Contents

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 lääst 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 dargestellt.

Die folgende Zusammenfassung basiert auf der unveröffentlichen Arbeit von Westermann et al.

3.1 Aufbau der Grammatik

3.1.1 Prüfungslogik

Die Regel $< ActivityModel >$ beschreibt die Grundstruktur des Aktivitätsmodell und setzt sich aus $< Acitvity >$ und $< ActivityConnection >$ zusammen. $< Acitvity >$ sind dabei Aktivitäten und kann entweder eine Startmakierung ($< ActivityStart >$), ein Vergleich ($< ActivityCondition >$), eine Aktion ($< ActivityAction >$) oder ein Label ($< ActivityDisplay >$) sein. $< ActivityConnection >$ hingegen definiert, welche Aktivtäten miteinander verbunden sind und setzt sich aus zwei Aktivitäten (ref(Acitvity source) und ref(Acitvity target)) und einer Beschriftung für die Kante ($< stringlabel >$) Eine Aktion kann dabei eine der folgenden Aktionen sein:

- Senden von Hauptuntersuchungs-Adatper-Anfragen (A1) $< ActivityPitaBuildInforRequest >$
- Lesen einer JSON Datei (A2) $< ActivityLoadExternalData >$
- Ausführung einer Datenverarbeitung (A3) $< ActivityFlowCall >$

$< ActivityFlowCall >$ setzt sich aus einem Flow-Template (ref(FlowTemplate)), mehreren Eingaben ($< ActivityPortValue >$ und $< TemplateParameterValue >$) und mehreren Transformationen ($< ValueTranformation >$). Die Transformation beschreibt dabei wie das Ergebnis der Datenverarbeitung weiter genutzt werden soll. Auf die Bedetung des Flow-Templates und der TemplateParameter-Value wird im verlaufe des Kapitel eingegangen $< FlowPortValue >$ setzt sich aus einer Reihe von primtiven Typen ($< FlowPortValue >$) oder einem verweis auf einer Aktion mit einer Transformation ($< ActivityPortValue >$) zusammen. $< ActivityPitaBuildInforRequest >$ uns $< ActivityLoadExternalData >$ setzen sich nur aus Eingaben vom primtiven Typ zusammen, welche für die Ausführung des zwecks notwendig sind zusammen. Der Vergleich kann etweder ein Binärvergleich $< ActivityBinaryCondition >$ oder ein Validierungsvergleich $< ActivityValidityCondition >$ sein. Der Binärvergleich setzt sich dabei aus einem Flow-Template, einem Operator ($<ActivityBinaryConditionOperator>$) und zwei Eingaben ($< ActivityPortValueright >$ und $< ActivityPortValueleft >$) zusammen. Das Label kann sich dabei aus mehreren Textfeldern ($<ActivityDisplayField>$) zusammen. Ein Textfeld besteht dabei aus einer Beschriftung ($<stringlabel>$), einer Farbe ($<stringcolor>$) und einem Verweis auf eine Aktion (ref(ActivityAction))

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

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

⟨*ActivityConnection*⟩ ::= ref(Activity source) ⟨*string label*⟩ ref(Activity target)

⟨*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

3.1.2 Datenverarbeitungs

Die Flow-Instanz (<*FlowInstance*>) kann als Funktion interpretiert werden und bildet eine oder mehrere Eingaben (<*FlowOutputPortlambdaArguments*>) auf eine oder mehrere Ausgaben (<*FlowOutputPortlambdaArguments*>) ab. Zusätzlich kann die Flow-Instanz aus einer oder mehreren Lamda-Definitionen (<*FlowLamda*>) beinhalten. Eine Lamda-Definition hat zusätzliche Eingaben und Ausgaben. Die Ein- und Ausgaben bestehen dabei einem Namen (<*stringname*>) , gefolgt vom Typ (<*Typ*>) und einem boolean (<*boolacceptsError*>) , welcher angibt ob Fehler akzeptiert werden oder nicht.

⟨*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

Ein Flow-Template (<*FlowTemplate*>) kann als abstrakte Oberklasse angesehen werden und besteht dabei aus einer Funktion (<*Flow*>) gefolgt von keiner oder mehreren Template-Parametern (<*TemplateParameter*>), welche Port- und Lambda-Definitionen generien können. Eine Funktion kann dabei eine vom System bereitgestellte (<*LibraryFlow*>) oder eine vom Benutzer selbst definierte (<*FlowModel*>) sein.

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

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

⟨*LibraryFlow*⟩ ::= ϵ

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

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

Grammatik TODO Flow-Template

Die vom System bereitgestellte Funktionen lassen sich dabei in eine von sieben Kategorien unterteilen: Hauptuntersuchungs-Adapter-Antworten, Zeichenkettenverarbeitung, Datum, Vergleichoperatoren, Konverter, Operatoren und Listenverarbeitung.

Das Flow-Model (<*FlowModel*>) ist, wie bereits erwähnt, die vom Benutzer selbst definierten Funktionen und besteht aus einer Flow-Instanz, gefolgt von mehreren möglichen Funktionen (<*FlowNode*>) und Verbindungen (<*FlowConnection*>). Flow-Instanz bestimmt die Ein- und Ausgaben des Flow-Models. kann dabei eine <*FlowNodeOutput*>, <*FlowNodeInput*>, <*FlowNodeLambda*> oder ein <*FlowNodeFlowCall*> sein. <*FlowNodeInput*> besteht aus einem Verweis an einem Verweis an einer Portdefinition, gefolgt von <*FlowPortValue*>. Hingegen <*FlowNodeOutput*> nur aus einer Portdefinition besteht. Die <*FlowConnection*> wird durch zwei Verweise definiert. <*FlowPortValue*> bietet dabei die Möglichekit konstante Werte an die Eingabeports anzulegen.

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

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

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

⟨*FlowNodeInput*⟩ ::= ref(FlowInputPort) ⟨*FlowPortValue*⟩

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

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

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

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

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

Grammatik TODO Flow-Modell

3.1.3 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

3.2 Ausführung

Im folgenden Abschnitt schauen wir uns an, wie die einzelnen Ebenen ausgeführt werden. Die folgende Zusammenfassung basiert auf der unveröffentlichen Arbeit von Westermann et al.

Die Prüfungslogik. Ein wichtiger Bestandteil der Prüfungslogik ist der Referenzstack. Der Referenzstack beeinhaltet alle Erge. Die Aktivitäten können auf den Referenzstack zugreifen und abgespeicherte Ergebnisse referenzieren und diese als Parameter für ihre Aktionen verwenden. Der Startpunkt jeder Prüfung ist die Startaktivität. Die Startaktivität darf pro Prüfung nur einmal vorkommen und ist dafür zuständig, dass der Referenzstack leer ist. Die Reihenfolge der auszuführenden Aktivitäten wird durch die Aktivitäten vorgegeben. Die Aktiväten geben nämlich das Label der nächst zu folgenden Kante zurück. Die Prüfungs ist beendet, sobald die Aktivität kein Label mehr zurückgibt.

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

Schleife soll durch ein Schleifenkonstruktor dargestellt werden. Prüfungslogik muss eine weitere Aktivitätsaktion erweitert werden. Das Schleifenkonstrukt greift dabei auf bereits vorhandene Regeln der Prüfungslogik zu. Das Schleifenkonstrukt greift dabei wie die anderen Aktivitätsaktionen auf den Referenzstack zu. Da der nächste Schleifendurchlauf nicht wieder auf den gleichen Eingabenwerten laufen soll, da diese wieder zu einem fehlerhaften Wert führen wird, muss ein Mechanismus im Schleifenkonstrukt implementiert werden, welcher einen neuen Wert holt. Der Schleifenkörper wird dabei nicht mithilfe von Rekursion oder Iteration ausgeführt, sondern durch entfaltung. Ye et al. beschreiben Schleifenentfaltung als eine gängige Methode um Compiler zu optimieren, weil mit dieser Methode die mehreren Schleifendurchläufe zu einer zusammengefasst werden. [9] Huang et al. beschreiben den ALgortithmus wie folgt TODO. Der beschrieben Ansatz kann für unseren Ansatz nicht 1:1 übernohmen werden, sondern muss etwas modifiziert werden. Unser Ziel ist es nicht nur einzelene Schleifendurchläufe zusammen zu fassen, sondern die ganzen Schleifendurchläufe in einer einzigen zusammenzufassen. Da bei unseren Lösungsansatz die maximale Anzahl an Schleifendurchläufen begrenzt ist und diese bereits vor der Ausführung der Prüfungs bekannt ist, kann diese Information beim modifizierten Ansatz berücksichtigt werden. Ein Beispiel in in Abbildung TODO. Bei dem Beispiel ist die Anzahl der Schleifen Durchläufe auf 3 begrenzt. In beiden Schleifen soll die Zeichenkette "Foo" 3-mal auf der Konsole ausgegeben werden. Der Schleifenkopf initalisiert am Anfang eine Variable. Anschließend wird eine Abbruchbedigung definiert und im Anschluss die veränderung der Variable pro Schleifendurchlauf festgelegt. Im Beispiel 1 wird die Funtkion console.log("Foo") pro Schleifendurchlauf einmal ausgeführt. Hingegen im Beispiel 2 wurde die Schleife entfaltet und die Funktion console.log("Foo") pro Schleifendurchlauf 3-mal ausgeführt. Da die Schleife aber nur noch einmal ausführt und dann abbricht, kann diese auch weggelassen werden. Zwischen den einzelnen Funktionen muss dafür gesorgt werden, dass die neue Wert zur verfgügung steht. Deswegen ist die Idee alle bisherigen Aktivitätaktion zu wiederholen, damit der aktuellste Wert vom Hauptuntersuchungs-Adapter ausgelesen wird und die Prüfung aufgrundlage dieses Wertes nochmals ausgefürt wird. Ein Beispiel ist in Abbildung TODO.

```
/*Beispiel 1*/
for (let i = 0; i <= 2; i++) {
    console.log("foo");
}

/*Beispiel 2*/
for (let i = 0; i <= 0; i++) {
    console.log("foo");
    console.log("foo");
    console.log("foo");
}
```

Abbildung TODO

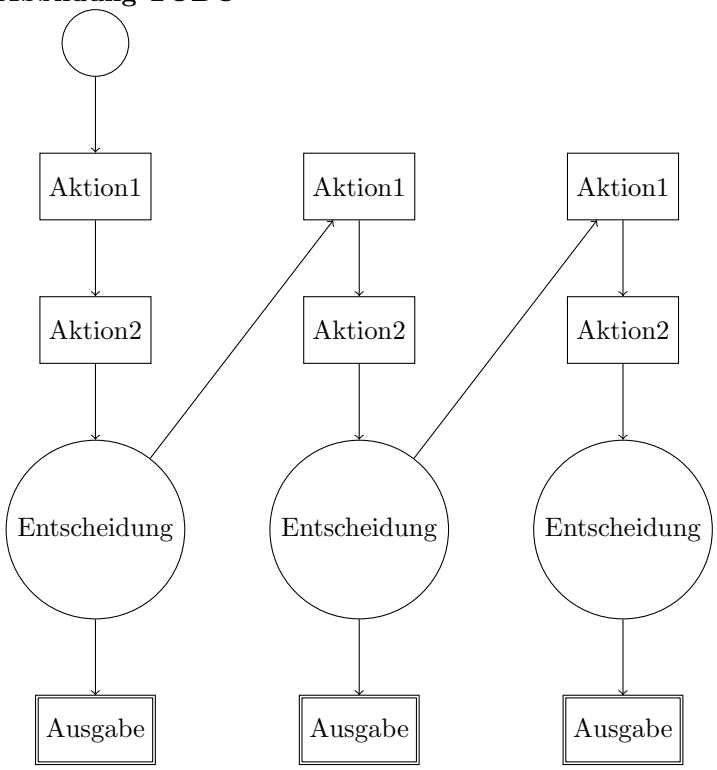

Abbildung TODO

4.2 2. Lösungsansatz

Schleife soll durch ein Konstrukt realisiert werden. Prüfungslogik muss um eine weitere Aktivitätsaktion erweitert werden. Das Schleifenkonstrukt greift dabei auf bereits vorhandene Regeln der Prüfungslogik zu. Das Schleifenkonstrukt greift dabei wie die anderen Aktivitätsaktionen auf den Referenzstack zu. Da der

nächste Schleifendurchlauf nicht wieder auf den gleichen Eingabenwerten laufen soll, da diese wieder zu einem fehlerhaften Wert führen wird, muss ein Mechanismus im Schleifenkonstrukt implementiert werden, welcher einen neuen Wert holt. Durch die einführung der Schleife entstehen neue Herausforderungen. Es können nun Endlosschleifen entstehen, welche dazuführen dass die ausgeführte Prüfung niemals terminieren wird. Außerdem liefert der Hauptuntersuchungs-Adapter keine linearen Werte (?), sondern nicht determenistische Werte.

Eine Endlosschleife kann von vorneherein ausgeschlossen werden, indem die maximalen Schleifendurchläufe begrenzt werden. Da die Werte des Hauptuntersuchungs-Adapter nicht vorhersehbar sind und die Prüfung nicht jedes mal die maximale Anzahl der Schleifendurchläufe ausführen soll, muss ein Algorithmus entwickelt werden, welcher sagt wann man davon ausgehen kann, wann die ausgelesenen Sensorwerte sicht großartig nicht mehr ändern und stabil sind.

Ein möglicher Lösungsvorschlag könnte nun folgendermaßen aussehen. Je nachdem welcher Typ der Eingabewert hat verläuft der Algorithmus anders. Es wird dabei nur zwischen Zahlen und Zeichenketten unterschieden. Bei Zeichenketten wird der aktuelle Wert mit dem Wert aus dem vorherigen Schleifendurchlauf verglichen. Dafür wird die Levenshtein-Distanz verwendet. Für die ersten beiden Schelifendurchläufe wird der Algorithmus übersprungen, weil die Levenshtein-Distanz noch kein Aussagekräftiges Ergebnis für den Anwendungsfall geben kann. Die Levenshtein-Distanz gibt die ähnlichkeit zwischen zwei Zeichenketten als Zahl an, indem sie die minimale Anzahl an Operation angibt, welche benötigt werden, damit die erste Zeichenkette der zweiten Zeichenkette gleicht. Je größer die Zahl ist destso "unterschiedlicher" sind die beiden Zeichenketten von einander.

Um zu schauen wie sich die Eingabe zu verschiedenen Zeiträumen verhält, berechnen wir Mittelwerte über TODO. Es sollten mindestens zwei Mittelwerte gebildet werden. Mehr als zwei Mittelwerte sind möglich, aber würden den Algorithmus entwindlicher machen. Der erste Mittelwert sollte über alle bisherigen Eingaben gebildet werden, um zu sehen wie sich die Eingabe auf langer Sicht verhält. Der zweite Mittelwert sollte über die letzten n Eingaben gebildert, um zu sehen wie sich die Eingabe auf kurzer Sicht verhält. Da die Werte der Levenshtein-Distanz sich für den Mittelwert nicht besonders anbieten, müssen die Zeichenketten in einen Zahlenwert umgewandelt werden.hließend Addieren. Für die Umwandlung eignet sich UTF-8 besonders gut. Da UTF-8 fast akke Schriftzeichen weltweit beinhaltet. Das kann geschaffen werden indem alle Zeichen der Zeichenkette in eine eindeutige Zahl umwandeln und die einzelnen Zahlen ansc Zusätzlich muss eine Gewichtung bei der Addition berücksichtigt werden, weil sonst Zeichenketten, die aus den gleichen Zeichen bestehen, den gleichen Wert bei der Addition rausbekommen. Das liegt daran, dass bei der Addition ohne Gewichtung nur die Wertigkeit der einzelnen Zeichen betrachtet wird, aber nicht deren Position. Dieses Problem wird mit der Gewichtung aufgelöst. Ein Beispiel dafür für die Addition mit Gewichtung ist in Abbildung TODO. Dies muss aber nicht für jedes Eingabepaar gemacht werden, sondern nur für Eingabepaare welche sich sehr ähneln, also eine niedrige Levenshtein-Distanz haben. Für Eingabepaare mit einer hohen Levenshtein-Distanz ist das

nicht notwendig, weil wir da bereits wissen, dass sich die Zeichketten stark von einerander unterscheiden. Ist die Differenz aus der umgewandelten umgewandelten Zeichenkette und einem Mittel kleiner als ein vordefinierter Schwellenwert, wissen wir dass die Zeichenkette sich nur ganz leicht von den durchschnittlichen Eingaben unterscheidet. Wenn dies nun mehrmals nacheinander vorkommt, kann davon ausgegangen werden, dass der Wert in diesen Wertebereich stagniert. Um dies im ALgortithmus auch zu berücksichtigen, wird ein n-Chance Mechanismus eingebaut der folgendermaßen Funktioniert:

- Wird der Schwellenwert unterschritten, wird unser n dekrementiert.
- Wird der Schwellwert übertroffen oder ist unsere Differenz gleich wird n zurückgesetzt.
- Erreicht n irgendwann die 0 wird die Schleife abgebrochen.

$"foo" = 102+111+111 = 324"oof" = 111+111+102 = 324 mit Gewichtung "foo" = 1 * 102 + 2 * 111 + 3 * 111 = 657"oof" = 1 * 111 + 2 * 111 + 3 * 102 = 639$
Abbildung TODO Beispiel Addition mit und ohne Gewichtung

Ist unser Eingabewert nun keine Zeichenkette, sondern eine Zahl entfällt der Umwandlungsschritt mit der Gewichtung. Es kann sofort mit den beschriebenen Mittelwertansatz angefangen werden.

5 Evaluation

5.1 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

5.2 2. Lösungsansatz

+keine Endlosschleife, weil maximale Schleifendurchläufe begrenzt sind. + -azyklisches verhalten wird verletzt, weil schleifenkonstrukt benötigt wird -

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.