die Expression wahr ist. Ist die Expression falsch, ist der Rückgabewert *false*.Diese Funktion wird immer aufgerufen, wenn die Expression als Bedingung verwendet wird(z.B. in einer IF Anweisung). In Anlehnung an C ist eine Expression wahr, wenn sie ungleich 0 ist. Für abgeleitete Funktionen kann das Verhalten selbst festgelegt werden. So gibt beispielsweise die Funktion "isEOF(File)" true zurück, wenn das Ende der Datei ereicht ist.

Statements

Der andere Funktionstyp ist von der Klasse Statement abgeleitet. Diese Funktionen haben keinen Rückgabewert und können nicht als Bedingung verwendet werden.

Beispiel: print("Text");
In der Befehlszeile wird die von Statement abgeleitete Klasse "print" aufgerufen.

Die abstrakte Klasse Statement besitzt nur das Interface IExecute mit der Methode "public void execute()". Diese Methode muss in der abgeleiteten Klasse mit der gewünschten Funktionalität gefüllt werden. Die "execute" Methode wird während des Ausführens des Programms aufgerufen.

Für Statements und Expressionen gilt: für alle verwendeten Parametertypen, muss ein entsprechender Konstruktor existieren. Soll die Funktion "print" als Parameter einen String oder eine Doublevariable enthalten, müssen auch die entsprechenden Konstruktoren für die Klasse "print" vorhanden sein.

Der Konstruktor der Funktionsklasse wird der Kompilierung aufgerufen. Die oben beschriebenen Methoden(execute, doubleValue usw.) werden zur Laufzeit des Programms ausgeführt. Eine Beschreibung aller bestehenden Funktionen befindet sich im Anhang A.4.

13.5. Fehlerbehandlung

Das IDEInterpreter-Plugin hat wie alle Simulatorplugins die Möglichkeit, Exceptions vom Typ "SimualtorPluginException" an den Simulator zu geben, um einen Fehler anzuzeigen. Wie im Abschnitt 12.3.2 beschrieben, besitzt die Klasse "SimualtorPluginException" einen Parameter, der die Art des Fehlers angibt. Die Art des Fehlers, den das Plugin an den Simulator gibt, ist abhängig davon, an welcher Stelle der Fehler auftritt. Der wichtigste Fehlertyp ist der "Runtime Error". Er tritt auf, wenn in einer Berechnung des Interpreters eine Division durch 0 erfolgt.

Zudem kann der Nutzer selbst Exceptions an den Simulator geben. Hierfür existieren die Funktionen "throwError(String)" und "throwWarning(String)" (siehe Anhang A.4). Die Funktionen geben einen Fehler vom Typ "Runtime Error" bzw. "Runtime Warning" mit einem String an den Simulator. So kann der Nutzer selbst im Quelltext festlegen, welche Bedingungen für einen Abbruch der Simulation vorliegen müssen.

Darüber hinaus können in den Funktionen zur Spracherweiterung (siehe Abs. 13.4.3) weitere Fehler auftreten. Bei solchen Funktionen, die für die Arbeit mit Dateien notwendig sind, kann es zum Beispiel vorkommen, dass Fehler vom Typ "IO Error" an den Simulator gegeben werden. Eine genaue Auflistung der Fehlertypen, welche in den bestehenden Funktionen auftreten können, befindet sich im Anhang A.4.

13.6. Antfile

Das Antfile des IDEInterpreter-Plugins führt neben der Kompilierung des Plugins noch folgende zusätzliche Aufgaben aus:

1. Aufrufen des Lexergenerators JLex. Der Aufruf des Lexergenerators erzeugt die Klasse "Scanner". Die bestehende Klasse wird überschrieben.

2. Aufrufen des Parsergeneratorsgenerators Jay, der die Klasse "Compiler" neu erzeugt.

Wurden Änderungen an den Kassen "Scanner" und "Compiler" vorgenommen, gehen diese beim Abarbeiten des Antfiles verloren. Änderungen der Klasse "Scanner" müssen deshalb an der Datei ./JLex/scanner.lex vorgenommen werden. Es gilt es zu beachten, dass es sich hierbei nicht um eine Java Datei handelt, sondern um eine "lex" Datei zum Erzeugen einer Java Datei. Die Klasse "Compiler" wird von Jay überschrieben. Um hier Änderungen vorzunehmen, muss die Datei ./Jay/Compiler.jay angepasst werden. Es gilt auch hier zu beachten, dass es sich nicht um eine Java Datei handelt, sondern um eine "jay" Datei zum Erzeugen einer Javadatei.

JLex existiert als Java Programm und kann somit plattformunabhängig genutzt werden. Im Gegensatz dazu existiert Jay als C Programm und ist somit nicht plattformunabhängig. Im Package des IDEInterpreter-Plugins befindet sich eine kompilierte Version von Jay für Linux, welche vom Antfile genutzt wird. Daher kann das Antfile in seiner bestehenden

Version nur unter Linux verwendet werden. Für andere Plattformen muss Jay erneut übersetzt und das Antfile angepasst werden. Der Quellcode von Jay liegt im IDEInterpreter Package in der Datei ./Jay/Jay_SourceCode.rar.

14. Simulationaction-Plugin

14.1. Beschreibung

Die Aufgabe des Simulationaction-Plugins besteht darin, dem Nutzer eine Interaktion während der Simulation zu ermöglichen. Im Unterschied zum IDEInterpreter-Plugin kann der Nutzer jedoch festlegen, zu welchem Zeitpunkt der Code ausgeführt wird.

14.2. Anwendung

14.2.1. Aktionen anlegen

Das Simulationaction-Plugin implementiert die IPreprocessing und die ISimulationprocessing Schnittstelle. Die beiden Schnittstellen können im "Plugin Chooser" Dialog nur gemeinsam aktiviert oder deaktiviert werden (siehe Abb. 12.2.1).

Bevor das Plugin in der Simulation genutzt werden kann, müssen die Aktionen angelegt werden, die während der Simulation ausgeführt werden sollen. Dazu muss der "Simulation Actions" Dialog (siehe Abb. 14.1)im Menüpunkt "Simulator Plugins" aufgerufen werden. In diesem Dialog kann im Textfeld der Quellcode eingegeben werden, der während der Simulation aufgerufen werden kann. Über die Combobox links oben kann festgelegt werden, über welche Schnittstelle der eingegebene Quellcode aufgerufen wird. Zur Auswahl stehen die beiden Schnittstellen, die das Plugin implementiert (IPreprocessing, ISimulationprocessing). Der Aktion muss im Textfeld oben links ein Name zugewiesen werden. Wurden diese Eingaben getätigt, kann die Aktion mit dem "Add" Button angelegt werden. Diese erscheint dann in der Listbox links unten. Mit Hilfe Des "Remove" Buttons können Aktionen wieder gelöscht werden.

Der Dialog bietet außerdem Funktionen zum Testen des eingegebenen Quellcodes(Button "Test Code"), Speichern von Aktionen (Button "Save Actions") in einer Datei und Laden von Aktionen aus einer Datei(Button "Load Actions").

14.2.2. Aktionen ausführen

Soll das Plugin während der Simulation benutzt werden, muss es im "Plugin Chooser" Dialog aktiviert sein. Zusätzlich muss im Plugin mindestens eine Aktion angelegt sein(siehe Abs. 14.2.1). Ist dies der Fall, wird vor Beginn der Simulation durch den Simulator die Methode "initPlugin()" des SimulationAction-Plugins ausgeführt. In dieser Methode werden alle Aktionen kompiliert und ein Dialog erzeugt. Dieser Dialog enthält für jede angelegte Aktion einen Button mit dem Namen der Aktion (siehe Abb. 14.2).

Wird während der Simulation einer der Buttons betätigt, merkt sich das Plugin die entsprechende Aktion zur Ausführung vor. Der kompilierte Code wird dann beim nächsten Aufruf der zugewiesenen Schnittstelle(Preprocessing oder ISimulationprocessing) durch

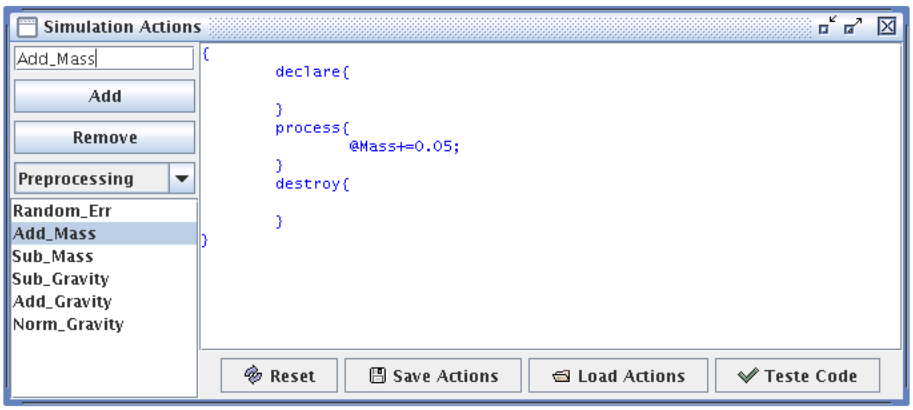


Abbildung 14.1.: Screenshoot Simulation Actions

Abbildung 14.2.: Screenshoot Simulationaction

den Simulator ausgeführt. Sollte ein Button mehrmals betätigt werden bevor die entsprechende Schnittstelle durch den Simulator aufgerufen wurde, wird trotzdem die Aktion nur einmal ausgeführt. Die Fehler, die im Code der Aktionen auftreten können, sind dieselben wie im IDEInterpreter-Plugin(siehe Abs. 13.5).

14.3. Implementierung

Wie oben beschrieben, benutzt das Plugin die Klassen des IDEInterpreter-Plugins. Um die Klassen nicht doppelt im Projekt zu halten, werden beim Ausführen des Ant-Files die Klassen des IDEInterpreter-Plugins in das Jar-File des Simulationaction-Plugin kopiert. Um das Plugin erfolgreich zu übersetzen, werden aus diesem Grund die Quellen des IDEInterpreter-Plugins benötigt.

Genau wie das IDEInterpreter Plugin implementiert auch das Simulatoraction-Plugin das IPluginLoad Interface und kann daher seine Daten im Savefile der FuzzyIDE speichern. Beim Laden eines FuzzyIDE Projektes werden die Aktionen wieder hergestellt.

15. Fuzzyplot-Plugin

15.1. Beschreibung

Wie in der Anforderungsanalyse beschrieben, wurde zur Ausgabe der Simulationsergebnisse ein Plugin implementiert, welches die Werte zweier Variablen in einem Koordinatensystem ausgibt. Das Plugin wurde hauptsächlich zu Testzwecken implementiert, da während des Redesigns bereits an einem anderen Plugin gearbeitet wurde, welches dieselben Aufgaben erfüllt und darüber hinaus mehr Funktionen bietet (siehe Anh. C).

15.2. Anwendung

Das Fuzzyplot-Plugin besitzt wie alle Plugins, die der Simulator benutzt, die ISimulation-Plugin Schnittstelle und darüber hinaus die während der Simulation genutzte ISimulationprocessing Schnittstelle. Um das Plugin zu nutzen, muss vor Beginn der Simulation im "Plugin Chooser" Dialog die ISimulationprocessing Schnittstelle aktiviert werden.

15.2.1. Plots anlegen

Zur Darstellung von Werten müssen vor Beginn der Simulation ein oder mehrere Plots angelegt werden. Um Plots anzulegen, wird der Dialog "Plot Generator" (siehe Abb. 15.1) benötigt, der im Menü "Simulator Plugins" zur Verfügung steht, sobald das Fuzzyplot-Plugin geladen wurde. Über die Comboboxen im Dialog(siehe Abb. 15.1) kann festgelegt werden, welche Variablen in einem Plot verwendet werden. Zur Verfügung stehen alle Variablen des Datenmodells (Fuzzy- und Tempvariablen) sowie eine Iterationsvariable, die als Wert die Anzahl der berechneten Simulationsschritte enthält.
Mit den Textboxen neben den Comboboxen kann der Bereich festgelegt werden, der im Plot dargestellt werden soll. Handelt es sich bei der ausgewählten Variable um eine Fuzzy-Variable, kann über die Checkbox "use UoD" angegeben werden, dass der Plot als Ausgabebereich die Basisskala der Fuzzy-Variablen verwenden soll.
Mit Hilfe des "Add" Buttons wird der Plot angelegt und erscheint in der Listbox. Soll ein Dialog während der Simulation nicht mehr genutzt werden, kann er über den "Remove" Button entfernt werden. Zum Entfernen aller vorhanden Plots dient der Button "Clear All".

15.2.2. Anzeige während der Simulation

Für den Fall, dass das Plugin im "Plugin Chooser" Dialog aktiviert und vor Beginn der Simulation mindestens ein Plot angelegt wurde, erscheint nach dem Start der Simulation für jeden angelegten Plot ein Dialog auf der Arbeitsfläche(Desktop). Die Plots(Fenster) werden in der Methode "initPlugin()" des Fuzzyplot-Plugins erstellt, die vom Simulator vor Beginn der Simulation aufgerufen wird.

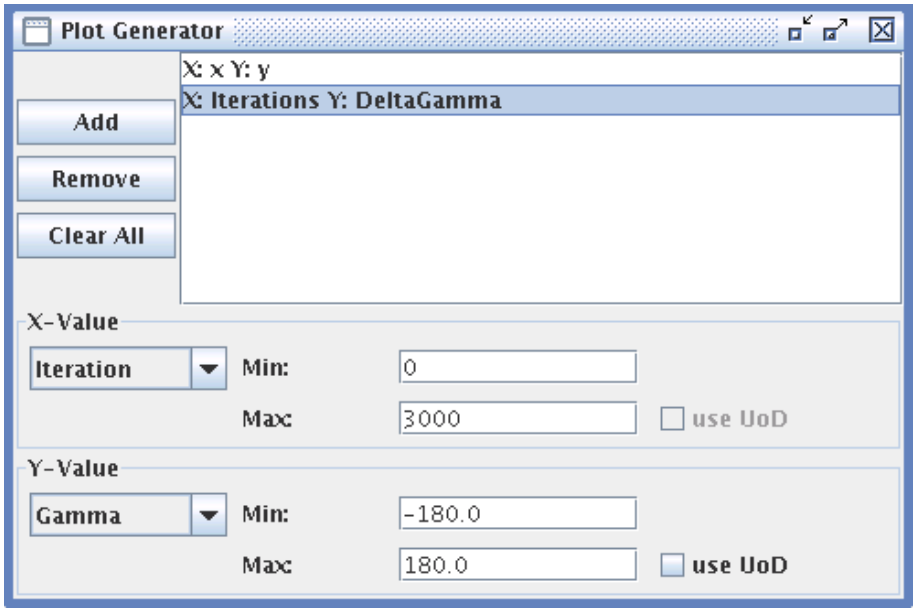


Abbildung 15.1.: Screenshot Plot Generator

Die Methode der ISimulationprocessing Schnittstelle wird aufgerufen, nachdem der Simulator die Berechnungen des Fuzzy-Systems durchgeführt hat. In der Methode liest das Plugin alle Werte, die in Plots verwendet werden und stellt diese in den Plot-Dialogen auf dem Desktop dar. Beim ersten Simulationsschritt wird allerdings noch nichts in den Plots angezeigt, da das Plugin die Simulationsergebnisse als miteinander verbundene Linien darstellt.

Die erste Linie wird beim zweiten Durchlauf des Simulators gezeichnet, da erst dann der Endpunkt der Linie bekannt ist (siehe Abb. 15.2).

Sollte während der Simulation der Wert einer Variablen außerhalb des Bereiches liegen, der im "Plot Generator" Dialog festgelegt wurde, wirft das Plugin eine "SimulationPluginException" vom Typ Visualisierungsfehler. Wurde im Dialog "Simulation Errorhandling"(siehe Abs. 12.3.2) festgelegt, dass Visualisierungsfehler behandelt werden, wird die Simulation beendet.

Wird die Simulation beendet, ruft der Simulator die "destroyPlugin()" Methode des Fuzzyplot-Plugins auf. In dieser Methode werden alle vorhandenen Plots vom Desktop entfernt.

15.3. Probleme

Wie in Abschnitt 15.1 beschrieben, wurde das Plugin geschrieben, um die Ergebnisse des Simulators zu testen. Aus diesem Grunde wurden nur die hierfür benötigten Funktionen umgesetzt. Nachteile des Plugins sind: es erfolgt keine Beschriftung in den Ausgabeplots, es können nicht mehr als zwei Werte in einem Plot dargestellt werden und die Ausgabe der Werte erfolgt nur im angegebenen Bereich.

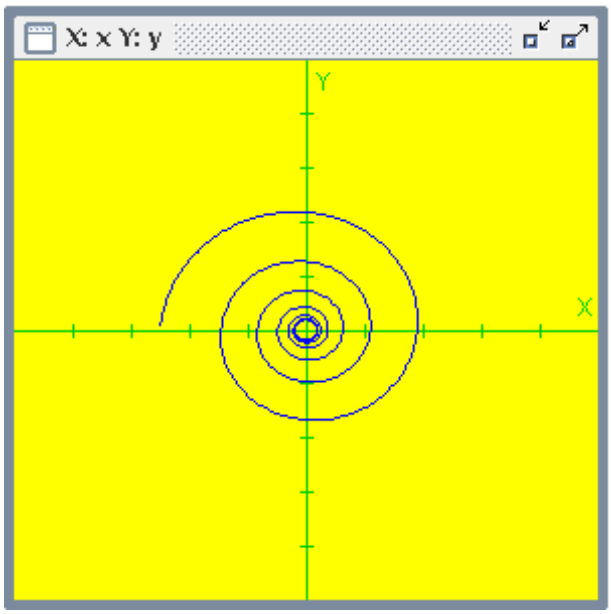


Abbildung 15.2.: Screenshot Fuzzyplot

Da das Plugin sehr einfach gehalten wurde, benötigt es sehr wenige Ressourcen und die Simulation läuft auch noch mit einer Vielzahl von Plots flüssig.

15.4. Speicherung

Um das Arbeiten mit der FuzzyIDE einfach zu gestallten, soll beim Laden eines FuzzyIDE Savefiles der gesamte Workspace wieder hergestellt werden. So müssen auch die Informationen über die angelegten Plots des Fuzzyplot-Plugins gespeichert werden. Damit das Plugin seine Daten im FuzzyIDE Savefile sichern kann, implementiert es das Interface "IPluginLoad". Das Interface besitzt die zwei Methoden "load" und "save". Diese werden aufgerufen, wenn ein FuzzyIDE Savefile geladen bzw. gespeichert wird. Über diese Methoden exportiert und importiert das Plugin die Informationen über die bestehenden Plots, so dass beim Laden eines Workspace alle Plots wieder hergestellt werden, die beim Speichern vorhanden waren.

16. Codegeneration

Da das Ziel der Entwicklung eines Fuzzy-Systems ein funktionstüchtiges Programm ist, muss das entwickelte Fuzzy-System in Quellcode einer Programmiersprache umgesetzt werden können. Die Codegeneration ist in FuzzyIDE als Plugin realisiert und unterstützt zurzeit Quellcodeausgaben in Java und C. Bei der Ausgabe als Java-Quellcode wird eine Klasse erzeugt, die alle nötigen Methoden zur Nutzung des Fuzzy-Systems bietet. Zur Anwendung der erzeugten Klasse wird die NRC-Klassenbibliothek [NRC] benötigt. Bei dem erzeugten Javacode besteht der Vorteil, dass die Ergebnisse des erzeugten Fuzzy-Systems mit den Ergebnissen aus der FuzzyIDE Entwicklungsumgebung übereinstimmen, da die FuzzyIDE ebenfalls die NRC-Bibliothek für die Berechnung verwendet.
Für den erzeugten C-Quellcode wird die Cubicalc-Runtime Bibliothek benötigt, die alle Funktionen bereitstellt, um das Fuzzy-System anzuwenden. Der Vorteil der C-Codegeneration ist die Performance von C und die Möglichkeit, den kompilierten Quellcode auf Mikrocontrollern zu nutzen. Der Nachteil ist der, dass die Ergebnisse der Berechnungen nicht vollständig mit den Ergebnissen der FuzzyIDE übereinstimmen müssen. Die Cubicalc-Runtime Bibliothek [CuR92] unterstützt nur einen Teil der Funktionalität der FuzzyIDE Entwicklungsumgebung. Beim Erstellen eines Fuzzy-Systems, welches später in C-Code umgewandelt werden soll, muss darauf geachtet werden, nur solche Funktionen zu verwenden, die von CubiCalc [Cub93] unterstützt werden.

16.1. Prinzip

Das Prinzip der Codegeneration ist simpel. Als Vorlage wird eine Datei benutzt, die den Quelltext enthält, welcher in allen Fällen gleich ist. In diese Datei werden dann alle nötigen Daten des Fuzzy-Systems eingetragen. Die Vorlagen befinden sich im Package *de.htwdd.robotic.fuzzyide.plugins.codegeneration.templets*. Bei der Generierung des Quellcodes wird abwechselnd Text aus der Vorlage und vom System generierter Text in die Ausgabedatei geschrieben. Beim Kopieren des Quellcodes aus der Vorlage wird der Text so lange übernommen, bis die Kopierfunktion die Zeichen "%%" liest. Der nächste Teil wird dann vom Plugin erzeugt, indem es die Daten des aktuellen Fuzzy-System der FuzzyIDE in den benötigten Quellcode umsetzt und in die Ausgabedatei schreibt. Diese Methode bringt den Vorteil, dass an den Quellcodevorlagen problemlos Änderungen vorgenommen werden können ohne das Plugin umzuschreiben. So kann beispielsweise bei der Java Vorlage einfach eine neue Methode hinzugefügt werden, da bei dieser Vorlage nur der Klassenname und der Konstruktor durch das Plugin erzeugt werden.

16.2. Java-Codegeneration

Um ein erstelltes Fuzzy-System als Javacode auszugeben, muss der NRC-Codegenerations-Dialog aufgerufen werden. In diesen Dialog sind die Ausgabedatei und der Packagename

der Klasse einzugeben. Der Klassenname der zu generierenden Klasse ergibt sich aus dem Namen der Ausgabedatei. Der erzeugte Code übernimmt die aktuellen Einstellungen der Inferenz-Einheit (Fuzzyfizierungsmethode, Defuzzyfizierungsmethode usw.) sowie die Sicherheitsfaktoren der Regeln aus der FuzzyIDE. Diese Einstellungen lassen sich noch mit Hilfe der Methoden der erzeugten Klasse ändern.
Die erzeugte Klasse besitzt folgende Methoden:

Konstruktor
Er erzeugt das Fuzzy-System mit allen Variablen und Regeln.

public void setInput(double[] inputValues)throws FuzzyException
Die Methode setzt den Input des Fuzzy-Systems und wirft eine Exception, wenn die Länge des Vektors *inputValues*nicht mit der Anzahl der Inputvariablen übereinstimmt oder der Parameter gleich null ist.

public void setInput(double input,int idx)throws FuzzyException
Die Methode setzt den Wert einer Inputvariablen des Fuzzy-Systems. Sie wirft eine Exception, wenn der Parameter *idx* außerhalb des Inputvariablenarrays ist.

public double[] getOutput()
Die Methode gibt die Outputwerte des Fuzzy-Systems zurück, die im letzten Schritt berechnet wurden.

public double getIdxOutput(int idx)throws FuzzyException
Die Funktion gibt einen Outputwert des Fuzzy-Systems zurück, welcher im letzten Schritt berechnet wurde. Die Methode wirft eine Exception, sollte der Parameter *idx* außerhalb des Outputvariablenarrays liegen.

public boolean[] getActivationRules()
Die Methode gibt ein Array mit booleschen Werten zurück. Diese geben die Aktivierung der Regeln an. Die Länge des Arrays entspricht dabei der Anzahl der Regeln des Fuzzy-Systems.

public void setRulesActiv(boolean[] activ)throws FuzzyException
Mit Hilfe der Methode können Regeln aktiviert oder deaktiviert werden. Sie wirft eine Exception, wenn die Länge des Arrays *inputValues* nicht mit der Anzahl der Regeln übereinstimmt

public double[] getActivGradeRules()
Das Array, welches die Methode zurückgibt, enthält den Aktivierungsgrad der Regeln beim letzten Berechnungsschritt. Die Werte schwanken dabei zwischen 0 und 1.Die Länge des Arrays entspricht dabei der Anzahl der Regeln des Fuzzy-Systems.

public void setCertFactor(double crt,int ruleIdx)throws FuzzyException
Die Methode setzt den Sicherheitsfaktor einer Regel des Fuzzy-Systems. Sie wirft eine Exception, wenn der Parameter *ruleIdx* außerhalb des Regelarrays ist.

public synchronized void calculateStep()throws FuzzyException
Die Methode berechnet einen Schritt des Fuzzy-Systems. Bevor die Berechnung erfolgt, sollten die Inputwerte gesetzt werden. Sie wirft eine Exception, wenn während der Berechnungen ein Fehler auftritt.

public long getIterations()
Die Methode gibt die Anzahl der berechneten Schritte zurück.

public void resetIteration()
Die Methode setzt die Iterationsvariable auf 0 zurück.

public void setDefuzzyMethode(int method)throws FuzzyException
Die Methode legt den Defuzzyfizierungsoperator für die Funktion *calculateStep()* fest. Mögliche Einstellungen sind DEFUZZ_AVERAGE, DEFUZZ_MOMENT und DEFUZZ_MAXIMUM. Als Standart wird die Einstellung der FuzzyIDE Inferenz-Einheit verwendet, die beim Generieren der Klasse eingestellt war.

public void setAccumulationMethode(int method)throws FuzzyException
Die Funktion setzt den Accumulationsoperator, der in der Funktion *calculateStep()* verwendet wird. Mögliche Einstellungen sind ACCUMULATION_UNION und ACCUMULATION_SUM. Als Standart wird die Einstellung der FuzzyIDE Inferenz-Einheit verwendet, die beim Generieren der Klasse eingestellt war.

Da die Variablen und Regeln innerhalb der erzeugten Klasse keine Namen mehr besitzen, müssen sie auf andere Weise unterschieden werden. Zur Unterscheidung der Variablen und Regeln wird für jede Variable und jede Regel ein "public static final" deklarierter Member vom Typ Integer angelegt. Für die Inputvariablen setzt sich der Name des Members aus "INPUT_VAR_" und dem Namen der Inputvariablen zusammen. Der Wert des Members entspricht dem Index der Inputvariablen. So kann beispielsweise die Funktion *setInput(double input,int idx)* zum Setzen der Inputvariablen "GAMMA" (des FuzzyIDE Datenmodells) mit dem Parameter INPUT_VAR_GAMMA aufgerufen werden.
Gleiches gilt für die Outputvariablen, die den Präfix "OUTPUT_VAR_" besitzen und für die Regeln, die den Präfix "RULE_" haben. Wichtig ist es, bei diesem Vorgehen zu unterscheiden, welche Indizes eine Methode benötigt: die von Inputvariablen, Outputvariablen oder Regeln.

Wurde der Javacode erfolgreich generiert, lässt sich die erstellte Javadatei mit folgendem Befehl kompilieren:

javac "Name der erzeugten Datei" **-classpath** "Pfad der NRC-Bibliothek"

16.3. CubiCalc C-Codegeneration

Die Codegeneration für die Cubicalc-Runtime Bibliothek wurde über Reverse Engineering erzeugt. Dazu wurde der von CubiCalc erzeugte Quellcode als Vorlage verwendet. Somit ist es möglich, dass der vom Codegenerations-Plugin erzeugte Quellcode Mängel aufweist, die ursächlich im Reverse Engineering liegen. In mehreren Versuchen hat sich gezeigt, dass

der generierte Code des Codegeneration Plugins in bestehende Systeme eingesetzt werden kann. Allerdings weichen die Ergebnisse des mit dem Codegenerations-Plugin erzeugten Codes leicht von den Ergebnissen des mit CubiCalc erzeugten Codes ab.

16.3.1. Probleme

Die Codegeneration in CubiCalc besitzt eine Reihe von Einstellungen, die bei der Codegeneration der FuzzyIDE nicht vorhanden sind. So unterstützt der FuzzyIDE Codegenerator nur eine Einstellung der Inferenz-Einheit. Der bei anderen Einstellungen von CubiCalc erzeugte Quellcode ließ sich über Reverse Engineering noch nicht entschlüsseln. Es ist weiteres Problem, dass die FuzzyIDE Modifikatoren und Einstellungen unterstützt, die in der Cubicalc-Runtime Bibliothek nicht vorhanden sind. So muss ein Fuzzy-System vor der Generierung des Quellcodes auf Kompatibilität geprüft werden. Sollte das Fuzzy-System nicht kompatibel sein, wird eine Fehlermeldung ausgegeben und das Generieren des Codes abgebrochen. Auch bei den Modifikatoren, die in der FuzzyIDE und in CubiCalc vorhanden sind, ist nicht gewährleistet, dass sie auf dieselbe Art und Weise umgesetzt sind. Es kann also auch hierbei zu Abweichungen in den Ergebnissen kommen.

Während des Reverse Engineerings konnten einige Parameter nicht eindeutig bestimmt werden. Zum Beispiel solche Parameter, die ihren Wert nie änderten. Für diese wurden die Werte fest übernommen. Bei anderen Parametern ist nicht sicher, dass die Berechnung in allen möglichen Fällen stimmt.

16.3.2. Kompatibilität

Wie oben beschrieben, muss ein Fuzzy-System kompatibel zur Cubicalc-Runtime Bibliothek sein, um es als Quellcode ausgeben zu können. Voraussetzungen für die Kompatibilität:

- **Modifikatoren:** Es dürfen in den Regeln des Fuzzy-Systems, welches als Quellode ausgegeben werden soll, nur folgende Modifikatoren vorkommen: *somewhat, very, not*. Außerdem ist die Nutzung der Modifikatoren weiter beschränkt. Es werden keine linguistischen Expressionen unterstützt, die mehrere Modifikatoren enthalten. Zulässig sind nur folgende Expressionen: *somewhat, very, not, not somewhat, not very*.

- **FuzzySets:**Es werden alle Fuzzysettypen außer dem Singleton-Fuzzyset unterstützt. Das Singleton Fuzzyset wird nicht unterstützt, da auch CubiCalc keine Singleton-Fuzzysets kennt. Es wäre möglich, Fuzzysets des Singleton-Fuzzysettyps als Triangel-Fuzzyset mit einer minimalen Fläche umzusetzen. Auf Grund der willkürlich festgelegten Fläche würden sich die Ausgaben des generierten Codes jedoch von denen der FuzzyIDE unterscheiden. So bleibt es dem Anwender überlassen, die Singleton-Fuzzysets seines Systems in entsprechende Triangel-Fuzzysets umzuwandeln.

- **Inferenz:** Als Inferenzstrategie muss SumProd und als Defuzzyfizierungsmethode das Flächenträgheitsmoment(COA) verwendet werden, da dies die einzige Strategie ist, für die das Plugin Code erzeugen kann.

16.3.3. Der generierte Code

Der mit CubiCalc generierte Quellcode besteht aus einem Headerfile und dem dazugehörigen C-File. Der Quellcode für die CubiCalc Runtime Bibliothek enthält keine Methoden im Gegensatz zum Quellcode, der für die NRC-Bibliothek erzeugt wird. Es werden nur eine Reihe von Strukturen und Makros erzeugt, die als Parameter für die Funktionen der Runtime Bibliothek benötigt werden. Da das gesamte Fuzzy-System in Strukturen umgesetzt wird, ist der Code sehr unübersichtlich. Aus diesem Grund sollen hier einige Details beschrieben werden, die es dem Anwender ermöglichen, den Code einzusetzen. Zusätzlich werden die Teile des Quellcodes beschrieben, die während des Reverse Engineering nicht völlig eindeutig bestimmt werden konnten.

16.3.4. Das Headerfile

Das Headerfile enthält eine Reihe von Makros, die das Arbeiten mit den Funktionen der CubiCalc Bibliothek erleichtern. Die Namen der Fuzzy-Variablen gehen bei der Generierung des Codes verloren. Die Zuordnung von In- und Outputs für die Variablen ist nur noch über Indizes möglich. Um dem Nutzer dennoch ein Arbeiten mit Variablennamen zu ermöglichen, wird für jede Variable ein Makro angelegt mit dem Namen der Variablen. Diese Makro enthält als Wert den Index der Variablen.

Variable: RangeError **Makro:** #define RangeError (0)

Zusätzlich werden Makros angelegt, die die Anzahl der Input- und Outputvariablen enthalten (INPUT_COUNT, OUTPUT_COUNT). Ein weiteres Makro WEIGHT_COUNT gibt die Anzahl der Sicherheitsfaktoren der Regeln an. Das Makro SCRATCH_SIZE hat währen des Reverse Engineerings große Schwierigkeiten bereitet. Der Wert des erzeugten Makros "SCRATCH_SIZE" gibt die Größe des temporären Speichers an, welcher von den CubiCalc Funktionen benötigt wird. Der vom Plugin berechnete Wert weicht bei komplexen Beispielen von den in CubiCalc berechneten Wert des Makros ab. Unbekannt ist, ob die Abweichung durch die Unterschiede in dem erzeugten Code entstehen oder durch eine falsche Berechnungsvorschrift für das Makro. Nach bisherigen Erkenntnissen ergibt sich der Wert des Makros aus der Anzahl der Fuzzysets(Input und Output) plus 1. Bei den getesteten Beispielen traten keine Probleme auf. Sollten es jedoch in den Funktionen der Runtime Bibliothek zu Speicherzugriffsfehlern kommen, kann dies ein Indiz dafür sein, dass das Makro einen falschen (zu kleinen) Wert besitzt.

Um die Strukturen des generierten C-Files in anderen C-Files zu nutzen, enthält das Headerfile die extern deklarierten Strukturen, welche zum Betreiben des Fuzzy-Systems benötigt werden.

16.3.5. Das C-File

Das C-File enthält die Strukturen, welche das Fuzzy-System definieren. Von den Strukturen sollen hier nur jene beschrieben werden, die für den Nutzer des Quellcodes von Belang sind.

Beim Erzeugen des Quellcodes werden alle Werte des Fuzzy-Systems auf den Bereich zwischen 0 und 1 normiert. Dies ist für das Arbeiten mit den Bibliotheksfunktionen der CubiCalc Runtime Bibliothek notwendig. Um die normierten Ergebnisse des Fuzzy-Systems wieder umzurechnen, werden zwei Strukturen angelegt.

```
FZ_IO_RANGE InputRanges[1] = {
        {140.00000,-70.00000}/* RangeError */
};

FZ_IO_RANGE OutputRanges[1] = {
        {120.00000,-60.00000} /* SteeringAngle */
};
```

Die erste Struktur enthält die Parameter zum Umrechnen der Inputvariablen vom normierten auf den ursprünglichen Bereich. Die zweite Struktur enthält die Parameter zum Umrechnen der Outputvariablen. Die Strukturen besitzen einen Eintrag für jede Input bzw. Outputvariable. Über das Makro mit den Variablennamen kann auf den entsprechenden Eintrag zu gegriffen werden (siehe Abs. 16.3.4). Jeder Eintrag besteht aus zwei Werten:

- Erstens der Größe der Basisskala der entsprechenden Variablen.
- Zweitens dem minimalen Wert der Basisskala.

Die wichtigste Struktur ist die "ROM_Parms" Struktur (siehe Anhang B). In ihr ist das gesamte Fuzzy-System enthalten. Sie enthält Referenzen auf alle Strukturen des C-Files mit Ausnahme der oben beschriebenen Strukturen zum Umrechnen der In- und Outputwerte. Außerdem enthält sie noch einige Parameter, die während des Reverse Engineerings nicht entschlüsselt werden konnten. Der Grund hierfür ist, dass diese Parameter unabhängig vom generierten Fuzzy-System immer denselben Wert besitzen.

16.3.6. Compilieren

Um den erzeugten Code in einer Applikation verwenden zu können, muss eine Main-Funktion geschrieben werden, welche die benötigten Funktionen der Runtime Bibliothek mit den erzeugten Strukturen aufruft. Um das fertige Programm zu kompilieren, werden folgende Headerfiles benötigt: fzoptr.h, fz.h, fzproto.h. Darüber hinaus muss die CubiCalc Runtime Bibliothek vorhanden sein.

Auf einem Linux System kann der Quellcode dann mit folgendem Befehl kompiliert werden:

gcc *.c -lfz

Voraussetzung ist dabei, dass sich die Headerfiles im Ordner */usr/include/* befinden und die Bibliothek *libfz.a* im Ordner */usr/lib* existiert. Ein Beispiel befindet sich im Anhang B und auf der CD im Verzeichnis: Codegen/CubiCalc/diff_cbc.

Teil V.

Ergebnisse

17. Fazit

Auf Grund der vorgenommenen Änderungen und Erweiterungen wurde das FuzzyIDE Projekt auf einen Stand gebracht, in dem es benutzt werden kann. Mit den vorhandenen angepassten Plugins bietet die FuzzyIDE ausreichende Funktionalität, um zu Lehrzwecken in Praktika eingesetzt werden zu können.

Um die FuzzyIDE für die Entwicklung von Fuzzy-Systemen einzusetzen, sind noch einige Funktionen zu implementieren. Einige der benötigten Funktionen sind im Kapitel 18 beschrieben. Bevor diese und andere Erweiterungen nicht vorgenommen wurden, kann die FuzzyIDE nicht mit kommerziellen Entwicklungsumgebungen wie CubiCalc konkurrieren.

17.1. Probleme

Die FuzzyIDE besitzt einige Funktionen, die während des Redesigns nicht getestet werden konnten. Der Grund hierfür ist, dass diese Methoden geschrieben wurden, um von Plugins benutzt zu werden, ohne dass entsprechende Plugins existieren. Für die gegenwärtige Version der FuzzyIDE stellt dies kein Problem dar. Die nicht getesteten Methoden müssen überprüft werden, wenn ein entsprechendes Plugin implementiert wird.

Mit der Einführung des Extension Point Mechanismus, als Ersatz für den bestehenden Pluginmanager, wurde die "Plugin Client" Funktion unbrauchbar(siehe Abs. 3.1). Die "Plugin Client" Funktion dient dem Herunterladen von Plugins von einem Server. Die Funktion lädt dazu die Jar-Datei des Plugins vom Server herunter. Der Extension Point Mechanismus benötigt außer der Jar-Datei noch eine XML-Datei (plugin.xml) zum Laden des Plugins. Die "Plugin Client" Funktion und der zugehörige Plugin-Server müssen deshalb noch angepasst werden.

17.2. Funktionsfähige Plugins

Mit der aktuellen Version der FuzzyIDE können folgende Plugins angewendet werden: Codegeneration-Plugin, Fuzzytree-Plugin ,Ruleditor2-Plugin, Console-Plugin, Simulationaction-Plugin, Variableneditor-Plugin, IDEInterpreter-Plugin, Fuzzyplot-Plugin, Fuzzyplot2-Plugin. Damit steht für jeden notwendigen Arbeitschritt zum Erstellen eines Fuzzy-Systems ein Plugin zur Verfügung:

- Variableneditor und Ruleeditor zum Anlegen und Bearbeiten von Variablen bzw. Regeln des Fuzzy-Systems.
- Die Plugins IDEInterpreter, Fuzzyplot, Fuzzyplot2 und Simulationaction zum Testen von Fuzzy-Systemen
- Codegeneration-Plugin zum Exportieren eines Fuzzy-Systems.

Beim Arbeiten mit der FuzzyIDE ist zu beachten, dass sich der Variableneditor noch in der Entwicklung befindet (siehe Anh. C) und es im Fuzzy-Plot2 Plugin noch nicht möglich, ist Plots im Savefile der FuzzyIDE zu speichern. Alle angelegten Plots im Fuzzyplot2-Plugin gehen beim Beenden der FuzzyIDE verloren.

17.3. Distribution

Als Ergebnis des Redesigns wurde eine Distribution erstellt. Diese enthält die FuzzyIDE mit allen funktionsfähigen Plugins sowie einige Beispiele zum Testen der Entwicklungsumgebung. Die Distribution existiert in zwei Versionen: als normales Java-Programm(CD Verzeichnis: ./FuzzyIDE_Projekt/Distribution/FuzzyIDE) und eine Version, in der die FuzzyIDE als Exe-Datei vorliegt(CD: /FuzzyIDE_Projekt/Distribution/FuzzyIDE_EXE). In der Exe-Version wurde das Jar-File der FuzzyIDE mit Hilfe des EXE-Wrappers JSmooth in eine Exe-Datei umgewandelt. Die Exe-Version ist nicht mehr plattformunabhängig. Sie kann nur mit Windowsbetriebssystemen eingesetzt werde. Im Gegensatz zur FuzzyIDE liegen die Plugins auch bei der Exe-Version als Jar-Datei vor.

17.3.1. Aufbau

Die beiden oben beschrieben Versionen besitzen in ihrem Verzeichnis folgende Unterverzeichnisse:

- **example:** Dieses Verzeichnis enthält einige Beispiele für die FuzzyIDE.
- **lib:** Dieser Ordner enthält die von der FuzzyIDE benötigten Bibliotheken inklusive der NRC-Bibliothek.
- **plugin:** Dieses Verzeichnis enthält die Plugins der FuzzyIDE. Der Ordner enthält für jedes Plugin ein Unterverzeichnis mit der Jar-Datei und der Datei "plugin.xml", die vom Extension Point Mechanismus zum Laden des Plugins benötigt wird.

17.3.2. Start

Um die FuzzyIDE zu starten, muss in der Exe-Version die "FuzzyIDE.exe" Datei ausgeführt werden. In der Jar-Version kann die FuzzyIDE mit der im Verzeichnis vorhanden Datei "fuzzyIDE.bat" gestartet werden (unter Linux/Unix mit ". fuzzyIDE.bat").

Nach dem Start der FuzzyIDE müssen die Plugins, die verwendet werden sollen (siehe FuzzyIDE Hilfe), über den Menüpunkt "Extension Points" geladen werden. Wird ein Savefile geladen, werden automatisch alle Plugins initialisiert, die bei der Speicherung der Datei vorhanden waren. Solle ein benötigtes Plugin nicht mehr vorhanden sein, wird über "System.err" eine Fehlermeldung ausgegeben.

17.3.3. Vorhandene Beispiele

In der Distribution existieren einige Beispielen, welche die Fähigkeiten der FuzzyIDE demonstrieren. Die Beispiele wurden aus CubiCalc in die FuzzyIDE übernommen. Die CubiCalc Varianten der Beispiele befinden sich unter auf der CD im Verzeichnis: ./CBC_examples.

DogCat

In diesem Beispiel wird ein Fuzzy-System zum Tracking eingesetzt. Es simuliert einen Hund, der eine Katze verfolgt.

Falter

Das Beispiel simuliert einen Nachtfalter, der um eine Lichtquelle kreist. Der Falter versucht dabei immer, den gleichen Winkel zum einfallenden Licht zuhalten.

Kreis

In dem Beispiel wird ein Fuzzy-System zum Zeichnen eines Kreises mit einem bestimmten Radius verwendet.

Kugel

Das Beispiel simuliert eine Eisenkugel, die im Magnetfeld eines Elektromagneten schwebt. Das Fuzzy-System wird dazu benutzt, die Magnetfeldstärke so zu regulieren, dass die Kugel in der Schwebe gehalten wird.

Truck

Das Truck Beispiel simuliert einen rückwärts einparkenden LKW. Das Fuzzy-System wird zum Steuern des LKWs verwendet.

Für jedes Beispiel existiert ein entsprechendes Unterverzeichnis im Ordner "example". Jedes Unterverzeichnis enthält folgende Einträge:

- **FuzzySystem.fzy:** Diese Datei enthält das gesamte Beispiel. Die Datei kann von der FuzzyIDE geladen werden.
- **IDEInterpreter:** Dieses Verzeichnis enthält den Quellcode des Beispiels für das IDEInterpreter-Plugin. Der Quellcode muss jedoch im Normalfall nicht geladen werden, da er bereits in der Datei "FuzzySystem.fzy" enthalten ist. Der Quellcode kann nur mit Hilfe des IDEInterpreter-Plugins geladen werden (siehe Abs. 13.2).
- **SimulatorAction:** Das Verzeichnis enthält die "Aktionen" für das Simulatoraction-Plugin. Das Laden der Datei ist im Normalfall nicht notwendig, da die Informationen auch in der Datei "FuzzySystem.fzy" enthalten sind. Die Datei kann nur vom Simulatoraction-Plugin geladen werden(siehe Abs. 14.2).
- **Codegeneration:** Der Ordner enthält das Beispiel als C und Java Quellcode. Der Quellcode wurde mit Hilfe des Codegeneration-Plugins erzeugt.

18. Erweiterungen

Da die Entwicklungen am FuzzyIDE Projekt mit der Abgabe der Diplomarbeit nicht eingestellt werden, sind hier einige Erweiterungen aufgeführt, welche im weiteren Verlauf der Entwicklung umgesetzt werden sollten. Grundlage für die hier aufgeführten Erweiterungen sind die während des Redesigns gewonnenen Erkenntnisse.

18.1. FuzzyIDE

18.1.1. Multiple Regelbasen

Die Fuzzy-Entwicklungsumgebung mbFuzzIT bietet die Möglichkeit, Fuzzy-Systeme zu erstellen, die mehrere Regelbasen benutzen. In dem erstellten Fuzzy-System können die Ergebnisse einer Regelbasis als Inputwerte für nachfolgende Regelbasen verwendet werden. Auf diese Weise lassen sich komplizierte Sachverhalte einfacher in einem Fuzzy-System modellieren. Die FuzzyIDE verfügt nicht über diese Möglichkeiten. Hier kann lediglich eine Regelbasis pro Fuzzy-System erstellt und genutzt werden. Um die FuzzyIDE so zu erweitern, dass sie ebenfalls mehrere Regelbasen unterstützt, sind weitreichende Änderungen am Datenmodell und am Simulator notwendig. Diese Änderungen hätten ebenfalls Rückwirkungen auf viele der Plugins. Der Aufwand für die Umsetzung multipler Regelbasen ist daher hoch, allerdings würde die Funktionalität der FuzzyIDE damit auch wesentlich erhöht.

18.1.2. Extension Point Mechanismus

Der von FuzzyIDE Teammitgliedern entwickelte Extension Point Mechanismus wird in der aktuellen Version nicht mit all seinen Fähigkeiten genutzt. Er sollte in zukünftigen Versionen weiter ausgebaut und eingesetzt werden. So ist zur Zeit bereits implementiert, dass beim Laden eines Savefiles alle Plugins mit Hilfe des Extension Point Mechanismus geladen werden, die beim Speichern des Savefiles vorhanden waren. Die nicht benötigten Plugins werden entfernt. Noch nicht implementiert ist jedoch das manuelle Entfernen der Plugins aus der FuzzyIDE. Darüber hinaus könnte der Mechanismus auch dafür verwendet werden, Teile der FuzzyIDE austauschbar zu gestalten. So wäre es beispielsweise möglich, über den Extension Point Mechanismus die Inferenz-Einheit des Simulators anzubinden, damit diese ausgetauscht werden kann.

18.1.3. Savefile

Im der aktuellen Version werden die Daten der FuzzyIDE(Fuzzy-System und die Informationen aus den Plugins) als serialisierte Datenobjekte gespeichert. Bei Änderungen an den Klassen, die serialisiert werden, kann bereits das Umbenennen eines Member-Objekts dazu

führen, dass alle gespeicherten Savefiles nicht mehr korrekt geladen werden können. Deshalb ist es sinnvoll, einen anderen Weg für die Speicherung der Daten zu wählen. Optimal wäre eine Speicherung in XML-Dateien.

18.2. Plugins

18.2.1. Im- und Exportplugins

In vielen Fällen ist die FuzzyIDE auf externe Daten angewiesen. Damit diese komfortabel im- und exportiert werden können, sind zusätzliche Plugins nötig. Zwar bietet das IDEInterpreter-Plugin die Möglichkeit, in Dateien zu schreiben und aus ihnen zu lesen, die Funktionen dafür sind jedoch nicht sehr komfortabel. Zudem liegen die Inputwerte möglicherweise in einer Form vor, die nicht verarbeitet werden kann(z.B. XML-Datei). Um diese Schwächen zu beheben, sind weitere Im- und Exportplugins zu implementieren(z.B. Plugins zum Lesen und Schreiben verschiedener Dateitypen).

Eine weitere Möglichkeit Daten zu im- und exportieren, ist eine DDE-Anbindung über ein Plugin. Dieses Plugin ermöglicht, die FuzzyIDE mit einem DDE-Server zu verbinden oder selbst einen DDE-Server zu betreiben.

Andere Entwicklungsumgebungen wie CubiCalc bieten diese Funktion zum Austausch von Werten mit anderen Anwendungen. Über ein DDE-Plugin könnte die FuzzyIDE Daten aus anderen Windowsanwendungen im- und exportieren. Da viele Windows Programme eine DDE-Serveranbindung besitzen, erschließen sich viele neue Möglichkeiten für die Anwendung der FuzzyIDE(z.B. der Import von Werten aus Excel).

18.2.2. Simulatorplugins

Die FuzzyIDE bietet bereits die Schnittstellen an, um während der Simulation das Fuzzy-System anzupassen (zum Aktivieren und Deaktivieren von Regeln, zum Setzen von Sicherheitsfaktoren usw.). Es gibt jedoch es noch keine Plugins, die diese Schnittstellen benutzen und dem Nutzer zugänglich machen. Sie sind noch zu entwickeln.

18.2.3. Analyseplugins

Es werden Plugins zur Analyse des Fuzzy-Systems benötigt. In CubiCalc gibt es die Möglichkeit, die Entscheidungsoberfläche als 3D-Diagramm auszugeben. Dabei werden zwei Inputvariablen für die X- und Y-Achse gewählt und für die Z-Achse des Diagramms eine Outputvariable. Im Diagramm ist dann zu erkennen, wie die Inputwerte des Systems umgesetzt werden. So kann erkannt werden, ob für alle Inputwerte Regeln vorhanden sind. Ein Plugin mit ähnlichen Fähigkeiten sollte auch für die FuzzyIDE realisiert werden, da es dem Nutzer die Entwicklung eines Fuzzy-Systems erleichtert.

Teil VI.

Anhang

A. IDEInterpreterplugin

A.1. BNF der Interpretersprache

Nicht Terminale Sysmbole:
FAKTOR,TERM. EXPRESSION, CONDITION, CONDITION_LIST, PART, PARAMETER_LIST, STATEMENT, BLOCK_STATEMENT, STATEMENT_LIST, INT_IDENT_LIST, DOUBLE_IDENT_LIST, FILE_IDENT_LIST, DECLAIR_LIST, DECLARE_SECTION, GLOBAL_INT_IDENT_LIST, GLOBAL_DOUBLE_IDENT_LIST, GLOBAL_FILE_IDENT_LIST, GLOBAL_LIST,GLOBAL_SECTION, PROCESS_SECTION, DESTROY_SECTION, PROGRAM, BLOCK

Terimale Symbole:
ident, fuzzyident, intvalue, floatvalue, fuction, k_if, k_return, k_do, k_while, k_int, k_double, global, process, destroy, stringvalue, declare

BNF:

<PROGRAM>::=
'{' <GLOBAL_SECTION> <DECLARE_SECTION>
<PROCESS_SECTION> <DESTROY_SECTION> '}'

<GLOBAL_SECTION>::=
|| global '{' '}'
|| global '{' k_var ':' <GLOBAL_LIST> '}'

<GLOBAL_LIST>::=
||k_int <GLOBAL_INT_IDENT_LIST> <GLOBAL_LIST>
||k_double <GLOBAL_DOUBLE_IDENT_LIST> <GLOBAL_LIST>
||k_file <GLOBAL_FILE_IDENT_LIST> <GLOBAL_LIST>

<GLOBAL_INT_IDENT_LIST>::=
ident ';'
||ident ',' <GLOBAL_INT_IDENT_LIST>
||ident '=' <EXPRESSION> ';'
||ident '=' <EXPRESSION> ',' <GLOBAL_INT_IDENT_LIST>

<GLOBAL_DOUBLE_IDENT_LIST>::=
ident ';'
||ident ','<GLOBAL_DOUBLE_IDENT_LIST>
||ident '=' <EXPRESSION> ';'
||ident '=' <EXPRESSION> ',' <GLOBAL_DOUBLE_IDENT_LIST>

<GLOBAL_FILE_IDENT_LIST>::=
ident ';'
||ident ',' <GLOBAL_FILE_IDENT_LIST>

<DECLARE_SECTION>::=
‖declare ’’ ’’
‖declare ’’ k_var ’:’ <DECLAIR_LIST> ’’

<DECLAIR_LIST>::=
‖k_int <INT_IDENT_LIST> <DECLAIR_LIST>
‖k_double <DOUBLE_IDENT_LIST> <DECLAIR_LIST>
‖k_file <FILE_IDENT_LIST> <DECLAIR_LIST>

<INT_IDENT_LIST>::=
ident ’;’
‖ident ’,’ <INT_IDENT_LIST>
‖ident ’=’ <EXPRESSION> ’;’
‖ident ’=’ <EXPRESSION> ’,’ <INT_IDENT_LIST>

<DOUBLE_IDENT_LIST>::=
ident ’;’
‖ident ’,’ <DOUBLE_IDENT_LIST>
‖ident ’=’ <EXPRESSION> ’;’
‖ident ’=’ <EXPRESSION> ’,’ <DOUBLE_IDENT_LIST>

<FILE_IDENT_LIST>::=
ident ’;’
‖ident ’,’ <FILE_IDENT_LIST>

<PROCESS_SECTION>::=
‖process <BLOCK>

<DESTROY_SECTION>::=
‖destroy <BLOCK>

<BLOCK>::=
’{’ <STATEMENT_LIST> ’}’
‖’{’ ’}’

<STATEMENT_LIST>::=
’;’
‖<STATEMENT> ’;’
‖<STATEMENT_LIST> <STATEMENT> ’;’
‖<BLOCK_STATEMENT>
‖<STATEMENT_LIST> <BLOCK_STATEMENT>

<BLOCK_STATEMENT>::=
k_if ’(’ <CONDITION_LIST> ’)’ <PART> k_else <PART>
‖k_if ’(’ <CONDITION_LIST> ’)’ <PART>
‖k_while ’(’ <CONDITION_LIST> ’)’ <PART>
‖k_for ’(’ <STATEMENT> ’;’ <CONDITION_LIST> ’;’
<STATEMENT> ’)’ <PART>

<PART>::=
<STATEMENT> ’;’
‖ <BLOCK>

<STATEMENT>::=

ident '=' <EXPRESSION>
||ident '+"+'
||ident '-"-'
||ident '-"=' <EXPRESSION>
||ident '+"=' <EXPRESSION>
||ident '*"=' <EXPRESSION>
||ident '/"=' <EXPRESSION>
||ident '%"=' <EXPRESSION>
||k_do <PART> k_while '(' <CONDITION_LIST> ')'
||function <PARAMETER_LIST>
||fuzzyident '=' <EXPRESSION>
||fuzzyident '+"+'
||fuzzyident '-"-'
||fuzzyident '-"=' <EXPRESSION>
||fuzzyident '+"=' <EXPRESSION>
||fuzzyident '*"=' <EXPRESSION>
||fuzzyident '/"=' <EXPRESSION>
||fuzzyident '/"=' <EXPRESSION>
||k_return

<PARAMETER_LIST>::=

')'
|| <EXPRESSION> ')'
|| <EXPRESSION> ',' <PARAMETER_LIST>
|| stringvalue ')'
|| stringvalue ',' <PARAMETER_LIST>

<CONDITION_LIST>::=

'(' <CONDITION_LIST> ')'
||<CONDITION> '&"&' <CONDITION_LIST>
||<CONDITION> '||"||' <CONDITION_LIST>
||<CONDITION>

<CONDITION>::=

<EXPRESSION>
|| <EXPRESSION> '="=' <EXPRESSION>
|| <EXPRESSION> '<"=' <EXPRESSION>
|| <EXPRESSION> '>"=' <EXPRESSION>
|| <EXPRESSION> '>' <EXPRESSION>
|| <EXPRESSION> '<' <EXPRESSION>
|| <EXPRESSION> '!' '=' <EXPRESSION>
|| '!' <EXPRESSION>

<EXPRESSION>::=

<TERM>
|| <EXPRESSION> '+' <TERM>
|| <EXPRESSION> '-' <TERM>
|| '+' <TERM>
|| '-' <TERM>

<TERM>::=

<TERM> '*' <FAKTOR>
||<TERM> '/' <TERM>
||<TERM> '%' <TERM>
|| <FAKTOR>

<FAKTOR>::=

ident
||fuzzyident
||intvalue
||floatvalue
||'(' <EXPRESSION> ')'
||function <PARAMETER_LIST>

A.2. UML-Klassendiagramm der Sprache

Die Abbildung A.1 zeigt den Aufbau der Skriptsprache als UML-Diagramm. Zur Übersichtlichkeit wurden die Klassen des Datentyps "File" und die Funktionen, welche über den Reflexion Mechanismus [Ess02] geladen werden, nicht aufgeführt.

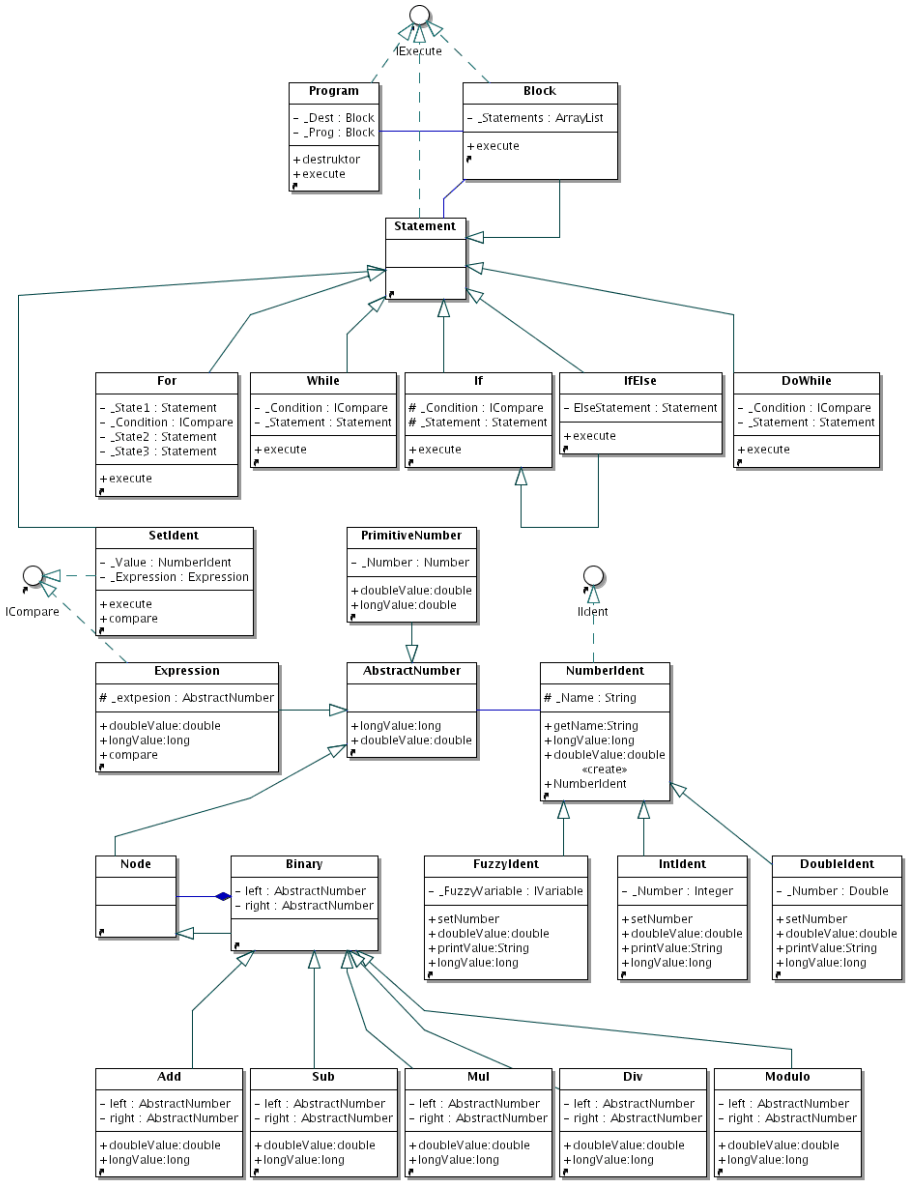


Abbildung A.1.: UML-Klassendiagramm IDEInterpreter

A.3. Beispiel erzeugter Code

Beispiel Quelltext:

```
{
        global{
        }

        declare{
                var:
                    int i , j;
                    double d= 0;
        }

        process{

                j= 2;
                for(i= 100 ; i > 0 ; i-- ){

                    if( ( i % 2 ) != 0 ){
                                d+= cos( 2 * j );
                                j++;
                    }
                else
                    print(d);
                }
                @delta= d;
        }

        destroy{
        }
}
```

Nach dem Aufruf des Compilers wird das Programm in einen Baum von Javaobjekten umgesetzt. Für den oben aufgeführten Quelltext ist der entsprechende Objektbaum in Abbildung A.2 zu sehen. Um die Abbildung übersichtlich zu halten, wurde der Baum nicht vollständig dargestellt. Die Objekte unterhalb der "SetIdent" Objekte wurden nicht aufgeführt. Der Objektbaum unterhalb eines "SetIdent" Objektes wurde für den Ausdruck "d+=cos(2 * j);" in der Abbildung A.3 dargestellt.

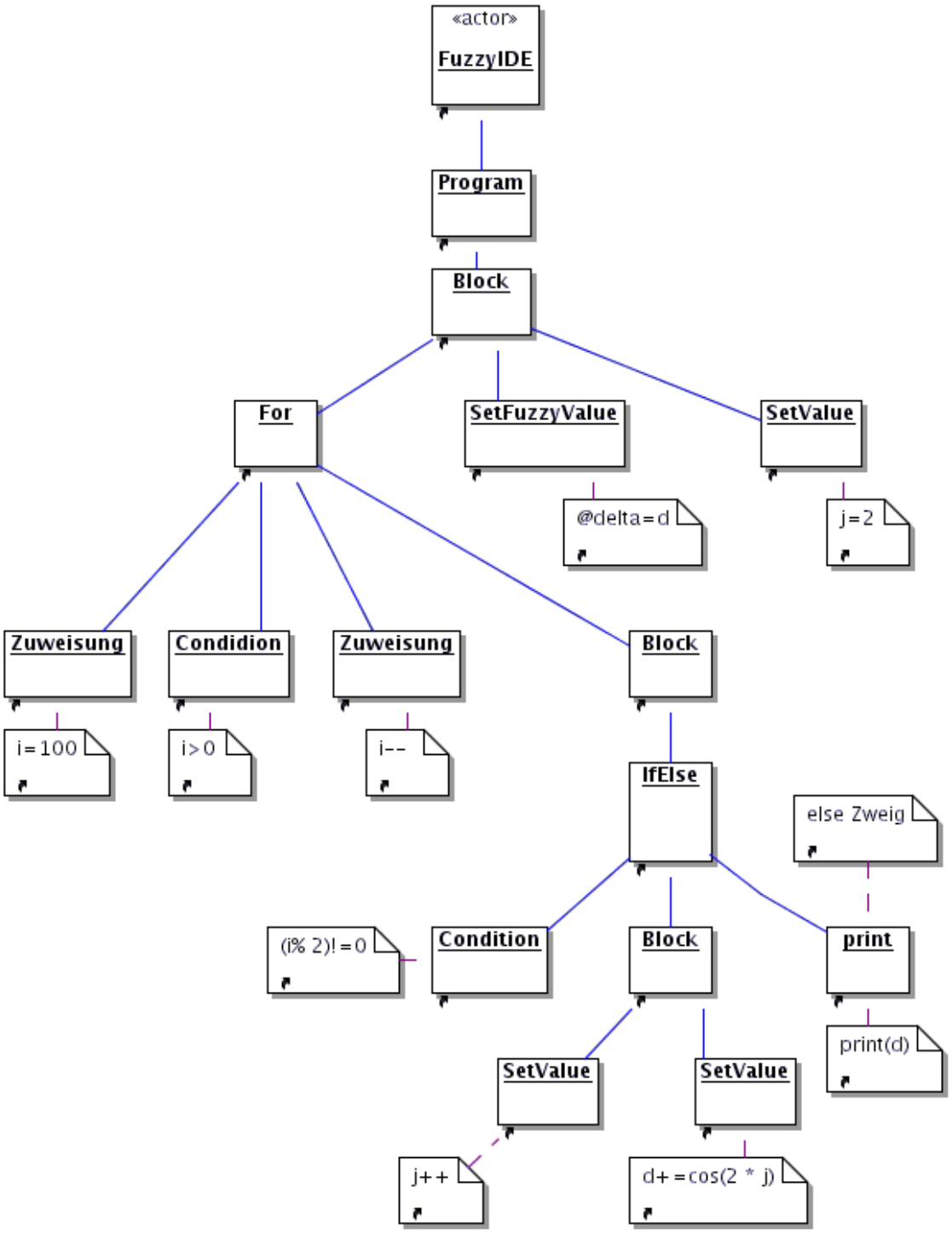


Abbildung A.2.: Objektbaum

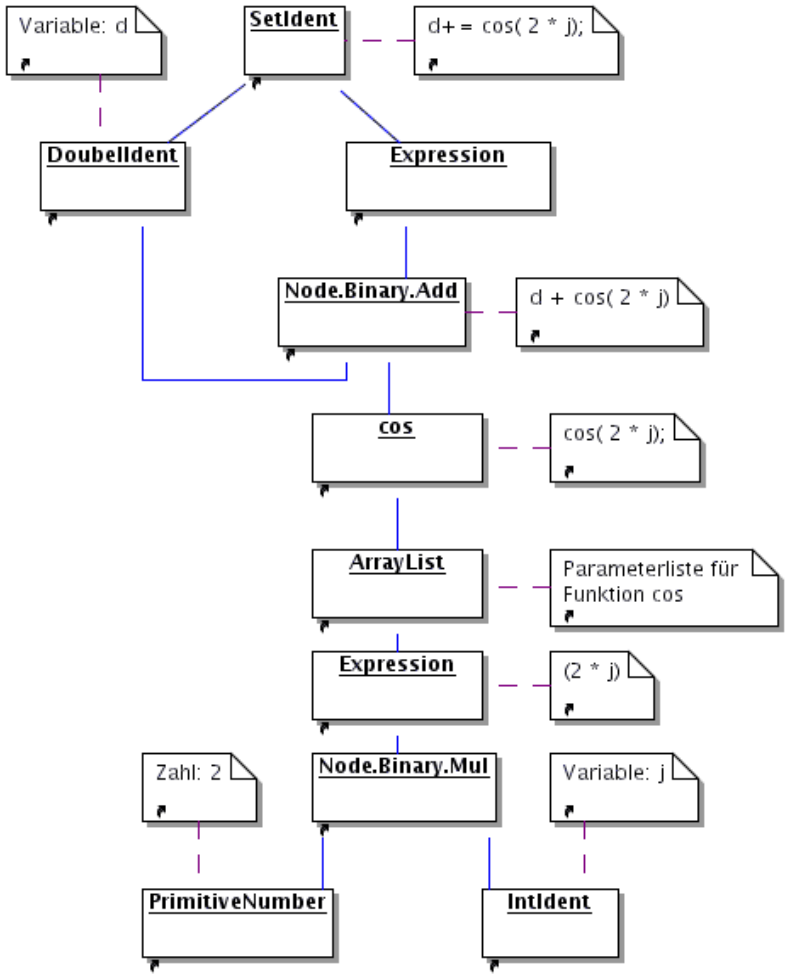


Abbildung A.3.: Objektbaum unterhalb von SetIdent

A.4. Bestehende Erweiterungen

Für die Skriptsprache des IDEInterpreters wurden während des Redesigns eine Reihe von Erweiterungen geschrieben, die in der Skriptsprache aufgerufen werden können. Jede dieser Erweiterungen besteht aus einer Klasse, welche mit Hilfe des IDEInterpreter Compilers geladen wird. Die Erweiterungen lassen sich in Expressions und Statements unterteilen. Expressions haben einen Rückgabewert. Sie können nur als Teil eines Statements aufgerufen werden. Im Gegensatz dazu haben Statements keinen Rückgabewert und können nicht Teil eines Ausdrucks sein. Die Klassen der Expressions und Statements befinden sich im Package "functions". Sie werden über den Java Reflexion Mechanismus zur Compilerlaufzeit geladen.

Statements

Procedure	Beschreibung	Parameter	Aufruf
Fehler Prozeduren			
throwError	Gibt eine Exception vom Typ "Runtime Error" an den Simulator zurück.	String	throwError("Text")
throwWarning	Gibt eine Exception vom Typ "Runtime Warning" an den Simulator zurück.	String	throwWarning("Text")
Datei Prozeduren			
open	Öffnet eine Datei. Wirft eine Exception vom Typ "IO Error", falls die Datei nicht geöffnet werden konnte.	File ,String	open(File,"Pfad")
close	Schließt eine Datei. Wirft eine Exception vom Typ "IO Error", falls das Schließen der Datei nicht erfolgreich war.	File	close(File)
seek	Setzt die Position in einer Datei. Wirft eine Exception vom Typ "IO Error", falls das Setzen der Position nicht erfolgreich war.	File, int	seek(File, int)
write	Schreibt in eine Datei. Wirft eine Exception vom Typ "IO Error", falls das Schreiben der Daten nicht erfolgreich war. Speicherung erfolgt als serialisiertes Objekt.	File, int File,double File,String	write(File,int) write(File,double) write(File,"Text")
writeInt	Schreibt in eine Datei. Wirft eine Exception vom Typ "IO Error", falls das Schreiben der Daten nicht erfolgreich ist. Speicherung erfolgt als "long" Wert.	File, int	writeInt(File,int)
print	Schreibt auf System.out .	String double int	print("Text") print(double) print(int)

Expressions

Funktion	Beschreibung	Zulässige Parameter	Aufruf
Mathematische Funktionen			
abs	absoluter Wert einer Zahl	double, int	abs(x)
acos	Arcuskosinus einer Zahl	double, int	acos(x)
asin	Arcussinus einer Zahl	double, int	asin(x)
atan	Arcustangens einer Zahl	double, int	atan(x)
cos	Kosinus einer Zahl	double, int	cos(x)
E	Eulerzahl	keine	E()
exp	e^x	double, int	exp(x)
log	ln_x	double, int	log(x)
max	Maximum zweier Werte	double, int	max(x , y)
min	Minimum zweier Werte	double, int	min(x , y)
PI	Konstante π	keine	PI()
pow	x^y	double,int	pow(x , y)
random	Zufallszahl zwischen 0 und 1	keine	random()
sin	sin(x)	double,int	sin(x)
sqrt	$\sqrt{x}$	double,int	sqrt(x)
tan	tan(x)	double,int	tan(x)
Winkel Funktionen			
angle180	normierter Winkel zwischen -180 und 180	double,int	angle180(x)
angle360	normierter Winkel zwischen 0 und 360	double,int	angle360(x)
Datei Funktionen			
pos	aktuelle Position in einer Datei (int)	File	pos(File)
isEOF	Test Dateiende ereicht	File	isEOF(File)
readLong	Liest "long" Wert aus Datei. Wirft eine Exception vom Typ "IO Error", falls das Setzen der Position nicht erfolgreich war.	File	readLong(File)

B. CubiCalc Codegeneration Beispiel

Headerfile

```
/*********************************************************************
Headerfile for circle.c
*********************************************************************/

/*********************************************************************
Inputvariables
*********************************************************************/
 #define RangeError (0)

#define INPUT_COUNT (1)

/*********************************************************************
Outputvariables
*********************************************************************/
 #define SteeringAngle (0)

#define OUTPUT_COUNT (1)

/*********************************************************************
Weigths
*********************************************************************/
 #define WEIGHT_COUNT (4)

/*********************************************************************
SCRATCH SIZE (size of the temp memory)
*********************************************************************/

#define SCRATCH_SIZE (7)


/*********************************************************************
Mainrulebank
*********************************************************************/
 #define COMMON_BANK (0)

/*********************************************************************
Variable declarations.
*********************************************************************/
```

```
 #ifndef FZ_NO_IO_RANGES
extern FZ_IO_RANGE InputRanges[];
extern FZ_IO_RANGE OutputRanges[];
#endif


/*****************************************************************************
Mainstruct contains all definitions for the fuzzysystem.
*****************************************************************************/
 extern struct fz_romparm ROM_Parms;
```

C-File

```
#include <fzoptr.h>
#include <fz.h>
# include <fzproto.h>

//%%
#ifndef FZ_NO_IO_RANGES
/*****************************************************************************
        Inputranges (uod)
*****************************************************************************/
 FZ_IO_RANGE InputRanges[1] = {

/***** RangeError ********************/
          {140.0,-70.0}
};

/*****************************************************************************
        Outputranges (uod)
*****************************************************************************/
 FZ_IO_RANGE OutputRanges[1] = {

/***** SteeringAngle *****************/
          {120.0,-60.0}
};
#endif

/*****************************************************************************
        The next structs contains the definitions for the outputsets!
*****************************************************************************/

/***** SteeringAngle *****************/
 static FZ_NUMERIC Out0Adjs[6] = {
          0.041667,0.25000,
          0.04917,0.09900,
          0.208333,0.25000,
};

/*****************************************************************************
        The next structs contains the definition of the inputsets.
*****************************************************************************/

/***** negativ ***********************/
 static FZ_ADJ_POINT RangeError_negativ[4]={
          {0.0000,1.0000},
          {0.000,1.000},
          {0.500,0.000},
          {1.0000,0.0000},
};
```

```
/***** nearzero *********************/
static FZ_ADJ_POINT RangeError_nearzero[5]={
        {0.0000,0.0000},
        {0.3929,0.000},
        {0.500,1.000},
        {0.6071,0.000},
        {1.0000,0.0000},
};

/***** positiv **********************/
static FZ_ADJ_POINT RangeError_positiv[4]={
        {0.0000,0.0000},
        {0.500,0.000},
        {1.000,1.000},
        {1.0000,1.0000},
};

/***** RangeError *******************/
static FZ_ADJ_PAIR In0Adjs[3] = {
        { 4, RangeError_negativ},
        { 5, RangeError_nearzero},
        { 4, RangeError_positiv},
};

/*****************************************************************************
        The next structs contains the definition of the inputvariables.
*****************************************************************************/
static FZ_ADJ_PAIR *InAdjs[1] = {
        In0Adjs
};

/*****************************************************************************
        The next structs contains the definition of the outputvariables.
*****************************************************************************/
static FZ_NUMERIC *OutAdjs[1] = {
        Out0Adjs
};

/*****************************************************************************
        The next structs contains the definitions for the rules.
*****************************************************************************/

/***** Indexstack for the rules. *****/
static FZ_COMPIDX Index0[] = {
        0,0,2,1,1,1,2,2,0,3,0
};

/***** Operatorstack for the rules. **/
static FZ_OP Bank0[] = {
        0x17,0x4e,0x18,0x4e,0x18,0x4e,0x06
};

/***** Weightstack for the rules. ****/
static FZ_NUMERIC Weights[4] = {
        1.0,
        1.0,
        1.0,
        1.0
};

/***** Rulestack ********************/
static FZ_OP *Rules[1] = {
        Bank0
};

/***** Indexstack *******************/
static FZ_COMPIDX *Indices[1] = {
```

```
        Index0
};

/*****************************************************************************
        Adjectivecount
*****************************************************************************/

/***** Input ************************/
 static FZ_COMPIDX InAdjCounts[3] = {
        3,
        1,
        3
};

/***** Output ***********************/
 static FZ_COMPIDX OutAdjCounts[1] = {
        3
};


/*****************************************************************************
        Mainstruct contains all definitions for the fuzzysystem.
*****************************************************************************/

 struct fz_romparm ROM_Parms = {
        Rules,
        Indices,
        InAdjCounts,
        OutAdjCounts,
        Weights,
        InAdjs,
        OutAdjs,
        _fzr_rules,
        _fzr_memptl,
        _fzr_respsc,
        0,
        4,
        3,
        2
};
```

C. Arbeiten von FuzzyIDE Teammitgliedern

Extension Point Mechanismus

Der Extension Point Mechanismus wurde von Clemens Altenburger und Romain Schmechta entwickelt. Er ersetzte den bestehenden Pluginmanager der FuzzyIDE. Der Mechanismus wurde erfolgreich in die FuzzyIDE eingebunden. Die bestehenden funktionierenden Plugins wurden an den neuen Mechanismus angepasst.

Ruleeditor2 Plugin

Das Ruleeditor2 Plugin ersetzt das bestehende Ruleeditor-Plugin der FuzzyIDE. Beide Plugins dienen dem Anlegen und Bearbeiten der Regeln des Fuzzy-Systems. Das alte Ruleeditor-Plugin wurde aufgegeben und nicht an die neue Version der FuzzyIDE angepasst, da das neue Plugin wesentlich mehr Funktionen als das alte Plugin bietet. Das neue Ruleeditor2-Plugin wurde von Ariel Küchler entwickelt und an die neue Version der FuzzyIDE angepasst.

Variableneditor-Plugin

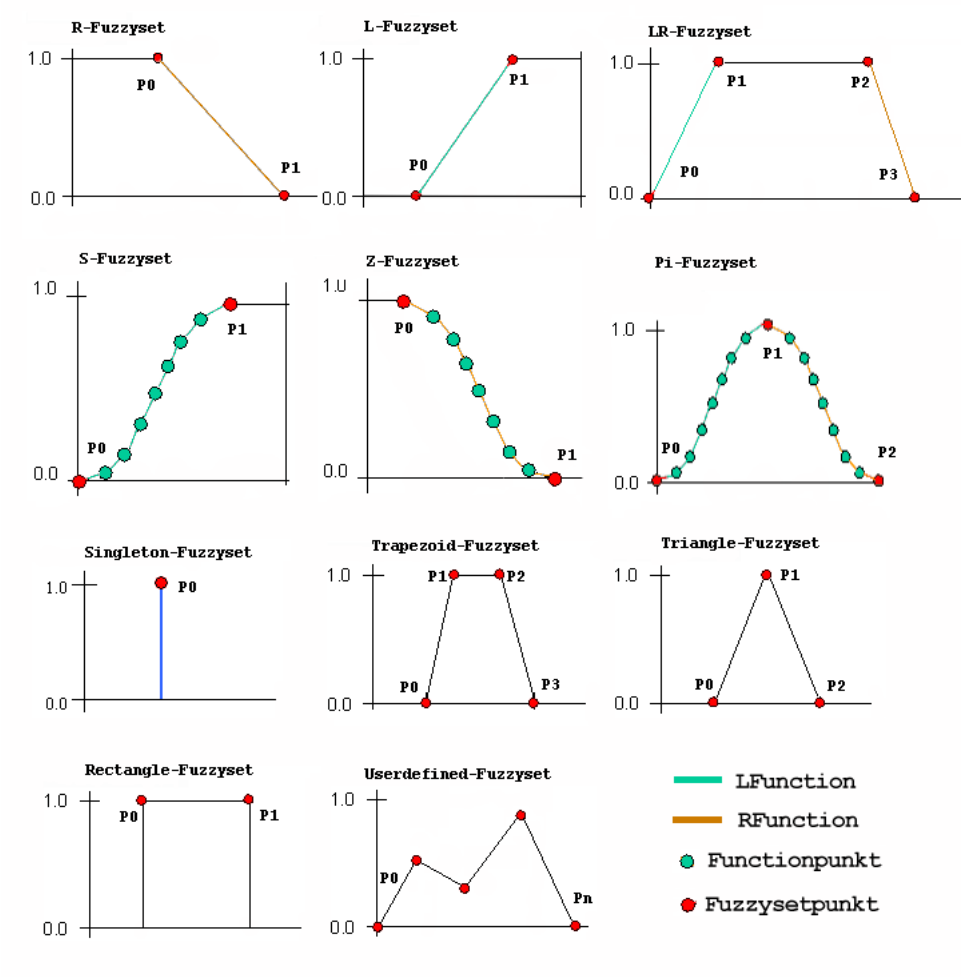
Das Variableneditor-Plugin dient dem Anlegen und Bearbeiten von linguistischen Variablen und deren Ausprägungen im Fuzzy-System. Das Plugin wird von Sascha Becher entwickelt. Es befindet sich noch in der Entwicklung. Eine Version mit eingeschränkten Funktionen kann jedoch bereits eingesetzt werden. Mit der vorhandenen Version kann gearbeitet werden. Noch nicht vollständig implementiert sind die Funktionen zum grafischen Editieren von Fuzzysets.

Fuzzyplot2-Plugin

Das Fuzzyplot2-Plugin wurde von Swen Blobel entwickelt. Das Plugin dient ebenso wie das Fuzzyplot-Plugin dem grafischen Ausgeben von Werten während der Simulation. Im Vergleich zum Fuzzyplot-Plugin bietet das Fuzzyplot2-Plugin einen größeren Funktionsumfang und bessere Handhabung. Das Plugin wurde zu einem Zeitpunkt entwickelt, als die Schnittstellen zur FuzzyIDE schon feststanden. Es musste somit nicht angepasst werden.

D. Fuzzysets

Überblick über die von der FuzzyIDE unterstützten Fuzzysettypen.



In den folgenden Abschnitten werden die abgebildeten Fuzzysettypen näher beschrieben. Die Tabelle "Einstellungen" in jedem Abschnitt bezieht sich dabei auf die Einstellungsmöglichkeiten für den entsprechenden Fuzzysettyp. Die Zugehörigkeitsfunktion für jeden Fuzzysettyp enthält Variablen (z.B. x_0, x_1). Diese beziehen sich auf die Werte der entsprechenden Punkte in der Abbildung D. Sollte die Zugehörigkeitsfunktion eine Funktion $f_L(x)$ oder $f_R(x)$ enthalten, bedeutet dies, dass die Zugehörigkeitsfunktion des Fuzzysets an dieser Stelle durch ein "Function" Objekt definiert wird (siehe Abs. 11.2.4). In der Abbildung sind die Bereiche, die durch ein "Function" Objekt definiert werden können, farbig markiert (siehe Legende in der Abbildung).

D.1. L-Fuzzyset

Einstellungen:

Parameter	Beschreibung	Typ
double	X Wert des Punktes P0	pflicht
double	X Wert des Punktes P1	pflicht
LFunction	Objekt zur Modellierung der Zugehörigkeitsfunktion zwischen P0 und P1	optional

Zugehörigkeitsfunktion:

Ohne Angabe einer LFunction

$$y = \begin{cases} 0 & \text{für } x < x_0 \\ \frac{x-x_0}{x_1-x_0} & \text{für } x_0 < x < x_1 \\ 1 & \text{für } x > x_1 \end{cases}$$

Mit Angabe einer LFunction

$$y = \begin{cases} 0 & \text{für } x \leq x_0 \\ f_L(x) & \text{für } x_0 < x < x_1 \\ 1 & \text{für } x \geq x_1 \end{cases}$$

D.2. R-Fuzzyset

Einstellungen:

Parameter	Beschreibung	Typ
double	X Wert des Punktes P0	pflicht
double	X Wert des Punktes P1	pflicht
RFunction	Objekt zur Modellierung der Zugehörigkeitsfunktion zwischen P0 und P1	optional

Zugehörigkeitsfunktion:

Ohne Angabe einer RFunction

$$y = \begin{cases} 1 & \text{für } x \leq x_0 \\ 1 - \frac{x-x_0}{x_1-x_0} & \text{für } x_0 < x < x_1 \\ 0 & \text{für } x \geq x_1 \end{cases}$$

Mit Angabe einer RFunction

$$y = \begin{cases} 0 & \text{für } x \leq x_0 \\ f_R(x) & \text{für } x_0 < x < x_1 \\ 1 & \text{für } x \geq x_1 \end{cases}$$

D.3. S-Fuzzyset

Einstellungen:

Parameter	Beschreibung	Typ
double	X Wert des Punktes P0	pflicht
double	X Wert des Punktes P1	pflicht
int	Anzahl der Punkte für die SFunction	pflicht

Zugeörigkeitsfunktion:

$$y = \begin{cases} 0 & \text{für } x \leq x_0 \\ 2\left(\frac{x-x_0}{(x_1-x_0)/2} + 1\right)^2 & \text{für } x_0 < x < (x_1 - x_0)/2 \\ 1 - 2\left(\frac{x-x_0}{(x_1-x_0)/2}\right)^2 & \text{für } (x_1 - x_0)/2) < x < x_1 \\ 1 & \text{für } x \geq x_1 \end{cases}$$

Die angegebene Zugehörigkeitsfunktion für den Bereich zwischen x_0 und x_1 entspricht dem Idealwert der Zugehörigkeitsfunktion. Die tatsächlich verwendete Funktion ist abhängig von der Anzahl der Punkte, mit der die SFunction dargestellt wird (siehe Tabelle Einstellungen). Zwischen den einzelnen Punkten der SFunction wird ein linearer Anstieg verwendet.

D.4. Z-Fuzzyset

Einstellungen:

Parameter	Beschreibung	Typ
double	X Wert des Punktes P0	pflicht
double	X Wert des Punktes P1	pflicht
int	Anzahl der Punkte für die ZFunction	pflicht

Zugeörigkeitsfunktion:

$$y= \begin{cases} 1 & \text{für } x \leq x_0 \\ 1 - 2\left(\frac{x-x_0}{(x_1-x_0)/2} + 1\right)^2 & \text{für } x_0 < x < (x_1 - x_0)/2 \\ 2\left(\frac{x-x_0}{(x_1-x_0)/2}\right)^2 & \text{für } (x_1 - x_0)/2) < x < x_1 \\ 0 & \text{für } x \geq x_1 \end{cases}$$

Die angegebene Zugehörigkeitsfunktion für den Bereich zwischen x_0 und x_1 entspricht dem Idealwert für das Fuzzyset. Die tatsächlich verwendete Funktion ist abhängig von der Anzahl der Punkte, mit der die ZFunction dargestellt wird (siehe Tabelle Einstellungen). Zwischen den einzelnen Punkten der ZFunction wird ein linearer Anstieg verwendet.

D.5. Pi-Fuzzyset

Einstellungen:

Parameter	Beschreibung	Typ
double	X Wert des Punktes P1	pflicht
double	Entfernung zwischen P0 und P2	pflicht
int	Anzahl der Punkte für die PiFunction	pflicht

Zugehörigkeitsfunktion:

$$y= \begin{cases} 0 & \text{für } x < x_0 \\ 2\left(\frac{x-x_0}{(x_1-x_0)/2} + 1\right)^2 & \text{für } x_0 < x < (x_1 - x_0)/2 \\ 1 - 2\left(\frac{x-x_0}{(x_1-x_0)/2}\right)^2 & \text{für } (x_1 - x_0)/2) < x < x_1 \\ 1 & \text{für } x = x_1 \\ 1 - 2\left(\frac{x-x_0}{(x_1-x_0)/2} + 1\right)^2 & \text{für } x_1 < x < (x_2 - x_1)/2 \\ 2\left(\frac{x-x_1}{(x_2-x_1)/2}\right)^2 & \text{für } (x_2 - x_1)/2) < x < x_2 \\ 0 & \text{für } x > x_2 \end{cases}$$

Die angegebene Zugehörigkeitsfunktion für den Bereich zwischen x_0 und x_2 entspricht dem Idealwert für das Fuzzyset. Die tatsächlich verwendete Funktion ist abhängig von der Anzahl der Punkte, mit der die PiFunction dargestellt wird (siehe Tabelle Einstellungen). Zwischen den einzelnen Punkten der PiFunction wird ein linearer Anstieg verwendet. Sicher ist jedoch, dass der X-Wert x_1 einen Zugehörigkeitswert von 1 besitzt.

D.6. Triangel-Fuzzyset

Einstellungen:

Parameter	Beschreibung	Typ
double	X Wert des Punktes P0	pflicht
double	X Wert des Punktes P1	pflicht
double	X Wert des Punktes P2	pflicht

Zugehörigkeitsfunktion:

$$\text{y=} \begin{cases} 0 & \text{für } x \leq x_0 \\ \frac{x-x_0}{x_1-x_0} & \text{für } x_0 < x < x_1 \\ 1 & \text{für } x = x_1 \\ \frac{x-x_1}{x_2-x_1} & \text{für } x_1 < x < x_2 \\ 0 & \text{für } x \geq x_2 \end{cases}$$

D.7. Singleton-Fuzzyset

Einstellungen:

Parameter	Beschreibung	Typ
double	X Wert des Punktes P0	pflicht

Zugeörigkeitsfunktion:

$$\text{y=} \begin{cases} 0 & \text{für } x \leq x_0 \\ 1 & \text{für } x = x_0 \\ 0 & \text{für } x \geq x_0 \end{cases}$$

D.8. Trapezoid-Fuzzyset

Einstellungen:

Parameter	Beschreibung	Typ
double	X Wert des Punktes P0	pflicht
double	X Wert des Punktes P1	pflicht
double	X Wert des Punktes P2	pflicht
double	X Wert des Punktes P3	pflicht

Zugeörigkeitsfunktion:

$$\text{y=} \begin{cases} 0 & \text{für } x \leq x_0 \\ \frac{x-x_0}{x_1-x_0} & \text{für } x_0 < x < x_1 \\ 1 & \text{für } x_1 < x < x_2 \\ \frac{x-x_2}{x_3-x_2} & \text{für } x_2 < x < x_3 \\ 0 & \text{für } x \geq x_3 \end{cases}$$

D.9. LR-Fuzzyset

Einstellungen:

Parameter	Beschreibung	Typ
double	X Wert des Punktes P0	pflicht
double	X Wert des Punktes P1	pflicht
double	X Wert des Punktes P2	pflicht
double	X Wert des Punktes P3	pflicht
LFunction	Objekt zur Modellierung der Zugehörigkeitsfunktion zwischen P0 und P1	pflicht
RFunction	Objekt zur Modellierung der Zugehörigkeitsfunktion zwischen P2 und P3	pflicht

Zugehörigkeitsfunktion:

$$\text{y=} \begin{cases} 0 & \text{für } x \leq x_0 \\ f_L(x) & \text{für } x_0 < x < x_1 \\ 1 & \text{für } x_1 < x < x_2 \\ f_R(x) & \text{für } x_2 < x < x_3 \\ 0 & \text{für } x \geq x_3 \end{cases}$$

D.10. Rectangle-Fuzzyset

Einstellungen:

Parameter	Beschreibung	Typ
double	X Wert des Punktes P0	pflicht
double	X Wert des Punktes P1	pflicht

Zugehörigkeitsfunktion:

$$y = \begin{cases} 0 & \text{für } x < x_0 \\ 1 & \text{für } x_1 \leq x \leq x_2 \\ 0 & \text{für } x > x_1 \end{cases}$$

D.11. Userdefined-Fuzzyset

Einstellungen:

Parameter	Beschreibung	Typ
Vector	Vektor mit FuzzySetPoint Objekten.	pflicht

Zugehörigkeitsfunktion:

Für das Fuzzyset kann keine Zugehörigkeitsfunktion angegeben werden. Die Zugehörigkeitsfunktion ergibt sich aus den FuzzySetPoint Objekten, die dem Konstruktor übergeben werden. Zwischen den einzelnen Punkten des Sets wird ein linearer Anstieg verwendet.

E. CD-Inhalt

Auflistung der wichtigsten Verzeichnisse auf der beigelegten CD:

Diplomarbeit/: Diplomarbeit in lesbarer Form als PDF-Dokument und Latex-Dateien.

FuzzyIDE_Projekt/: Dateien des FuzzyIDE Projektes.

- **CVS_AusgangsStand/:** Ausgangstand des FuzzyIDE Projektes.
- **CVS_Abgabestand/:** Abgabestand des FuzzyIDE Projektes.
- **Distribution/:** Im Verlauf des Redesigns erstellte Distribution.
  - **FuzzyIDE/:** Distribution als Java-Projekt.
  - **FuzzyIDE_EXE/:** Mit JSmooth erzeugte Exe-Version der Distribution.

Codegen/: Mit Hilfe des Codegenerations-Plugins erzeugte Quellcodedateien.

- **CubiCalc/:** Erzeugter C-Quellcode.
  - **RUNTIME/:** CubiCalc Runtime Bibliothek Quellen.
  - **example/:** Die Beispiele aus der Distribution als C-Quellcode.
- **NRC/:** Erzeugter Java-Quellcode.
  - **lib/:** Benötigte NRC-Bibliothek [NRC].
  - **example/:** Die Beispiele aus der Distribution als Java-Quellcode.

Tools/: Verwendete Werkzeuge.

- **JSmooth/:** EXE-Wrapper zum Erzeugen der Exe-Version der FuzzyIDE.
- **JEP/:** Java Math Expression Parser (siehe Abs. 11.2.4)[JEP].
- **JLex/:** Lexergenerator JLex [JLE].
- **JAY/:** Parsergenerator JAY [JAY].

CBC_examples/: Die Originalbeispiele aus CubiCalc, die als Vorlage für die Beispiele der FuzzyIDE Distribution verwendet wurden.

startFuzzyIDE.bat: Die Datei startet die FuzzyIDE.

Einige Verzeichnisse enthalten "README.txt" Dateien, diese erleutern den genauen Aufbau der Verzeichnisse, sowie die Anwendung der enthaltenen Dateien.

Thesen

1. Die im ursprünglichen Datenmodell auftretenden Probleme: Referenzen auf veraltete Objekte und doppelte Datenhaltung, welche auf Grund der gekapselten NRC-Objekte auftreten, machen grundlegende Änderungen am Datenmodell notwendig.

2. Zum effektiven Nutzen der FuzzyIDE ist es notwendig, den bestehenden Simulator zu ersetzen, da er nur schwer an unterschiedliche Anforderungen (Simulationen) angepasst werden kann.

3. Für eine effektive Nutzung des neuen Simulators und der Simulatorplugins ist es notwendig, mehrere Plugins während einer Simulation zu verwenden und die anstehenden Aufgaben der Simulation auf spezialisierte Plugins zu verteilen.

4. Mit dem neuen Simulator der FuzzyIDE, welcher mehre Plugins nutzen kann, ist es möglich, sehr unterschiedliche Simulationen mit den bestehenden Plugins durchzuführen.

5. Für die effektive Nutzung des Simulators wird ein Simulatorplugin benötigt, mit dem die Programmumgebung des zu erstellenden Fuzzy-Systems simuliert werden kann. Diese Funktionalität wurde mit Hilfe des IDEInterpreter-Plugins erreicht.

6. Da die NRC-Bibliothek nach dem Redesign nur noch in der Klasse "Calculation" des Simulators verwendet wird, ist die FuzzyIDE wesentlich unabhängiger von der NRC-Bibliothek. Dies ermöglicht es in Zukunft, eventuell auch andere Bibliotheken einzusetzen oder die FuzzyIDE so zu erweitern, dass sie die Fuzzy-Berechnungen selbst vornehmen kann.

7. Nach der Entwicklung eines Fuzzy-Systems ist es notwendig, das System in einer Form exportieren zu können, in der es in einem eigenen Programm angewendet werden kann. Um dies zu ereichen, ist es notwendig, das Fuzzy-System als Quellcode einer Programmiersprache auszugeben.

8. Die im Verlauf des Redesigns erstellte Distribution bietet ausreichende Funktionalität, um ein Fuzzy-System zu erstellen und dieses zu testen

Glossar

ANT: Open Source Werkzeug zum Kompilieren von (Java-) Projekten. Die notwendigen Arbeiten beim Übersetzen des Projektes werden in einer XML-Datei(build.xml) gespeichert.

Ausprägung: Ein Wert, den eine linguistische Variable annehmen kann. Der Ausprägung ist in einen Fuzzy-System eine unscharfe Menge(Fuzzyset) zugeordnet

Compiler: Programm zum Übersetzen eines Quelltextes einer Programmiersprache in ein semantisch äquivalentes Programm einer Zielsprache.

CVS: (Concurrent Versions System) Programm zur Versionskontrolle. Es ermöglicht das parallele Arbeiten mehrerer Entwickler an einem Projekt.

Defuzzyfiezirung: Operation zum Umwandeln einer unscharfen Menge in einen scharfen Wert.

Fuzzy-Rege:l Eine Wenn-Dann-Regel in der Wissensbasis eines Fuzzy-Systems.

Fuzzy-System: System, das Inputwerte mit Hilfe von fuzzylogischen Berechnungen in Outputwerte umwandelt.

Fuzzy-Variable: Linguistische Variable in einem Fuzzy-System.

Fuzzyfiezirung: Operation zum Umwandeln eines scharfen Wertes in eine unscharfe Menge.

FuzzyIDE: Fuzzy-Entwicklungsumgebung, der Name setzt sich zusammen aus fuzzy(engl. unscharf) und IDE (Integrated Development Environment) für integrierte Entwicklungsumgebung.

Fuzzyset: (engl. fuzzy set) Der Begriff setzt sich zusammen auf dem englischen Wort fuzzy für unscharf und dem englischen Wort set für Menge. Der Begriff beschreibt also eine unscharfe Menge.

Interpreter: Programm, das eine virtuelle Maschine auf einer realen Maschine umsetzt. Ein Interpreter führt den Code, der für eine virtuelle Maschine erzeugt wurde, auf einer realen Maschine aus.

JAY: Open Source Tool zum Generieren eines Parsers. Von Jay generierte Parser werden als Java-Klasse ausgegeben. Die Klasse enthält ein Interface. Über dieses wird der benötigte Lexer aufgerufen.

JLex: Open Source Tool zum Erzeugen eines Lexers. Von JLex generierte Lexer werden als Java-Klasse ausgegeben.

Lexer: Kurzform für einen lexikalischen Scanner. Ein Programm, das einen Text nach bestimmten Gesichtspunkten in Morpheme(auch Atome, Morpheme, Token) zerlegt. Ein Lexer wird als Teil eines Compilers benötigt. Die erkannten Morpheme unterscheiden sich von Lexer zu Lexer.

Linguistische Variable: Eine Variable, die als Werte Worte in einer natürlichen oder künstlichen Sprache annehmen kann.

Member: Datenelement einer Klasse.

OS: (Operating System) Betriebssystem.

Parser: Programm oder Programmteil, der prüft, ob ein Eingabetext einer bestimmten Grammatik entspricht. Wird als Teil eines Compilers benötigt, um den Eingabequelltext zu prüfen und Funktionen aufzurufen.

Plugin: Ergänzungs- oder Zusatzmodul für ein bestehendes Programm, das über eine Schnittstelle(Pluginschnittstelle) angebunden wird.

Redesign: Englisch für neu entwerfen. Beschreibt die Überarbeitung eines bestehenden Programms.

Referenz: Verweis auf ein (Java-)Objekt. Über eine Referenz kann auf die Methoden und die öffentlichen Datenmember des Objektes zugegriffen werden.

Term: Siehe Ausprägung.

Thread: Leichtgewichtiger Prozess. Stellt einen Ausführungsstrang in einem Prozess dar. Er teilt sich mit den andern Threads des Prozesses die Systemressourcen(Speicher, geöffnete Dateien usw.).

UoD: (engl. univers of discus) Basisskala einer Fuzzy-Variablen.

Wissensbasis: Eine Reihe von Fuzzy-Regeln, in welchen das Expertenwissen des Fuzzy-Systems gespeichert ist.

XML: (Extensible Markup Language) Die Abkürzung steht für erweiterbare Auszeichnungs-Sprache. Die Sprache ist ein Standard für das Erstellungen von Dokumenten in Form einer Baumstruktur.

Literaturverzeichnis

[ANT] *Dokumentation Ant.* http://ant.apache.org/manual/index.html.

[Beca] Prof. A. Beck. *Unterlagen aus dem Fach "Compiler/Interpreter".*

[Becb] Prof. A. Beck. *Unterlagen aus dem Fach "JAVA".*

[Cub93] *CubiCalc Manual*, 2nd edition, 1993.

[CuR92] *CubiCalc RTC Manual*, 1st edition, 1992.

[Ess02] Friedrich Esser. *Java2.* Galileo Computing, 1st edition, 2002.

[Her99] Helmut Herold. *Linux-Unix-Kurzreferenz.* Addison-Wesley, 2nd edition, 1999.

[Iwe] Prof. H. Iwe. *Unterlagen aus dem Fach "Wissensbasierte Fuzzy-Systeme".*

[JAY] *Dokumentation Jay.* http://www.cs.rit.edu/ ats/projects/lp/doc/jay/package-summary.html.

[JEP] *Dokumentation Java Math Expression Parser.* http://www.singularsys.com/jep/.

[JLE] *Dokumentation JLex.* http://www.cs.princeton.edu/ appel/modern/java/JLex/current/manual.html.

[JSM] *Dokumentation Jay.* http://jsmooth.sourceforge.net/features.php.

[NRC] *Dokumentation NRC FuzzyJ Toolkit.* http://www.iit.nrc.ca/IR_public/fuzzy/.

[Pil03] Dan Pilone. *UML. Kurz und gut.* O'Reilly, 2003.

[uDW04] Uwe Schneider und Dieter Werner. *Taschenbuch der Informatik.* Hanser Fachbuchverlag, 2004.

[uEW95] Doris Danziger und Egon Weiner. Fuzzyentwicklungsumgebungen. Technical report, Fachhochschule Münster, 1995.

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die Diplomarbeit selbständig und ohne unzulässige Hilfe Dritter verfasst habe. Ich versichere zudem, dass keine anderen Quellen und Hilfsmittel als die angegebenen verwendet wurden.

_____

Thomas Kummer