

Contents

1	Einleitung	2
1.1	Aufgabe und Motivation	2
1.2	Aufbau der Arbeit	2
2	Terminologie	3
2.1	Schleifen	3
2.2	Domain Specific Language	3
2.3	Visual Programming Language	4
2.4	Visual Language	4
2.5	Datenfluss-basierte Sprachen	4
2.6	Datenfluss-basierte Systeme	5
3	Aufbau der domainspezifischen Sprache	6
3.1	Aufbau der Grammatik	6
3.1.1	Prüfungslogik	6
3.1.2	Datenverarbeitungs	9
3.1.3	Typsystem	10
3.2	Ausführung	11
4	Implementierung	13
4.1	1. Lösungsansatz	13
4.2	2. Lösungsansatz	14
5	Evaluation	17
5.1	1. Lösungsansatz	17
5.2	2. Lösungsansatz	17
6	Literaturverzeichnis	18

1 Einleitung

1.1 Aufgabe und Motivation

1.2 Aufbau der Arbeit

Die vorliegende 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

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 Algorithmus 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. 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 Problem auf mehrere (Teil-)Probleme. Für die Teilprobleme werden dann einzelne Lösungen gesucht, welche anschließend zusammengesetzt 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 horizontaln 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 Anwendungsbereich entwickelt wurde und normalerweise auch auf diesen beschränkt ist. Das Ziel einer DSL ist es in dem klar abgegrenzten 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 zeitgich 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 einen eingeschränkten 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 Algorithmus 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 Charntaweehun eignen sich Flussdiagramme sehr gut zum Programmieren, weil vor allem viele Einsteiger in der Programmierung erstmal Flussdiagramme erstellen um das Problem zu visualisieren. [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 Abhängigkeit zwischen Daten vorgegeben und das System bestimmt die Reihenfolge selbst. [12] VPLs haben den Vorteil, dass diese einfach, visuell darstellbar sind, transparent und Interaktiv sind. Einfachheit, weil weniger Programmierkonzepte zum Programmieren benötigt werden. Visuell darstellbar, weil Transparent, weil Datenabhängigkeiten anschaulich dargestellt werden. Interaktiv, weil der Entwickler direkt Feedback bekommen kann. [14] Bei VPL ist das meist genutzte 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 ausdrückstärker als Worte sind und haben einen höheren Wiedererkennungswert. Durch die eingeschränkte Syntax sind VLs nicht so flexibel und ausdrückstark 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 gerichtete Pfeile miteinander verbunden werden. Dabei beschreiben die Pfeile die Datenabhängigkeiten 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 auf einmal 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 zu 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 Algorithmus. 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 parallel ausgeführt werden können.[6] Durch diese Ausführung kann in den meisten ein Effizienzsteigerung geschaffen werden, weil das Programm nicht mehr vom Programmzähler abhängig ist. [1] Johnston et. al beschreiben in ihrer wissenschaftlichen Arbeit 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 aufgeführt werden, sobald die Operanden zur Verfügung stehen. [8] Die Vorteile einer DFA sind, dass diese hohe Performance, Flexibilität und hohe Effektivität fördert. [6] Die Ausführung kann dabei datengesteuert oder bedarfsgesteuert sein. Bei einer bedarfsgesteuerten Ausführung 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 gleichzeitig ausgeführt werden kann, da datenabhängigkeiten 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 der Bedarf an allen Ergebnissen vorhanden ist.[11] Bei der Ausführung fließen die Ergebnisse einer Funktion direkt in eine andere und werden dort transformiert oder gefiltert. [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 $\langle symbol \rangle$ sind nichtterminal $::=$ bedeutet dass $symbol$ durch $_expression_$ ersetzt wird $_expression_$ ist eine sequenze von nichtterminalen und terminale Kleene-Stern * wiederholung Alternation | oder Sequenz erlaubt auch Klammern um die Reihenfolge der Regel zu definieren Softwareprüfung lässt sich visuell von zwei seiten betrachten.

$\langle symbol \rangle ::= _expression_$

Grammatik TODO Backus-Naur-Form

Grammatik lässt sich in 3 Ebenen unterteilen Prüfungslogik, Datenverarbeitung und Typsystem Prüfungslogik führt Entscheidung im Prüfungsablauf aus und bestimmt die Reihenfolge der Aktionen. außerdem datenerfassung Datenverarbeitung ist für die Datentransformation auswertung zuständig. Also Funktionen, welche keine Nebeneffekte besitzen, weil sie unabhängig von der restlichen Softwareprüfung stattfinden. Typsystem ermöglicht die statische analyse der ausführbarkeit Softwareprüfung lässt sich visuell von zwei seiten betrachten. Einmal als Datenflussgraphen, indem Teil-Funktionen als Blöcke dargestellt werden und Funktionsparameter/Ergebnisse als Ports. Einmal als Aktivitätsdiagramm, in dem nur Startzustand, Endzustände, Aktions- und Entscheidungsblöcke dargestellt.

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

3.1 Aufbau der Grammatik

3.1.1 Prüfungslogik

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

- Senden von Hauptuntersuchungs-Adapter-Anfragen (A1) $\langle ActivityPitaBuildInforRequest \rangle$
- Lesen einer JSON Datei (A2) $\langle ActivityLoadExternalData \rangle$
- Ausführung einer Datenverarbeitung (A3) $\langle ActivityFlowCall \rangle$

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

$\langle \text{ActivityModel} \rangle ::= \langle \text{Activity} \rangle^* \langle \text{ActivityConnection} \rangle$

$\langle \text{Activity} \rangle ::= \langle \text{ActivityStart} \rangle \mid \langle \text{ActivityAction} \rangle \mid \langle \text{ActivityCondition} \rangle \mid \langle \text{ActivityDisplay} \rangle$

$\langle \text{ActivityConnection} \rangle ::= \text{ref}(\text{Activity source}) \langle \text{string label} \rangle \text{ref}(\text{Activity target})$

$\langle \text{ActivityStart} \rangle ::= \epsilon$

$\langle \text{ActivityAction} \rangle ::= \langle \text{ActivityFlowCall} \rangle \mid \langle \text{ActivityPitaBuildInforRequest} \rangle \mid \langle \text{ActivityLoadExternalData} \rangle$

$\langle \text{ActivityFlowCall} \rangle ::= \text{ref}(\text{FlowTemplate}) \langle \text{ActivityPortValue} \rangle^* \langle \text{TemplateParameterValue} \rangle^* \langle \text{ValueTransformation} \rangle^*$

$\langle \text{ActivityPitaBuildInforRequest} \rangle ::= \langle \text{string abdFilename} \rangle \langle \text{string requestAlias} \rangle \langle \text{string expectedSystems} \rangle^* \langle \text{number timeout} \rangle$

$\langle \text{ActivityLoadExternalData} \rangle ::= \langle \text{Type dataType} \rangle \langle \text{string dataSource} \rangle$

$\langle \text{ActivityPortValue} \rangle ::= \langle \text{FlowPortValue} \rangle \mid \langle \text{ActivityPortRefernce} \rangle$

$\langle \text{FlowPortValue} \rangle ::= \langle \text{string} \rangle \mid \langle \text{number} \rangle \mid \langle \text{bool} \rangle \mid \langle \text{date} \rangle \mid \langle \text{FlowPortValue} \rangle^*$

$\langle \text{ActivityPortRefernce} \rangle ::= \text{ref}(\text{ActivityAction}) \langle \text{ValueTransformation} \rangle^*$

$\langle \text{ValueTransformation} \rangle ::= \langle \text{string objectReference} \rangle \mid \langle \text{number listIndex} \rangle$

$\langle \text{ActivityCondition} \rangle ::= \langle \text{ActivityBinaryCondition} \rangle \mid \langle \text{ActivityValidityCondition} \rangle$

$\langle \textit{ActivityBinaryCondition} \rangle ::= \text{ref}(\text{FlowTemplate}) \langle \textit{ActivityBinaryConditionOperator} \rangle$
 $\langle \textit{ActivityPortValue left} \rangle \langle \textit{ActivityPortValue right} \rangle$

$\langle \textit{ActivityValidityCondition} \rangle ::= \langle \textit{ActivityPortValue} \rangle^*$

$\langle \textit{ActivityBinaryCondition} \rangle ::= '=' \mid '\neq' \mid '<' \mid '\leq' \mid '>' \mid '\geq'$

$\langle \textit{ActivityDisplay} \rangle ::= \langle \textit{ActivityDisplayField} \rangle^*$

$\langle \textit{ActivityDisplayField} \rangle ::= \langle \textit{string label} \rangle \langle \textit{string color} \rangle \text{ref}(\text{ActivityAction})$

Grammatik TODO Aktivitätsmodell

3.1.2 Datenverarbeitungs

Die Flow-Instanz ($\langle FlowInstance \rangle$) kann als Funktion interpretiert werden und bildet eine oder mehrere Eingaben ($\langle FlowOutputPort lambdaArguments \rangle$) auf eine oder mehrere Ausgaben ($\langle FlowOutputPort lambdaArguments \rangle$) ab. Zusätzlich kann die Flow-Instanz aus einer oder mehreren Lambda-Definitionen ($\langle FlowLambda \rangle$) beinhalten. Eine Lambda-Definition hat zusätzliche Eingaben und Ausgaben. Die Ein- und Ausgaben bestehen dabei aus einem Namen ($\langle stringname \rangle$), gefolgt vom Typ ($\langle Type \rangle$) und einem boolean ($\langle bool acceptsError \rangle$), welcher angibt ob Fehler akzeptiert werden oder nicht.

$$\langle FlowInstance \rangle ::= \langle FlowOutputPort lambdaArguments \rangle^* \langle FlowInputPort lambdaArguments \rangle^* \langle FlowLambda \rangle^*$$

$$\langle FlowLambda \rangle ::= \langle FlowOutputPort lambdaArguments \rangle^* \langle FlowInputPort lambdaArguments \rangle^*$$

$$\langle FlowInputPort \rangle ::= \langle string name \rangle \langle Type \rangle \langle bool acceptsError \rangle$$

$$\langle FlowOutputPort \rangle ::= \langle string name \rangle \langle Type \rangle \langle bool producesError \rangle$$

Grammatik TODO Flow-Instanz

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

$$\langle FlowTemplate \rangle ::= \langle Flow \rangle \langle TemplateParameter \rangle^*$$

$$\langle Flow \rangle ::= \langle LibraryFlow \rangle \mid \langle FlowModel \rangle$$

$$\langle LibraryFlow \rangle ::= \epsilon$$

$$\langle TemplateParameter \rangle ::= 'String' \mid 'Number' \mid 'Bool' \mid \langle TemplateParameterList \rangle$$

$$\langle TemplateParameterList \rangle ::= \langle TemplateParameter \rangle$$

Grammatik TODO Flow-Template

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

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

$$\langle FlowModel \rangle ::= \langle FlowInstance \rangle \langle FlowNode \rangle^* \langle FlowConnection \rangle^*$$

$$\langle FlowNode \rangle ::= \langle FlowNodeOutput \rangle \mid \langle FlowNodeInput \rangle \mid \langle FlowNodeLambda \rangle \mid \langle FlowNodeFlowCall \rangle$$

$$\langle FlowNodeOutput \rangle ::= \text{ref}(\text{FlowOutputPort})$$

$$\langle FlowNodeInput \rangle ::= \text{ref}(\text{FlowInputPort}) \langle FlowPortValue \rangle$$

$$\langle FlowNodeLambda \rangle ::= \text{ref}(\text{FlowLambda}) \langle FlowPortValue \rangle^*$$

$$\langle FlowNodeFlowCall \rangle ::= \text{ref}(\text{FlowTemplate}) \langle FlowPortValue \rangle^* \langle TemplateParameterValue \rangle^*$$

$$\langle FlowConnection \rangle ::= \text{ref}(\text{FlowOutputPort source}) \text{ref}(\text{FlowOutputPort target})$$

$$\langle TemplateParameterValue \rangle ::= \langle string \rangle \mid \langle number \rangle \mid \langle bool \rangle \mid \langle TemplateParameterValueList \rangle$$

$$\langle TemplateParameterValueList \rangle ::= \langle TemplateParameterValue \rangle^*$$

Grammatik TODO Flow-Modell

3.1.3 Typsystem

unterstützt die gleichen Primitiv-Typen wie JSON-Format String, Number und Bool zusätzlich Date und PtiaResponse. Außerdem werden auch generische Typen unterstützt, 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

$$\langle Type \rangle ::= \langle TypePrimitive \rangle \mid \langle TypeOptional \rangle \mid \langle TypeList \rangle \mid \langle TypeObject \rangle$$

$$\langle TypePrimitive \rangle ::= \text{'String'} \mid \text{'Number'} \mid \text{'Bool'} \mid \text{'Data'} \mid \text{'PtiaResponse'}$$

$$\langle TypeOptional \rangle ::= \langle Type \rangle \text{'?'}$$

$$\langle TypeList \rangle ::= \langle Type \rangle \text{'[]'}$$

$\langle TypeObject \rangle ::= \text{'{' } (\langle string\ key \rangle \text{' : ' } \langle Type \rangle)^* \text{' } \text{'}'}$

$\langle TypeGeneric \rangle ::= \text{'$' } \langle string\ genericName \rangle$

$\langle TypeReference \rangle ::= \text{ref}(\text{Type})$

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 beinhaltet 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 Aktivitäten 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 `IEvaluateCibtext`, direkt in die Funktion implementiert. Die Hilfsklasse stellt die Evaluate-Funktion bereit und speichert die Werte der Ausgabeports. Das Ziel der Implementierung ist es die Abhängigkeiten zwischen Eingabe- und Ausgabeports herzustellen. Unterstützt wird sie dabei von der Klasse `RuntimeContext`.

Ein anderer wichtiger Bestandteil der Datenverarbeitung ist der Ergebniscache. In dem Cache werden die Werte aller Ausgabeports gespeichert. Wird nun nach einem Wert für einen Eingabeport gesucht, wird nach der zugehörigen `InputNode` mit der anliegenden Kante gesucht und geschaut ob für den verbundenen Ports bereits ein Ergebnis im Ergebniscache vorliegt. Sollte das nicht der Fall sein, wird die Funktion ausgeführt und das Ergebnis 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 Ergebniscache 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 beinhalten Informationen über die Verbindung. Daraufhin werden die Referenz-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 Abhängigkeiten auf andere Flow-Modelle hat. Bei der Prüfung wird über alle Verbindungen von Ports iteriert und falls ein generischer Typ vorkommt, wird diese Typ-Zuweisung gespeichert. Im Anschluss werden die Ports des Flow-Modells überprüft und versucht die generischen Typen aufzulösen. Generische Typen, welche nicht aufgelöst werden konnten, werden dann beim Flow-Aufruf aufgelöst. Vorausgesetzt die nicht zu auflösenden Typen sind Teil der Argumente und Ergebnisports des Flow-Modells. Abschließend werden die Verbindungen von Flow-Ports validiert, indem diese auf Zuweisungskompatibilität überprüft werden, bei den übrig gebliebenen generischen Typen kommt es zu keinen Problemen, weil generische Typen in beide Richtungen zuweisungskompatibel 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 Verbindung 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 weitergegeben. 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 Referenzierte *<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 Sensorwertes ist bereits möglich. Aktuell unterstützt die zugrundeliegende 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 Eingabewerten 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 ALgorithmus wie folgt TODO. Der beschriebene Ansatz kann für unseren Ansatz nicht 1:1 übernommen werden, sondern muss etwas modifiziert werden. Unser Ziel ist es nicht nur einzelne Schleifendurchläufe zusammen zu fassen, sondern die ganzen Schleifendurchläufe in einer einzigen zusammenzufassen. Da bei unserem Lösungsansatz die maximale Anzahl an Schleifendurchläufen begrenzt ist und diese bereits vor der Ausführung der Prüfungslogik bekannt ist, kann diese Information beim modifizierten Ansatz berücksichtigt werden. Ein Beispiel 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 initialisiert am Anfang eine Variable. Anschließend wird eine Abbruchbedingung definiert und im Anschluss die Veränderung der Variable pro Schleifendurchlauf festgelegt. Im Beispiel 1 wird die Funktion `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 Verfügung steht. Deswegen ist die Idee alle bisherigen Aktivitätsaktionen zu wiederholen, damit der aktuellste Wert vom Hauptuntersuchungs-Adapter ausgelesen wird und die Prüfung auf Grundlage dieses Wertes nochmals ausgeführt wird. Ein Beispiel ist in Abbildung TODO.

```

1  /*Beispiel 1*/
2  for (let i = 0; i <= 2; i++) {
3      console.log("foo");
4  }
5
6  /*Beispiel 2*/
7  for (let i = 0; i <= 0; i++) {
8      console.log("foo");
9      console.log("foo");
10     console.log("foo");
11 }

```

Abbildung TODO

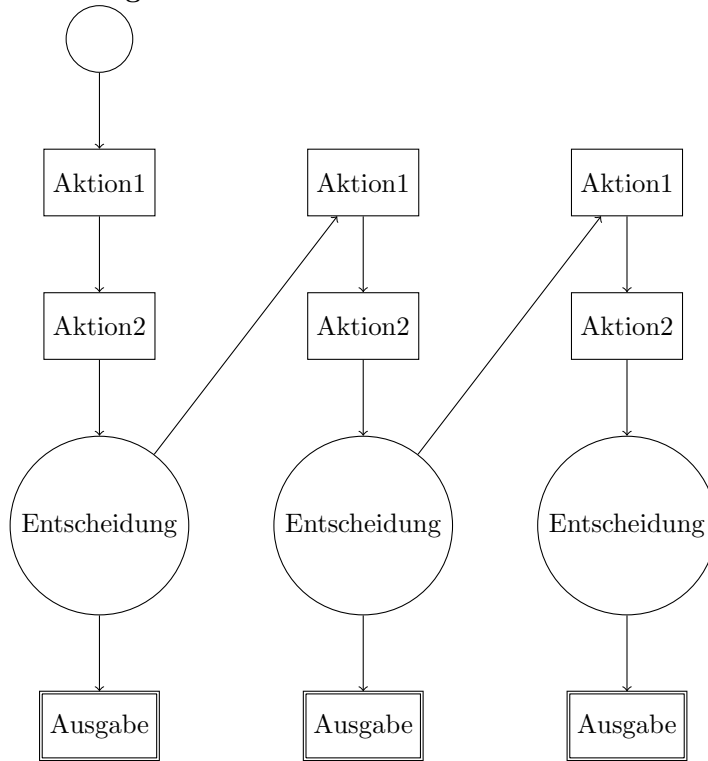


Abbildung TODO

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 deterministische 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 sich 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 Schleifendurchläufe 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 desto "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 gebildet, 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. Hießend Addieren. Für die Umwandlung eignet sich UTF-8 besonders gut. Da UTF-8 fast alle Schriftzeichen weltweit beinhaltet. Das kann geschaffen werden indem alle Zeichen der Zeichenkette in eine eindeutige Zahl umwandeln und die einzelnen Zahlen anschließend 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*mitGewichtung*"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] Lućanin, 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*.