folgende Begriffe sollen definiert werden:
Visual Programming Language
Grammatik
Domain Specific Language
Schleifen
Flowchart
(Fixpunktberechnung)

Contents

1 Einleitung **3**

2 Terminologie **4**
2.1 Schleifen 4
2.2 Domain Specific Language 4
2.3 Visual Programming Language 5
2.4 Visual Language 5
2.5 Datenfluss-basierte Sprachen 5
2.6 Datenfluss-basierte Systeme 6

3 Aufbau der domainspezifischen Sprache **7**
3.1 Prüfungslogik 7
3.2 Datenverarbeitungs 9
3.3 Typsystem 11

4 Implementierung **12**
4.1 1. Lösungsansatz 12
4.2 2. Lösungsansatz 13

5 Evaluation **15**
5.1 15
5.1.1 1. Lösungsansatz 15

6 Literaturverzeichnis **16**

1 Einleitung

2 Terminologie

2.1 Schleifen

Eine Schleife ist eine Kontrollstruktur, welche einen Code-Abschnitt mehrmals ausführt. [21] Sie stellt dabei oftmals die zeitintensivste Komponente eines Programms dar, weil die Ausführung der Schleife sehr viel Zeit in anspruch nehmen kann. [6] Der Algrorithmus der Kontrollstruktur kann bei iterativ oder rekursiv umgesetzt werden. Beide Ansätze haben dabei das gleiche Ziel, aber setzen die Schleife anders um. Bei der Iteration wird die Schleife mehrmals wiederholt wird. Bei der Rekursion ruft sich die zu wiederholende Funktions mehrmals selbst auf. Die Iteration verwendet dabei einen Akkumulativenansatz, welcher das zu lösende Problem schrittweise löst und diesen Vorgang solange wiederholt bis eine vordefinierte Abbruchbedingung erreicht wurden ist. Die Rekursion verwendet keinen Akkumulativenansatz, sondern reduziert das eigentlich zu lösende Problam auf mehrere (Teil-)Probleme. Für die Teilprobleme werden dann einzelne Lösungen gesucht, welche anschließend zusammengesatz werden um das eigentliche Problem zu lösen. [2] Laut Chen L. spiegelt die Iteration das menschliche Denken wieder und ist daher besonders für lineare Probleme geeignet. Hingegen die Rekursion für lineare oder sequentielle Probleme geeignet sind, welche Zwischenergebnisse oder Teillösungen benötigen. [2] Die Iteration lässt sich dabei in zeitabhängige oder horizontale Iteration unterteilen. Bei der zeitabhängigen Iteration ist das Ergebniss des aktuellen Schleifendurchlaufes von dem Ergebniss des vorherigen Schleifendurchlaufes abhängig. Hingegen bei der horizontalrn Iteration die Ergbenisse der Schelifendurchläufe unabhängig voneinander sind. [4]

2.2 Domain Specific Language

Eine domänenspezifische Sprache (DSL) ist eine Programmiersprache, welche für einen bestimmten Anwendungsbreich entwickelt wurde und normalerweise auch auf diesen beschränkt ist. Das Ziel einer DSL ist es in dem klar abgegrenzenten Anwendungsbereich die Probleme effizient zu lösen. [18] Dabei fallen DSL nicht in die Gruppe der General-Purpose Language (GPL) wie z.B Java, C++ oder Python, sondern bildern das Gegenstück dazu. [15] Die Syntax ist dabei oftmals eingeschränkter und erlauben nur eine bestimmte auswahl an Notationen und Befehlen. In manchen Fällen hat eine DSL aber auch eine GPL als Zweitsprache. [18] DSLs lassen sich in externe und interene unterteilen. Externe DSLs haben ihre eigene Syntax. Dadurch kann eine größere flexibilität geschaffen werden, aber zeitglich ist der Aufwand für den Entwickler sehr hoch, weil alle relevanten Tools selbst implementieren muss. Außerdem braucht der Benutzer länger Zeit um die Syntax zu lernen. [7] Zur Laufzeit wird die externe DSLs dann in eine GPL übersetzt. [15] Interne DSLs verwenden die Syntax einer GPL und kann über eine Programmierschnittstelle oder Bibliothek aufgerufen werden. [15] Allgemein sind DSLs kompakt, wiederverwendbar, effizient und domänenspezifisch. Aber auf der anderen Seite ist die erstellung einer DSL kost-

spielig und haben einen hohen Lernaufwand für den Benutzer. Zudem haben sie nur ein eingeshcränkten Anwendungsbereich und sind nur begrenzt Verfügbar. [18]

2.3 Visual Programming Language

Das Ziel von VPLs ist es die Darstellung der Programmierlogik zu verbessern und den Ablauf des Programms zu verstehen. [14] Außerdem soll der Benutzer sich mehr auf die Implementierung des ALgortithmus konzentrieren anstatt auf die Syntax, weil diese auf die IDE übertragen wird. [10] Das schaffen VPLs indem sie es erlauben Programme direkt mithilfe von Flussdiagrammen zu erstellen, welche vom Computer direkt interpretiert und ausgeführt werden können. [13] Laut Charntaweekhun eignen sich Flussdiagramme sehr gut zum Programmieren, weil vorallem viele Einsteiger in der Programmierung erstmal Flussdiagramme erstellen um das Problem zu visualieren. [13] VPLs setzen dabei das Konzept der Visual Programming (VP). Bei VPLs stehen dem Programmierer nur ein bestimmter Satz an grafischen Elementen gegenüber statt dem ganzen Alphabet wie bei GPLs. Dadurch ist das Programm leichter zu verstehen und Fehler z.B. Semantik oder Syntax Fehler lassen sich bereits beim erstellen des Programms vermeiden. [10] VPLs lassen sich dabei in Imperativ und Deklarativ unterteilen. In einer imperativen VPL wird vom Programm vorgegeben, in welcher Reihenfolge die Operationen ausgeführt werden. Bei einer deklativen VPL hingegen wird vom Programm nur die Abhänigkeit zwischen Daten vorgegeben und das System bestimmt die Reihenfolge selbst. [12] VPLs haben den Vorteil, dass diese einfach, visuell darstellbar sind, tranparent und Interaktiv sind. Einfachheit, weil weniger Programmierkonzepte zum Programmieren benört werden. Visuell darstellbar, weil Transparent, weil Datenabhänigkeiten anschaulich dargestellt werden. Interaktiv, weil der Entwickler direkt Feedback bekommen kann. [14] Bei VPL ist das meist genutze Paradigma der Datenfluss-basierte Ansatz. [5] Zusammengefasst kann man sagen, dass VPLs die Vorteile von Flussdiagrammen und nicht die Nachteile der klassischen Programmierung kombiniert. [14]

2.4 Visual Language

Visual Language (VL) drücken sich eher mit Bildern statt Texten aus. [4] Dabei werden hauptsächlich grafische Tools und visuelle Metaphoren verwendet. Bilder eignen sich besonders gut zum Programmieren, weil Bilder ausdruckstärker als Worte sind und haben einen höheren Wiedererkennungswert. Durch die eingeschränkte Syntax sind VLs nicht so flexibel und ausdruckstar wie Text-basierte Sprachen. [19]

2.5 Datenfluss-basierte Sprachen

Unter einer Datenfluss-basierten Sprache (DL) versteht man, dass die Daten von einer Funktion in die andere geht. Dabei wird das Programm als Graphen

dargestellt[11] Beim Graphen handelt es sich um einen gerichteten Graphen (DG). Die Funktionen werden als kreisförmige Knoten (Node) dargestellt. Die Nodes können durch gerichtite Pfeile miteinander verbunden werden. Dabei beschreiben die Pfeile die Datenabhänigkeiten im Graphen.[1] Der DG lässt sich in feinkörnig und Grobkörnig unterteilen. Feinkörnig bedeutet, dass jeder Knoten genau eine Instruktion durchführt. Beim Grobkörnig hingegen kann ein Knoten mehrere Instruktionen aufienmal ausführen-[6] Zudem lässt sich ein DG basierend auf der Zyklenstruktur in zyklisch und azyklisch unterteilen. [8] DLs sind oftmals funktionale Programmiersprachen, aber können auch textbasiert sein. [1] Der Vorteil einer DLs ist, dass diese durch einen Graphen dargestellt werden können [11] und dadurch die Programme einfach zur verstehen sind. [5] Da ein Programm viele Funktionen haben kann, kann ein Graph schnell unübersichtlich werden. Damit dies vermieden werden kann, gibt es sogenannte Mikrofunktionen. Mikrofunktionen sind Knoten, welche auf einen Teilgraphen verweisen. Der Teilgraph beinhaltet dabei die eigentliche Darstellung des Algorithmuses. Durch diese Möglichkeit lassen sich auch ganz einfach Rekursionen in einem Graphen darstellen.[11] Eine DL führt den Code nicht streng sequentiell aus. Das führt dazu, dass unabhängige Instruktionen parellel ausgeführt werden können.[6] Durch diese Ausführung kann in den meisten ein Effizentsteigerung geschaffen werden, weil das Programm nicht mehr vom Programmzähler abhängig ist. [1] Johnston et. al beschreiben in ihrer Wissenschaftlichenarbeit eine Menge von Eigenschaften. So sollen DLs frei von Seiteneffekten sein, den Lokalitätsprinzip folgen und keine Variablen überschreiben.[1]

2.6 Datenfluss-basierte Systeme

Datenfluss-basierte Systeme (DFA) ist eine Computerarchitektur, welche auf DLs basiert. Die DFA wurde eingeführt um den Flaschenhals der von-Neumann-Architektur zu vermeiden. Je nach Implementierung kann nur lokaler Speicher verwendet werden und die Funktionen können sofort aufgefürt werdne, sobald die Operanden zur verfügung stehen. [8] Die Vorteile einer DFA sind, dass diese hohe Performance, flexibilität und hohe efektivität fördert. [6] Die Ausführung kann dabei Datengesteuert oder Bedarfsgesteuert sein. Bei einer Bedarfgesteuerten Ausführen werden die Funktionen ausgeführt, sobald diese ein Signal über ihr Ausgangspfeil bekommt und alle benötigten Operanden vorhanden sind, Hingegen bei der Datengesteuerten Ausführung wird die Funktion sofort ausgeführt, sobald alle benötigten Operanden vorhanden sind. [1] parallelismus, weil mehr als eine Instruktion gleichzeit ausgeführt werden kann. da datenabhänigkeiten überprüft werden. In einem DFA fließen Daten als Token durch das System. Schaut man sich die beiden Ausführungen genauer an, kann man sagen, dass die Datengesteuerte Ausführung nichts anderes als eine Bedarfsgesteuerte Ausführung ist, bei der bereits ders Bedarf an allen Ergebnissen vorhanden ist.[11] Bei der Ausführung fließen die Ergbenisse einer Funktion direkt in eine andere und werden dort tranfsformiert oder gefilter. [16]

3 Aufbau der domainspezifischen Sprache

Die zugrundeliegende Grammatik basiert auf der Backus-Naur-Form (BNF) Notation. Der Aufbau einer BNF wird anhand der Grammatik 3 erklärt ¡symbol¿ sind nichtterminale ::= bedeutet dass symbol durch _expression_ersetzt wird _expression_ist eine sequenze von nichterminalen und terminale Kleene-Stern * wiederholung Alternation |oder Sequenz erlaubt auch Klammern um die Reihenfolge der Regel zu definieren Softwareprüfung läässt sich visuall von zwei seiten betrachen.

⟨*symbol*⟩ ::= _expression_

Grammatik TODO Backus-Naur-Form

Grammatik lässt sich in 3 Ebenenunterteilen Prüfungslogik, Datenverarbeitung und Typsystem Prüfungslogik führt Entscheidung im Prüfungsablauf aus und bestimmt die Reihenfolge der Aktionen. außerdem datenerfassung Datenverarbeitung ist für die Datentranformation auswertung zustädnig. Also Funktionen, welche keine Nebeneffekte besitzen, weil sie unabhängig von der restoichen Softwareprüfung stattfinden. Typsystem ermöglicht die statische analyse der ausführbarkeit Softwareprüfung läässt sich visuall von zwei seiten betrachen. Einmal als Datenflussgraphen, indem Teil-Funktionen als Blöcke dargestellt werden und Funktionsparamter/Ergebnisse als Ports. Einmal als Aktivitätsdiagramm, in dem nur Startzustand, Endzustände, Aktions- und Entscheidungsblöcke dargeste

3.1 Prüfungslogik

Das Aktivitätsmodell $< ActivityModel >$ besteht aus einer Reihe von Aktivitäten $< Activity >$. Aktivitäten können dabei entweder eine Startmarkierung, eine Aktivitätsaktion, einem Vergleich oder visualles Label sein. Die Startmarkierung muss pro Prüfungslogik genau einmal vorkommen. Ein vergleich kann dabei entweder eine Binärentscheidung oder eine Validierungsentscheidung sein. Die Validierungsentscheidung nimmt als Eingabe einen Wert und überprüft ob diese Werte vorhanden sind. Die Binärentscheidung nimmt als Parameter zweite Werte, einen Vergleichoperatur und eine referenz zu einer Funtkion. Dabei werden beide Werte als Eingabe für die referenzierte Funktion verwendet. Als Vergleichoperatoren stehen $=$ und $\neq$ sowie Relationale Operatoren zur verfügung. Eine Aktivitätaktion ¡ActivityAction¿ kann dabei einer der folgenden Aktionen ausführen: Bevor das Ergbeniss aus der vorherigen Aktionsaktivität verwendet wird, kann eine Transformation auf dieses Ergebniss angewendet werden.

- Senden von Hauptuntersuchungs-Adatper-Anfragen (A1)
- Lesen einer JSON Datei (A2)
- Ausführung einer Datenverarbeitung (A3)

A1 nimmt als Parameter den Namen der auszuführenden Anfrage, eine Beschreibung für den debugger, eine Liste von anzusprechnenden System im Fahrzeig und die maximale Zeitdrauer einer Anfrage. A2 nimmt als Eingabe den Typ der zu ladenenden Datei und die dazugehörige URI. A3

⟨*ActivityModel*⟩ ::= ⟨*Activity*⟩* ⟨*ActivityConnection*⟩

⟨*Activity*⟩ ::= ⟨*ActivityStart*⟩ | ⟨*ActivityAction*⟩ | ⟨*ActivityCondition*⟩ | ⟨*ActivityDisplay*⟩

⟨*ActivityStart*⟩ ::= ϵ

⟨*ActivityAction*⟩ ::= ⟨*ActivityFlowCall*⟩ | ⟨*ActivityPitaBuildInforRequest*⟩ | ⟨*ActivityLoadExternalData*⟩

⟨*ActivityFlowCall*⟩ ::= ref(FlowTemplate) ⟨*ActivityPortValue*⟩* ⟨*TemplateParameterValue*⟩*
⟨*ValueTransformation*⟩*

⟨*ActivityPitaBuildInforRequest*⟩ ::= ⟨*string abdFilename*⟩ ⟨*string requestAlias*⟩
⟨*string expectedSystems*⟩* ⟨*number timeout*⟩

⟨*ActivityLoadExternalData*⟩ ::= ⟨*Type dataType*⟩ ⟨*string dataSource*⟩

⟨*ActivityPortValue*⟩ ::= ⟨*FlowPortValue*⟩ | ⟨*ActivityPortRefernce*⟩

⟨*FlowPortValue*⟩ ::= ⟨*string*⟩ | ⟨*number*⟩ | ⟨*bool*⟩ | ⟨*date*⟩ | ⟨*FlowPortValue*⟩*

⟨*ActivityPortRefernce*⟩ ::= ref(ActivityAction) (ValueTransformation)*

⟨*ValueTransformation*⟩ ::= ⟨*string objectReference*⟩ | ⟨*number listIndex*⟩

⟨*ActivityCondition*⟩ ::= ⟨*ActivityBinaryCondition*⟩ | ⟨*ActivityValidityCondition*⟩

⟨*ActivityBinaryCondition*⟩ ::= ref(FlowTemplate) ⟨*ActivityBinaryConditionOperator*⟩
⟨*ActivityPortValue left*⟩ ⟨*ActivityPortValue right*⟩

⟨*ActivityValidityCondition*⟩ ::= ⟨*ActivityPortValue*⟩*

⟨*ActivityBinaryCondition*⟩ ::= '=' | '≠' | '<' | '≤' | '>' | '≥'

⟨*ActivityDisplay*⟩ ::= ⟨*ActivityDisplayField*⟩*

⟨*ActivityDisplayField*⟩ ::= ⟨*string label*⟩ ⟨*string color*⟩ ref(ActivityAction)

Grammatik TODO Aktivitätsmodell

3.2 Datenverarbeitungs

Eingabe ¡FlowInputPort¿ und Ausgabe $< FlowOutputPort >$ Funktionen höherer Ordnung $< FlowLamda >$ Eine Funktion höher Ordnung besteht aus zustzälciehn Eingabe- und Ausgabeports Eingabe- und Ausgabeports nehmen als Parameter einen Namen des Ports, den Typ und ob Fehlererlaub ist.

Ein Funktions Template $< FlowTemplate >$ besteht aus einer Funktion $Flow$ und belieg vielen Parametern $< TemplateParameter >$ Die Parameter generieren Port- und Lambda-Defintion Funktionen welche vom Autorensystem $< LibraryFlow >$ und selbst definierte Funktionen $< FlowModel >$

Ein Flow-Modell ist ein DAG bei dem mehrere Funktionen mitienander verbunden werden. Einzelne Funktionen werden Nodes $< FlowNode >$ genannt. Das Flow-Modell wird durch eine $< FlowInstance >$, Reihe von Funktionen und Verbidnugen definiert. Die Funktion kann dabei eine Eingabe ¡$FlowNodeInput >$, eine Ausgabe $< FlowNodeOutput >$, einer Lambda Referenz $< FlowNodeLambda >$ oder eine Funtkions Referenz $FlowNodeFlowCall$ Konstante Werte $FlowPortValue$

⟨*FlowInstance*⟩ ::= ⟨*FlowOutputPort lambdaArguments*⟩* ⟨*FlowInputPort lambdaArguments*⟩* ⟨*FlowLambda*⟩*

⟨*FlowLambda*⟩ ::= ⟨*FlowOutputPort lambdaArguments*⟩* ⟨*FlowInputPort lambdaArguments*⟩*

⟨*FlowInputPort*⟩ ::= ⟨*string name*⟩ ⟨*Type*⟩ ⟨*bool acceptsError*⟩

⟨*FlowOutputPort*⟩ ::= ⟨*string name*⟩ ⟨*Type*⟩ ⟨*bool producesError*⟩

Grammatik TODO Flow-Instanz

⟨*FlowTemplate*⟩ ::= ⟨*Flow*⟩ ⟨*TemplateParameter*⟩*

⟨*Flow*⟩ ::= ⟨*LibraryFlow*⟩ | ⟨*FlowModel*⟩

⟨*LibraryFlow*⟩ ::= ϵ

⟨*TemplateParameter*⟩ ::= 'String' | 'Number' | 'Bool' | ⟨*TemplateParameterList*⟩

⟨*TemplateParameterList*⟩ ::= ⟨*TemplateParameter*⟩

Grammatik TODO Flow-Template

⟨*FlowModel*⟩ ::= ⟨*FlowInstance*⟩ ⟨*FlowNode*⟩* ⟨*FlowConnection*⟩*

⟨*FlowNode*⟩ ::= ⟨*FlowNodeOutput*⟩ | ⟨*FlowNodeInput*⟩ | ⟨*FlowNodeLambda*⟩ | ⟨*FlowNodeFlowCall*⟩

⟨*FlowNodeOutput*⟩ ::= ref(FlowOutputPort) ⟨*FlowPortValue*⟩

⟨*FlowNodeLambda*⟩ ::= ref(FlowLambda) ⟨*FlowPortValue*⟩*

⟨*FlowNodeFlowCall*⟩ ::= ref(FlowTemplate) ⟨*FlowPortValue*⟩* ⟨*TemplateParameterValue*⟩*

⟨*FlowConnection*⟩ ::= ref(FlowOutputPort source) ref (FlowOutputPort target)

⟨*FlowConnection*⟩ ::= ref(FlowOutputPort source) ref (FlowOutputPort target)

⟨*TemplateParameterValue*⟩ ::= ⟨*string*⟩ | ⟨*number*⟩ | ⟨*bool*⟩ | ⟨*TemplateParameterValueList*⟩

⟨*TemplateParameterValueList*⟩ ::= ⟨*TemplateParameterValue*⟩*

Grammatik TODO Flow-Modell

3.3 Typsystem

unterstüzt die gleichen Primitiv-Typen wie JSON-Format String, Number und Bool zusätzlich Date und PtiaResponse. Außerdem werden auch gernerische Typen unterstüztz, weil nicht immer von vorneherein der Typ bekannt ist. Date ist eine Datumsangabe PtiaResponse ist eine Antwort einer Hauptuntersuchungs-Anfrage Diese Typen lassen sich an optionalen, Listen oder Objekt-Typen kapseln

⟨*Type*⟩ ::= ⟨*TypePrimtive*⟩ | ⟨*TypeOptional*⟩ | ⟨*TypeList*⟩ | ⟨*TypeObject*⟩

⟨*TypePrimtive*⟩ ::= 'String' | 'Number' | 'Bool' | 'Data' | 'PtiaResponse'

⟨*TypeOptional*⟩ ::= ⟨*Type*⟩ '?'

⟨*TypeList*⟩ ::= ⟨*Type*⟩ '[]'

⟨*TypeObject*⟩ ::= '{' (⟨*string key*⟩ ':' ⟨*Type*⟩)* '}'

⟨*TypeGeneric*⟩ ::= '$' ⟨*string genericName*⟩

⟨*TypeReference*⟩ ::= ref(Type)

Grammatik TODO Typ-Defintion mit generischen und Referenz-Typen

4 Implementierung

Bei der Implementierung muss nicht nur auf des Design des Schleifenkonstrukt geachtet werden, sondern auch auf neue Sachen, welche durch die Implementierung entstanden sind. Bei den Schleifendurchläufen wird nicht auf die Ergebnisse des letzten Durchlaufs zugegriffen werden, sondern der Schleifenkörper soll die Entscheidungen auf Grundlage des aktuellen Sensorwertes treffen. Das auslesen des aktuellen Sensorwerts ist bereits möglich. Aktuell unterstützt die zugrundeliegenede Implementierung noch keine Variablen. Um das zu ändern muss die Grammatik bearbeitet werden-

4.1 1. Lösungsansatz

Der Benutzer gibt von vorneherein eine Zahl a an, welche die maximale Anzahl von Schleifendurchläufe beschränkt. Für die Zahl muss dafür folgendes gelten TODO. Die Idee des Ansatzes ist es, den Schleifenköper nicht Iterativ oder Rekursiv ausführen, sondern a-mal auszurollen. Dafür wird der Schleifenköper und die nachfolgenden Anweisungen a-mal kopiert. Die Schleife wird dadurch nicht dynmasisch ausgeführt, sondern statisch in den Code implementiert. Dadurch entstehen $a+1$ Graphen. Jeder dieser Graphen repräsentiert einen ursprüngliche Iteration. Dabei werden die einzelnen Graphen mit ihren direkten Nachbaran verbunden. Da die aktuell zugrunde liegende Implementierung determenistisch ist und aktuell nur auf die gleiche Eingabe zugegriffen werden kann, muss ein Mechanismus implementiert werden, welcher den aktuellen Sensorwert ausliest und diesen an die nachfolgenden Anweisungen weitergibt. Dieser Vorgang muss für jede neu eingefügte Verbindung wiederolt werden. Dieser Ansatz wird auch von Ye et al. im Konferenz-Paper "Loop Unrolling Based on SLP and Register Pressure Awareness" beschrieben.

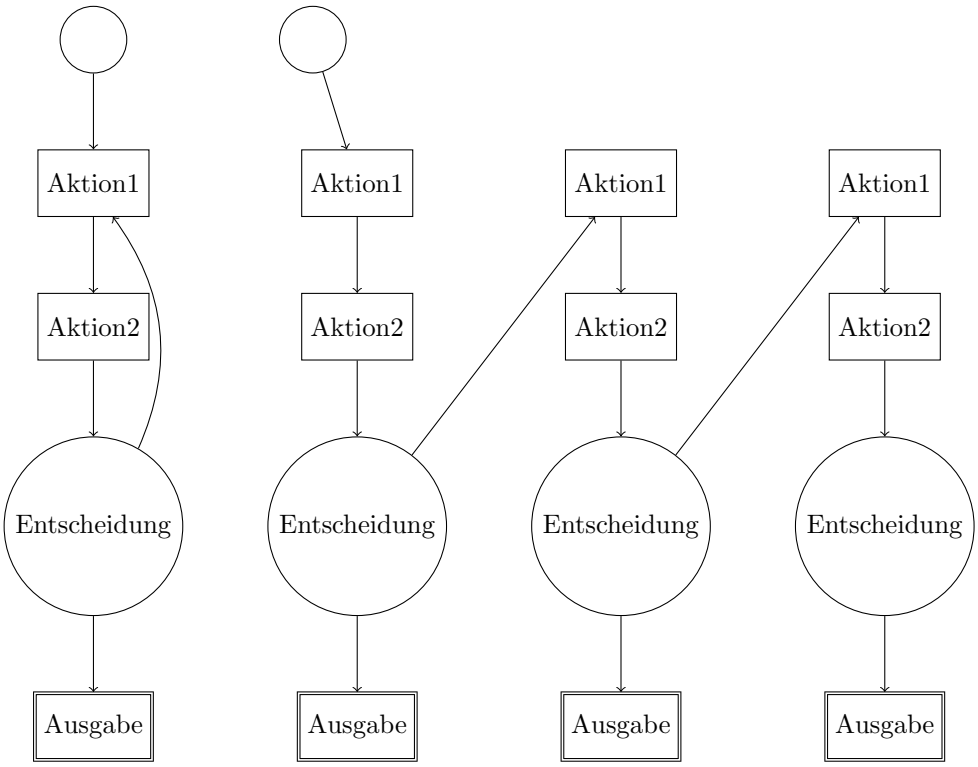

Abbildung TODO Algorithmus Schleifenentfaltung

Auf die Vor- und Nachteile der Implementierung wird im Kapitel 5 eingegangen.

4.2 2. Lösungsansatz

Levenshtein-Distanz für String Wie auch schon beim 1. Lösungsansatz gibt der Benutzer von vorneherein eine Zahl a an, welche die Anzahl an Schleifendurchläufen beschrnänkt. Für die Zahl a gelten die gleichen Bedingungen wie im 1. Lösungsansatz beschrieben. Zusätzlich wird noch eine Zahl TODO benötigt, welche auch der Benutzer angeben muss. TODO soll dabei die Funktionen eines Grenzwertes übernehmen. Bei diesem Ansatz werden mehrere Mittelwerte gebildet und geschaut, wie sich der neu ausgelesene Sensorwert sich im Verhältnis zu den Mittelwerten verhält. Es wird die Differenz zwischen Mittelwert und aktuellen Sensorwert gebildert. Anschließend wird geschaut auf die Differenz größer als TODO ist. Die Mittelwerte bilden wir einmal über alle bisherigen Sensorwerte und einmal über die letzten b Sensorwerte. Dadurch haben wir die Mittelwerte für einen kurzen und längeren zeitraum. Sollte das der Fall sein, wissen wir das die Sensorwerte sich noch nicht stabilisiert haben und wir können den Vorgang wiederholen. Da wir nicht bereits nachdem ersten stabilisierten Wert aufhören wollen, sondern erst wenn der Wert über einen längeren Zeitraum stabil ist, führen wir folgenden n-Chance-Mechanismus ein:

- Sollte der Grenzwert unterschritten werden, wird der Counter um 1 erhöht.
- Sollte der Grenzwert überschritten werden, wird der Counter wieder auf 0 gesetzt.
- Sollte der Counter irgendwann n erreichen, wissen wir das sich die Werte stabilisiert haben und wir davon ausgehen können dass das zu erwartende Ergebniss nicht mehr rauskommt.

5 Evaluation

5.1

5.1.1 1. Lösungsansatz

+einfach zu implementieren, da wir kein schleifenkonstrukt mehr benötigen. +keine Endlosschleife, weil es keine Schleifen gibt +keine Zyklen, weil der Ablauf linear ist +weniger Sprünge, weil keine for oder while Bedingungen vorhanden sind +möglicher Performance gewinn, weil Schleifen-Overhead entfällt -größerer Codeumfang, da der eigentliche schleifenkörper a-mal im code implemtniert werden muss -höherer verbraucht an ressourcen zB Speicher mehr code = mehr speicher -möglicherweise ineffizient, wenn der faktor zu groß gewählt wird -schlechtere Lesbarkeit -wenn bereits nach 3 durchlaufen feststeht, dass das gewünschte ergbeniss nicht mehr erreicht werden kann werden trotzdem die restlichen schritte ausgeführt

6 Literaturverzeichnis

[1] Johnston, W., Hanna, J., & Millar, R. (2004). *Advances in dataflow programming languages*. ACM Computing Surveys, 36(1), 1–34.

[2] Chen, L. (2021). *Iteration vs. Recursion: Two Basic Algorithm Design Methodologies*. SIGACT News, 52(1), 81–86.

[3] Arvind, & Culler, D. (1986). *Dataflow Architectures*. LCS Technical Memos.

[4] Ambler, A., & Burnett, M. (1990). *Visual forms of iteration that preserve single assignment*. Journal of Visual Languages & Computing, 1(2), 159–181.

[5] Mosconi, M., & Porta, M. (2000). *Iteration constructs in data-flow visual programming languages*. Computer Languages, 26(2), 67–104.

[6] Fan, Z., Li, W., Liu, T., Tang, S., Wang, Z., An, X., Ye, X., & Fan, D. (2022). *A Loop Optimization Method for Dataflow Architecture*. In 2022 IEEE 24th Int Conf on High Performance Computing & Communications; 8th Int Conf on Data Science & Systems; 20th Int Conf on Smart City; 8th Int Conf on Dependability in Sensor, Cloud & Big Data Systems & Application (HPCC/DSS/SmartCity/DependSys) (pp. 202–211).

[7] Gévay, G., Soto, J., & Markl, V. (2021). *Handling Iterations in Distributed Dataflow Systems*. ACM Comput. Surv., 54(9), 199:1–199:38.

[8] Alves, T., Marzulo, L., Kundu, S., & França, F. (2021). *Concurrency Analysis in Dynamic Dataflow Graphs*. IEEE Transactions on Emerging Topics in Computing, 9(1), 44–54.

[9] Ye, Z., & Jiao, J. (2024). *Loop Unrolling Based on SLP and Register Pressure Awareness*. In 2024 20th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD) (pp. 1–6).

[10] Lučanin, D., & Fabek, I. (2011). *A visual programming language for drawing and executing flowcharts*. In 2011 Proceedings of the 34th International Convention MIPRO (pp. 1679–1684).

[11] Davis, A., & Keller, R. (1982). *Data Flow Program Graphs*. All HMC Faculty Publications and Research.

[12] Boshernitsan, M., & Downes, M. (2004). *Visual Programming Languages: A Survey*. EECS University of California, Berkeley.

[13] Charntaweekhun, K., & Wangsiripitak, S. (2006). *Visual Programming using Flowchart*. In 2006 International Symposium on Communications and Information Technologies (pp. 1062–1065).

[14] Burnett, M., Baker, M., Bohus, C., Carlson, P., Yang, S., & Van Zee, P. (1995). *Scaling up visual programming languages*. Computer, 28(3), 45–54.

[15] Kurihara, A., Sasaki, A., Wakita, K., & Hosobe, H. (2015). *A Programming Environment for Visual Block-Based Domain-Specific Languages*. Procedia Computer Science, 62, 287–296.

[16] Hils, D. (1992). *Visual languages and computing survey: Data flow visual programming languages*. Journal of Visual Languages & Computing, 3(1), 69–101.

[17] Sousa, T. (2012). *Dataflow Programming Concept, Languages and Applications*. Doctoral Symposium on Informatics Engineering, 7.

[18] Van Deursen, A., Klint, P., & Visser, J. (2000). *Domain-specific languages: an annotated bibliography*. ACM SIGPLAN Notices, 35(6), 26–36.

[19] Roy, G., Kelso, J., & Standing, C. (1998). *Towards a visual programming environment for software development*. In Proceedings. 1998 International Conference Software Engineering: Education and Practice (Cat. No.98EX220) (pp. 381–388). IEEE Comput. Soc.

[20] Weintrop, D. (2019). *Block-based programming in computer science education*. Communications of the ACM, 62(8), 22–25.

[21] Gumm, H.P., & Sommer, M. (2016). *Band 1 Programmierung, Algorithmen und Datenstrukturen*. De Gruyter Oldenbourg.