Contents

1 Einleitung

1.1 Aufgabe und Motivation

1.2 Aufbau der Arbeit

Die vorliegenede Arbeit setzt sich aus TODO nachfolgenden Kapiteln zusammen. Im zweiten Kapitel TODO Anschließend daran wird im dritten Kapitel TODO Darauf aufbauen wird im vierten und fünften Kapitel TODO Der Fokus im letzten Kapitel liegt auf TODO

2 Terminologie

Der Fokus dieses Kapitels liegt auf der Definition zentraler Fachbegriffe und dadurch einen einheitlichen terminologischen Rahmen zu schaffen. Das Ziel dieses Kapitels ist es die Verständlichkeit der nachfolgenden Kapitel zu erhöhen und die theoretischen Grundlagen der Arbeit zu festigen.

Schleifen

Eine Schleife ist eine Kontrollstruktur, die einen Programm-Abschnitt mehrmals ausführt. [21] Häufig ist dabei die Schleife die zeitintensivste Komponente eines Programms, da ihre die Ausführung sehr viel Zeit in anspruch nehmen kann. [6] Der Algrorithmus dieser Kontrollstruktur kann dabei iterativ oder rekursiv implementiert 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. [2] Die Iteration verwendet dabei einen Akkumulativenansatz. Dabei wird das Problem schrittweise gelöst. Der Prozess wird solange wiederholt bis eine vordefinierte (Abbruch-)Bedingung erfüllt ist. [2] Im gegensatz zur Iteration verwendet die Rekursion keinen Akkumulativenansatz, sondern zerlegt das Problam in mehrere (Teil-)Probleme. Für die Teilprobleme werden dann einzelne Lösungen erarbeitet, welche dann kombiniert werden um das eigentliche Problem zu lösen. [2] Laut Chen L. spiegelt die Iteration das menschliche Denken wieder, weshalb sie sich besonders für lineare Probleme eignet. Die Rekursion hingegen ist für Probleme geeignet, welche Zwischenergebnisse oder Teillösungen benötigen. [2] Eine Schleife kann dabei in zeitabhängig oder horizontal unterteilt werden. Bei einer zeitabhängigen Iteration ist das Ergebniss des aktuellen Schleifendurchlaufs vom Ergebnis des vorherigen Durchlaufs ab. Hingegen bei horizontalen Schleifen die Ergbenisse der einzelnen Schleifendurchläufe unabhängig voneinander sind. [4]

Domain Specific Language

Bei einer domänenspezifische Sprache (DSL) handelt es sich um eine Programmiersprache, die mit dem Ziel entwickelt wurden ist, spezifische Aufgabenstellungen innerhalb eines begrenzten Anwendungskontexts (Domaine) besonders effektiv zu bearbeiten. [18] DSLs bilden das Gegenstück zu General-Purpose Languages wie Java, C++ oder Python. [15] Dabei verfügen DSLs oftmals über eine reduzierte Syntax, die ausschließlich für den jeweilige Domaine relevant ist. Teilweise wird diese durch eine GPL ergänzt. [18] Es wird zwischen externen und internen DSLs unterschieden. 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] Die Vorteile von DLSs liegen in ihrer strukturellen Klarheit und Spezialisierung. Dem gegenüber stehen die Nachteile eines hohen Initialaufwands sowie einer begrenzten Flexibilität und Verfügbarkeit. [18]

Visual Programming Language

Das Hauptziel visueller Programmiersprachen (VPLs) besteht in der Verbesserung der Darstellung der Programmierlogik sowie in der Erleichterung des Verständnisses von Programmabläufen. [14] Dadurch soll der Fokus des Programmierens stärker auf konzeptuelle statt symtaktische Aspekte verlagern. Die syntaktischen Aspekte werden dabei von der Entwicklungsumgebung übernommen. [10] Die Umsetzung von Programmen erfolgt dabei durch die Möglichkeit, Programme in Form von Flussdiagrammen zu erstellen, die unmittelbar vom System interpretiert und ausgeführt werden können. [13] Nach Charntaweekhun bieten Flussdiagramme einen didaktischen Vorteil, da sie es insbesondere Programmieranfängern ermöglichen, komplexe Abläufe visuell zu erfassen und zu strukturieren. [13] ——————————VPLs setzen dabei das Konzept der Visual Programming (VP). [**?**] Im Gegensatz zu GPLs, die eine freie textuelle Eingabe haben, verwenden VPLs nur eine begrenzte Menge an vordefinierten grafischen Elemente. Dadurch wird die Lesbarkeit erhöht und syntaktische Fehler reduziert.[10] Die Klassifikation visueller Programmiersprachen unterscheidet zwischen imperativen und deklarativen Modellen. Ersteres gibt die exakte Reihenfolge der Operationen vor, während letzteres lediglich Datenabhänigkgieten spezifiziert und die Ausführungsreihenfolge dem System überlasst.[12] Die stärken von VPLs liegen in ihrer Einfachheit, visuelle Darstellbarkeit, Transparenz und Interaktivität.[14] Die Mehrheit der VPLs basiert auf einem datenflussgesteuerten Ansatz, bei dem Programme durch den Fluss von Informationen zwischen Operatoren strukturiert werden. [5] Zusammengefasst kann man sagen, dass VPLs die Vorteile von Flussdiagrammen und nicht die Nachteile der klassischen Programmierung kombiniert. [14]

Visual Languae

Visuelle Sprachen (VL) sind Programmiersprachen, bei denen die Informationsdarstellung primär über grafische Elemente und nicht über textuelle Komponenten erfolgt. [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]

Datenfluss-basierte Sprachen

Als Datenfluss-basierte Sprache (DL) wird eine Programmiersprache verstanden, bei der die Daten von zwischen Funktionnen weitergeleitet werden. Dabei werden Programme als Graphen dargestellt. [11] Der zugrunde liegende Programmgraph ist als gerichteter Graph (DG) definiert, wobei Funktionen als Knoten dargestellt werden, die durch gerichtete Kanten miteinander verbunden sind. Die Richtung der Kanten spiegelt die Datenabhängigkeiten wieder.[1] Datenflussgraphen lassen sich hinsichtlich ihrer Granularität in feinkörnig und grobkörnig unterteilen. In einem feinkörnigen Graphen führt jeder Knoten extakt eine Instruktions aus, während grobkörnige Graphen mehrere Instruktionen pro Knoten ausführen können.[6] Zusätzlich lässt sich ein DG basierend auf der Zyklenstruktur in zyklisch und azyklisch unterteilen. [8] DLs sind überwiegend funktional geprägt, aber können auch text-basiert 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] Für komplexe Programmen kann eine reine Graphendarstellung schnell unübersichtlich werden. Zur Struktuierung komplexer Programme werden Mikrofunktionen eingesetzt, bei denen einzelne Knoten auf untergeordnete Teilgraphen verweisen. Dadurch lassen dann auch rekursive Abläufe modellieren. [11] DLs führen Instruktionen nicht in einer festen Sequenz aus. Dadurch können unabhängig voneinander ausführbare Instruktion parallel verarbeitet werden. [6] Diese Form der Ausführung ermöglicht eine Effizienzsteigerung, da der Ablauf nicht mehr durch einen zentralen Programmzähler gesteuert wird. [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]

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] Bei den Ergebnissen erfolgt eine direkte Datenweitergabe zwischen Funktionen, wobei Transformation und Filterung integraler Bestandteil der Verarbeitung sind. [16] DFA-Systeme zeichnen sich durch hohe Effizienz, flexible Strukturen und leistungsstarke Ausführungsmechanismen aus. [6] Innerhalb von DFA-System wird zwischen daten- und bedarfgetriebener Ausführung unterschieden. Beim ersteren werden die Funktionen ausgeführt, sobald die Funktion alle benötigten Operanden hat und ein Signal bekommen hat. Hingegen beim zweiteren die Funktion schon ausgeführt wird, sobald alle Operanden vorhanden sind. [1] Die datengesteuerte Ausführung ist dabei ein Spezialfall der bedarfgesteuerten Ausführung, da hier bereits vorneherein ein Bedarf an allen Ergebnissen besteht. [11] Ein wesentlicher Vorteil datenflussbasierter Systeme besteht im möglichen Parallelismus, da mehrere Instruktionen gleichzeitig ausgeführt werden können, sofern keine Datenabhängigkeiten bestehen. [6]

3 Systemanalyse

In diesem Kapitel wird das System hinsichtlich seiner strukturellen und funktionalen Eigenschaften analysiert. Zunächst wird die zugrundeliegende Grammatik betrachtet, woraufhin die Ausführungslogik beschrieben wird. Abschließend werden die problematischen Stellen im Hinblick auf die geplante Erweiterung analysiert. Ziel dieses Kapitels ist es, diese problematischen Stellen zu erfassen und aufzubereiten, um eine Grundlage für die folgenden Kapitel zu schaffen.

3.1 Grammatik

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.

3.1.1 Prüfungslogik

Die in Abbildung TODO abgebildeten Regeln beschreiben das Aktivitätsmodell. Zentral ist dabei die Regel TODO, welche festlegt, dass ein Aktivitätsmodell aus mehreren Aktivitäten und einer TODO besteht. Die Regel TODO beschreibt die möglichen Aktivitäten innerhalb des Aktivitätsmodells. Dabei kann eine Aktivität entweder eine Startmarkierung, ein Vergleich, eine Aktion oder ein Label sein.

Ein Vergleich kann entweder ein Binärvergleich oder ein Validierungsvergleich sein. Ersteres besteht dabei aus einem verweis auf einen Flowtemplate gefolgt von einem Operator und zwei Argumenten. Als Operatoren stehen $=$ und $\neq$ sowie Relationale Operatoren zur verfügung. Der Validierungsvergleich hingegen besteht nur aus einer Sequenz von TODO.

Bei den Aktionen wird zwischen Hauptunsersuchungs-Adpater Anfragen, lesen von JSON-Dateien und Ausführung einer Datenverarbeitung unterschieden. Der Aufbau einer Hauptuntersuchungs-Adapter Anfrage umfasst dabei den Namen der auszuführenden Anfrage, eine Beschreibung, eine Liste mit anzusprechenden Systemen im Fahrzeug und eine maximale Ausführungsdauer. Im Gegensatz dazu setzt sich das lesen von JSON-Dateien aus dem Datentyp der zu lesenden Datei und einer URI zu der Datenquelle zusammen. Die letzte Aktionform, die Ausführung einer Datenverarbeitung, besteht aus einem Verweis auf

einem FlowTemplate, gefolgt von einer Beschreibung der Eingaben, die für die Datenverarbeitung erforderlich sind, sowie einer Transformation, die beschreibt wie das Ergebnis weiterverwendet werden soll. Die Beschreibung der Eingaben setzt sich aus den Symbolen ActivityPortValue und TemplateParameterValue zusammen.

Die Struktur eines Label wird durch eine Kombination aus einer Beschriftung, einer Farbe und einem Verweis auf eine Aktion beschrieben.

⟨*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 in Abbildung TODO dargestellten Regeln definieren die Struktur einer Flow-Instanz. Dabei legt Regel TODO fest, dass eine Flow-Instanz aus mehreren Eingabe- und Ausgabeports sowie aus Funktionen höherer Ordnung besteht. Der Aufbau einer Funktion höherer Ordnung umfasst wiederum zusätzliche Eingabe- und Ausgabeports. Ein Port besteht aus einem Namen, gefolgt vom Datentyp des Ports und einem Wahrheitswert der Angibt ob an diesem Port Fehler erlaubt sind.

⟨*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 Flowtemplate wird durch die in Abbildung TODO spezifierten Regeln beschrieben. Bei den TemplateParameter wird zwischen einer Zeichenkette, Zahlen oder einem Wahrheitswert unterschieden. Ein Flow kann dabei eine eigene erstelle Funktion sein oder eine vordefinierte Funktion.

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

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

⟨*LibraryFlow*⟩ ::= ϵ

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

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

Grammatik TODO Flow-Template

Die Regeln des FlowModel sind in Abbildung TODO formal dargestellt. Durch die Regel TODO wird bestimmt, dass ein FlowModel aus einer

⟨*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.

Anders ist es hingegen bei der Ausführung der Datenverarbeitung. Dort basiert die Ausführung auf einer Execute-Funktion, welche die Werte der Ausgabeports berechnet. Die Funktion nimmt als Parameter Template-Parameter und eine Evaluate-Funktion. Bei den vom System bereitgestellten Funktionen wird die Execute-Funktion mithilfe der Hilfsklasse vom Typ IExectuibCibtext, direkt in die Funktion implementiert. Die Hilfsklasse stellt die Evaluate-Funktion bereit und speichert die Werte der Ausgabeports. Das Ziel der Implentierung ist es die Abhängigkeiten zwischen Eingabe- und Ausgabeports herzustellen. Untersützt wird sie dabei von der Klasse RunetimeContext.

Ein anderer wichtiger Bestandteil der Datenverarbeitung ist der Ergebniscache. In dem Cache werden die Werte alle Ausgabeports gespeichert. Wird nun nach einem Wert für einen Eingabeport gesucht, wird nach der zugehörigen InputNode mit der anliegden Kante gesucht und geschaut ob für den verbunden Ports bereits ein Ergebnis im Ergebniscache vorliegt. Sollte das nicht der Fall sein, wird die Funktion ausgeführt und das Ergbenis im Ergebniscache gespeichert.

Auch bei der Ausführung von Lambdas spielt der Ergebniscache eine große Rolle. Sobald ein Wertes für ein Eingabeport benötigt werden, werden die Ergebnisse der dazugehörigen Ausgabeports in den Ergebiscache geschrieben und alle Funktionen die im Kind-verhältnis stehen invalidiert, indem die Ergebnisse im Ergebniscache gelöscht werden. Anschließend kann die Berechnung des gesuchten Wertes beginnen, indem der grade beschriebene Algorithmus angewendet wird.

Als erstes werden alle Referenzen auf Flow-Templates aufgelöst, damit wir die Flow-Instanz erhalten. Sobald die Flow-Instanz erstellt ist, ist bekannt, welche Ports und Lambdas bei dem Funktions-Aufruf existieren und es werden Objektreferenzen zwischen den Ports erstellt. Die Objektreferenzen sollen die Modell Analyse erleichtern und beeinhalten Informationen über die Verbindung. Daraufhin werden die Refernz-Typen aufgelöst, indem diese durch konkrete Typen ersetzt werden. Nun kann mit der eigentlichen Validierung angefangen werden. Die Validierung wird pro Flow-Modell ausgeführt und es wird mit dem Flow-Modell angefangen, welches am wenigsten Abhänigkeiten auf andere Flow-Modelle hat. Bei der Prüfung wird über alle Verbindungen von Ports iteriert und falls ein gernerischer Typ vorkommt, wird diese Typ-Zuweisung gespeichert. Im Anschluss werden die Ports des Flow-Modells überprüft und versucht die generischen Typen aufzulösen. Gernerische Typen, welche nicht aufgelöst werden konnten, werden dann beim Flow-Aufruf aufgelöst. Vorausgesetzt die nicht zu auflösenden Typen sind Teil der Argumente und Ergbenisports des Flow-Models. Abschließend werden die Verbindungen von Flow-Ports validiert, indem diese auf Zuweisungskompatiblität überprüft werden, bei den übrig gebliebenen generischen Typen kommt es zu keinen Problemen, weil generische

Typen in beide Richtungen zuweisungskopatibel sind.

Sobald die Prüfungs abgeschlossen ist, kann mit der Port-Fehler Überprüfung begonnen werden. Dafür muss erneut eine Sortierung vorgenommen werden. Es werden zuerst die Abhängigkeiten einer Flow-Node vor der Flow-Node überprüft. Bei der Port-Fehler Überprüfung werden die Eingabeports und deren Verbidnung validiert. Bei der Validierung wird geschaut, ob am dem dazugehörigen Ausgabeport ein Fehler vorliegt. Sollte das der Fall sein und der Eingabeport akzeptiert keine Fehler, dann muss der Fehler propagiert werden. In dem Fall würde beim Eingabeport ein Fehler auftreten und der Flow-Node würde nicht ausgeführt werden. Außerdem würde der Fehler an die dazugehörigen Ausgabeports weitergegebn. Um dies zu verhindern TODO.

Im Anschluss können dann die Nodes validiert werden. Bei der Validierung wird geprüft ob das Argument $<boolacceptsError>$ der $<FlowNodeInput>$ den gleichen Wert wie der Refenzierte $<FlowInputPort>$ hat. Ist das nicht der Fall wird eine Fehlermeldung für das Flow-Modell ausgegeben.

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

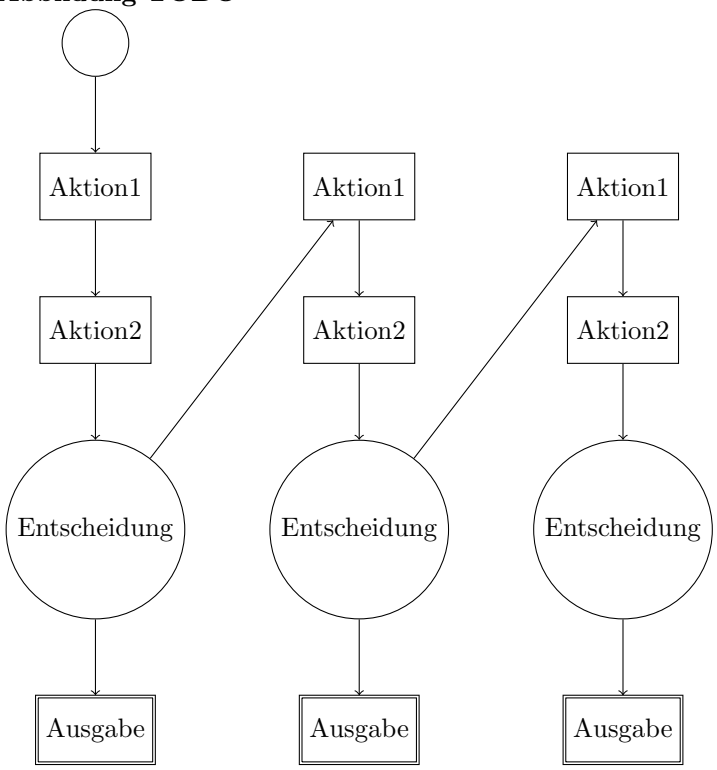

Abbildung TODO

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.