Contents

1 Einleitung

In diesem Kapitel wird in kürze erläutert was die Motivation und Aufgabe der Bachelorarbeit ist. Im Anschluss daran werden sich andere Systeme angeschaut und erklärt, wie diese das Problem lösen. Anschließend wird ein kurzen Überblick über die kommenden Kapitel gegeben.

1.1 Aufgabe und Motivation

Prüfungen, die mithilfe grafischer Editoren modelliert werden, müssen in vielen Fällen eine Vielzahl von wiederkehrenden Aufgaben abbilden. Diese Wiederholungen können bei der aktuellen Implementierung nur schwer abgebildet werden, da Schleifen als Konstrukte nicht zur Verfügung stehen. Stattdessen müssen zu wiederholende Abläufe mehrfach und explizit modelliert werden. Dies führt zu Redundanz, erhöht die Komplexität der Modelle und erschwert deren Wartbarkeit.

In klassischen Programmiersprachen wird das Problem der Wiederholung durch die Verwendung von Schleifenkonstrukten gelöst, welche eine kompakte und dynamische Modellierung ermöglichen. In grafischen Editoren ohne native Unterstützung von Schleifen ist es hingegen notwendig, Prüfungen statisch und mehrfach abzubilden. Eine pragmatische Lösung besteht darin, Prüfungen mehrfach auszuführen oder die zu wiederholenden Abläufe manuell zu duplizieren und diese mit bedingten Verzweigungen zu Verbinden. Dies ist jedoch oft nicht praktikabel, da Prüfungen häufig an variable oder unvorhersehbare Bedingungen angepasst werden müssen. Die einzige Möglichkeit, um dynamische Daten während einer Prüfung effizient und wartbar zu bearbeiten, besteht darin, ein Schleifenkonstrukt in das bestehende System zu integrieren. Daraus ergibt sich ein klares Interesse, entsprechende Erweiterungen zu konzipieren und zu implementieren.

Im laufe der Bachelorarbeit sollen mehrere Möglichkeiten konzepiert und implementiert werden, welche es ermöglichen sollen dynamsichen Daten innerhalb einer Prüfung zu verarbeiten. Dafür muss im Vorfeld das zugrunde liegende System analysiert und bearbeitet werden, sodass Zyklen vom System verarbeitet werden können. Nach der Implementierung sollen diese Bewertet werden.

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 um 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 diese dabei 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. Ersteres wiederholt die Schleife mehrmals. Hingegen die Rekursion sich mehrmals selbst aufruft. [2] Zur umsetzung verwendet die Iteration 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 im Anschluss wieder 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 Schleife hängt das Ergebnis des aktuellen Schleifendurchlaufs vom Ergebnis des vorherigen Schleifendurchlaufs ab. Andererseits sind die Ergebnisse der Schleifendurchläufe bei der horizontalen Schleife unabhängig voneinander. [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 lösen. [18] DSLs bilden das Gegenstück zu General-Purpose Languages (GPL) wie Java, C++ oder Python. [15] Anders als bei GPLs 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

Visuelle Sprachen (VL) sind Sprachen, 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]

Eine spezielle Form visueller Sprachen stellen die visuellen Programmiersprachen (VPL) dar, bei denen grafische Darstellungen gezielt für die Erstellung von Programmen genutzt werden. Das Hauptziel VPL 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 verlagert werden. Die syntaktischen Aspekte werden von der Entwicklungsumgebung übernommen. [10] Die Umsetzung von Programmen erfolgt 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] Im Gegensatz zu GPLs, die eine freie textuelle Eingabe haben, verwenden VPLs nur eine begrenzte Menge an vordefinierten grafischen Elemente. Infolgedessen 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] Visuelle Programmiersprachen (VPLs) basieren überwiegend auf einem datenflussgesteuerten Modell, bei dem die Strukturierung von Programmen durch den Austausch von Informationen zwischen Operatoren erfolgt. [5] Zusammengefasst kann man sagen, dass VPLs die Vorteile von Flussdiagrammen und nicht die Nachteile der klassischen Programmierung kombiniert. [14]

Datenfluss-basierte Sprachen

Als Datenfluss-basierte Sprache (DL) wird eine Programmiersprache verstanden, bei der die Daten zwischen Funktionnen weitergeleitet werden. Die Programme werden dabei als Graphen dargestellt. [11] Der zugrunde liegende Programm-

graph 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] Neben der Granularität lässt sich ein Datenflussgraph auch in Zyklenstrukturen unterteilen. Dabei wird zwischen zyklisch und azyklisch unterschieden. [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]

Eine Computerarchitketur, die auf DLs basiert, wird Datenfluss-basiertea System (DFA) genannt. DFS wurde eingeführt um den Flaschenhals der von-Neumann-Architektur zu vermeiden. [8]

Je nach Implementierung kann nur lokaler Speicher verwendet werden. [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 eine hohe Effizienz, flexible Strukturen und leistungsstarke Ausführungsmechanismen aus. [6] Ein weiterer Vorteil besteht im möglichen Paralellismus. Da bei fehlender Datenabhängigkeit mehrere Insturktionen gleichzeitig ausgeführt werden können. [6] Innerhalb von DFA-System wird zwischen daten- und bedarfgetriebener Ausführung unterschieden. Ersteres führt die Funktionen aus, sobald alle benötigten Operanden vorhanden sind und ein Signal vorliegt. Hingegen bei der bedarfgetriebenen Ausführung die Funktion bereits ausgeführt wird, sobald die Operanden vorhanden sind. [1] Aus dem Grund kann die datengesteuerte Ausführung als Spezialfall einer bedarfsgesteuerten Ausführung angesehen werden, bei der ein Bedarf an allen Ergebnissen von vorneherein besteht. [11]

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 FlowInstance, einer Auflistung alle genutzen Flow-Node und definition der Verbindungen zwischen den Eingabe- und Ausgabeports. Bei den Flow-Node wird zwischen verweis auf Eingabeports, verweis auf Lambda-Funktionen, verweis auf Funktion mit Auflistung von Parametern oder verweis auf Ausgabeports unterschieden. Bis auf das letztere haben alle Flow-Node die Möglichkeit ihre Eingabeports durch konstante Werte zu spezifizieren.

⟨*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.

Prüfungslogik

Die Prüfungslogik stellt die oberste Ebene einer Prüfung dar und wird als Kontrollfluss modelliert. Als Startpunkt jeder Prüfung fungiert die Startaktivität, die pro Prüfung nur einmal vorkommt. Die Reihenfolge der auszuführenden Aktivitäten wird durch die grade ausgeführte Aktivitäte vorgegeben, da jede Aktivität das Label der zu folgende Kante zurückgibt. Eine Prüfung ist beendet, sobald die auszuführende Aktivität kein Label mehr zurückgibt.

Ein wichtiger Bestandteil der Prüfungslogik ist der Referenzstack. Der Referenzspeicher dient als Speicher für die Ergebnisse der Aktivitäten. Alle Aktivitäten können auf den Referenzstack zugreifen und auf die gespeicherten Ergebnisse referenzieren, um diese als Parameter für ihre Funktionen zu verwenden.

Datenverarbeitung

Die Ausführung der Datenverarbeitung erfolgt auf Grundlage einer Execute-Funktion, die für die Berechnung der Ausgabeport zuständig ist. Diese Funktion nimmt als Parameter eine Evaluate-Funktion und Template-Parameter. Bei Funktionen, die vom System bereitgestellt werden, wird die Execute-Funktion von der Hilfeklasse IExectuibCibtext bereitgestellt und die Execute-Funktion wird direkt in die FUnktion implentiert.

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 wichtiger Bestandteil der Datenverarbeitung ist der Ergebniscache. Der Ergebniscache speichert die Werte der Ausgabeports. Wird ein Wert für einen Eingabeport gesucht, erfolgt zunächst die Suche nach dem zugehörigem InputNode sowie der entsprechenden eingehen Kante. Anschließend wird geprüft, ob für den mit der Kante verbundene Ports ein Wert im Ergebniscache vorleigt. Ist dies nicht der Fall sein, wird die Funktion ausgeführt und der Wert für den Ausgabeport im Ergebniscache gespeichert. Nicht nur für die normalen Funktion spielt der Ergebniscache eine große Rolle, sondern auch für die Lambda-Funktionen.

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.

3.3 Codeanalyse

Die aktuelle Umsetzung des Codes ermöglicht es aktuell nicht die geplante. Konkret lassen sich zwei Probleme aus der aktuellen Umsetzung ableiten:

- P1 Die Modellanalyse erlaubt keine Zyklen im Graphen

- P2 Die Ausführungsumgebung geht davon aus, dass Zyklen nicht erlaubt sind

P1 - Die Modellanalyse erlaubt keine Zylken

Aktuell wird in der Klasse ActivityCycleCheckResolver überprüft, ob im Graphen Zyklen vorhanden sind. Die Überprüfung erfolgt dabei mithilfe des TODO Algorithmus. Der Algorithmus markiert als erstes in einem Dictonairy alle Knoten des Graphen als nicht besucht. Anschließend wird eine tiefensuche über alle Knoten gemacht und jeder Knoten wird in eine Liste hinzugefügt. Dabei werden die Nachfolgenden Knoten als erstes der Liste hinzugefügt. Es erfolgt also ein topologisches Sortieren nach Post order. Auf der topolischen sortieren Liste wird eine erneute Tiefensuche durchgefügrt, die für jeden erreichbaren Knoten einen Komponent mit der gemeinsamen Wurzel (root) erstellt. Dabei werden Komponents als erstes für alle Vorgänger Knoten erstellt. Nun wurden alle zusammenhängende Elemente des Graphens gefunden und die zusammenhängende Elemente können in einem Dictonairy übertragen werden. Dabei wird der root der Schlüssel eines Dictonairy Elements sein und der Value der Knoten. Hat ein Dictonairy Element mehr als ein Element im Value ist ein Zyklus vorhanden und diese Dictonairy Element wird in einem anderen Dictonairy gespeichert, welches zum erstellen von Fehlermeldungen verwendet wird

Hier liegt auch das Problem. Wenn ein Zyklus erlaubt ist, darf das Dictonairy Element nicht im das Dictonairy für die Fehlermeldung gespeichert werden, sondern dieser Schritt muss übersprunen werden.

P2 - Die Ausführungsumgebung geht davon aus, dass Zyklen nicht erlaubt sind

In der Klasse VirtualMachineCompilerActivity wird das Modell in eine kompilierte Darstellung überführt. Die Überführung funktioniert in der aktuellen Umsetzung so, dass eine Funktion erstellt wird und alle Knoten des Graphens nacheinander hinzugefügt wird. Bei einer Verzweigung wird eine extra Funktion erstellt, die ab der dem Punkt automatisch TODO. Bei einer Vereinigung werden die Funktionen wieder zu einer gemergt. Es wird aktuell nicht erkannt, ob der Knoten bereits hinzugefügt wurden ist.

Hier liegt auch das Problem. Wenn Zyklen nun erlaubt sind, würde eine Endlosschleife entstehen, weil der Algorithmus den bereits hinzugefügten Teilgraphen erneut hinzufügen. Es muss also eine Möglichkeit geschaffen werden, dass mehrere TODO

4 TODO

4.1 Anforderungsphase

Die geplante Erweiterung sieht die Einführung von zwei Schleifenblock vor, die die Wiederholung von Aktivitäten ermöglichen sollen. Dadurch soll es möglich sein Prüfungen mit dynamsichen Daten zu erstellen. Die Interaktion erfolgt über die bereits vorhandene grafische Benutzeroberfläche, in dieser kann der Benutzer die Anzahl der maximalen Iterationen und die Abbruchbedingung festlegen. Beide Blöcke sind von der Benutzeroberfläche gleiche und unterscheiden sich nur in der Logik. Dabei soll der Schleifenblock dem TODO ähneln. Der Schleifenblock hat einen onFalse und einen onTrue Pfad, welcher Pfad genommen wird wird durch die Abburchbedingung bestimmt. Bei der Ausführung entscheidet der onFalse Zweig über eine mögliche Wiederholung. Die Entscheidung vom Block A basiert dabei auf die Anzahl der bereits ausgeführten Iteration. Hingegen Block B zusätzlich die Eingaben als Kriterium miteinbezieht. Der onTrue-Zweig führt zur beendigung des Schleifenblocks.

4.2 Entwursphase

Im vorherigen Unterkapitel wurde der Aufbau der Schleifenblöcke beschrieben. Nun wird die technische Umsetzung beschrieben. Der Fokus wird dabei auf die Logik für die Entscheidung ob eine Wiederholung stattfindet oder nicht. Die beiden Schleifenblöcke, Block A und Block B, werden im folgenden einzeln betrachtet, da sie unterschiedliche Logiken haben.

Durch die einführung des Schleifenblocks können Probleme entstehen, welches zu einem nicht gewollten Veralten führen können. Es ist möglich, dass Endlosschleifen entstehen können.

Block A

Block A soll die Wiederholung von Aktivitäten durch explizite Ausführung ermöglichen. Zur umsetzung dieses Ziels wird die Schleifenentfaltung als Grundlage verwendet. Unter Schleifenentfaltung versteht man, dass die Instruktione im Schleifenkörper mehrmals pro Iteration auszuausgeführt wird, um dadurch die häufigkeit der Iterationen zu verringern. [9] Anstatt rekursive oder iterative Wiederholungen zu verwenden, wird jede potenzielle Iteration als eigenständigen und bedingten Codeblock realisiert. Dazu muss eine Anpassung an der klasisschen Schleifenentfaltungsverfahren vorgenommen werden. Das Ziel der modifizierung ist es, die Schleifen solange zusammenzufassen bis kein erneuter Schleifendurchlauf notwendig ist und dadurch den Kontrollfluss durch Verzweigungen und verschtelungen ohne Iterationen abzubilden. Da die maximale Anzahl an Schleifendurchläufen von vorneherein bekannt ist, kann für jede potenzielle Ausführen des Schleifenkörpers eine Kopie erstellt werden. In jeder Kopie wird der Schleifenkörper ausgeführt und anschließend um die Abbruchbedingung der Schleife ergänzt. Im Falle

einer erfüllten Abbruchbedingung wird hingegen keine weitere Kopie ausgeführt. Stattdessen wird die Ausführung gemäß dem vorgesehenen Kontrollfluss fortgesetzt. Auf Abbildung TODO wurde der Ansatz Graphisch dargestellt. In Abbildung TODO wird dieser Ansatz an einem konkreten Beispiel veranschaulicht. Im Beispiel soll "foo" drei mal einzeln auf der Console ausgegeben werden. In der oberen Schleife Hingegen in der unteren Schleife der modifizierte Ansatz verwendet wurden ist. Da die untere Schleife nur noch genau einmal ausgefürt wird, kann diese einfach weggelassen werden.

Block B

Block B soll die Wiederholung von Aktivitäten durch iterative Schleifen ermöglichen. Ist die Abbruchbedingung nicht erfüllt, wird Entschieden ob eine Wiederholung stattfindet oder nicht. Dabei wird geschaut wie sich das dynamische Datum über die Iteration hinweg verhält. Es wird überprüft, ob die Anzahl der Iterationen bereits die maximale Anzahl an Iteration überschritten hat oder der Chancen-Zähler den Wert 0 erreicht hat. Da der Vergleich auf Mittelwerten basiert, müssen die Eingaben in einen Gleitkommawert umgewandelt werden. Zunächst wird dafür die Eingabe in eine Zeichenkette umgewandelt und anschließend versucht als Gleitkommazahlen zu interpretieren. Schlägt die Interpretation fehl, handelt es sich nicht um eine Zahl. In dem Fall muss die Zeichenkette zeichenweise mithilfe von UTF-8 in eine Dezimalzhal überführt werden. UTF-8 bietet sich zur Umwandlung an, da es bereits eine eindeutige und standardisierte Codierung für über 1,1 Millionen Unicode-Zeichen bereitstellt und somit keine zusätzliche Festlegung eines eigenen Codierungsschemas erforderlich ist. Die einzelnen Zahlen werden dann mit ihren Index multipliziert und im anschluss addiert. Der Schritt mit der multiplizierung ist notwendig, weil die addtion kommutativ ist und somit keine berücksichtigung der Zeichenfolge erfolgt, wird durch Multiplikation mit der Zeichenposition eine positionsabhängige Gewichtung sichergestellt. Die Werte vom Typ double werden anschließend in einer Liste gespeichert, die alle bisherigen Eingaben beinhaltet. Basierend auf dieser Liste wird dann ein Mittelwert über alle bisherigen Werte gebildet und ein gleitender Mittelwert über die letzen 3 Werte. Anschließend wird die differenz zwischen den aktuellen Wert und den Mittelwerten gebildet. ist die differenz kleiner als ein threshold wird der counter dekremntiert. ist die differenz gleich oder größer als der Threshold wird der Chancen-Zähler zurückgesetzt. Erreicht der Chancen-Zähler 0 werden keine weiteren Wiederholungen ausgeführt, da der wert sehr wahrscheinlich stagniert. Der Chancen-Zähler erweist sich als erforderlich, da dieser als Steuermechanismus für die Wiederholungslogik dient und eine Begrenzung der Iterationen bei ausbleibender signifikanter Veränderung sicherstellt. Dadurch soll verhindert werden, dass eine ineffiziente Fortsetzung der Schleife stattfindet. Sollten weniger als 3 Werte in der Ergbenissliste drin sein, wird die Berechnung übersprungen. Dadurch kann gewährleistret werden, dass das System einlaufen kann und die Mittelwerte erst gebildet werden, wenn eine aussagekräftige Datenbasis vorhanden ist.

Auf die Vor- und Nachteile der einzelnen Blöcke wird im späteren Verlauf eingegangen.

5 Implementierung

In diesem Kapitel wird die vorgenommene Implementierung beschrieben. Dabei wird zunächst TODO in Teilbereiche unterteilt und anschließend die funktionsweise der einzelnen Änderungen erklärt. Als Grundlage für dieses Kapitel werden die vorangegangen Kapitel dienen insbesondere Kapitel TODO. Das Ziel ist es die Implementierung verständlich und nachvollziehbar zu machen.

Zur umsetzung der Wiederholungslogik im Block B wurden vier Klassen definiert. Dabei handelt es sich unter anderem um die Klasse DeviationChecker, die in Abbildung TODO dargestellt ist.
Ziel der Klasse ist es den Mittelwert zu berechnen und die Differenz zum aktuellen Wert auf ihren Schwellenwertüberschreitung hin zu überprüft. Zur Umsetzung dieser FUnktionalität hat DeviationChecker eine Mittelwert-Variable vom Datentyp Double sowie die drei Methoden calcute, calculateMean und isDeviationWithinThreshold.

Die Methode calculateMean übernimmt die Berechnung des aktuellen Mittelwerts, indem über die Liste der Eingaben iteriert wird und diese aufsummiert werden. Der berechnete Summenwert wird durch TODO geteilt und anschließend als Rückgabewert zurückgegeben.
Mit Hilfe der Methode isDeviationWithinThreshold wird die Differenz berechnet und überprüft ob die Differenz den Schwellenwert unterschreitet. Dies erfolgt durch die Berechnung des Betrags der Differenz zwischen Mittelwert und dem aktuellen Wert. Je nach Ergebnis wird dann ein entsprechender Wahrheitswert zurückgegeben.
Der Ablauf der Klasse wird in der Methode calculate gesteuert. Hier wird zunächst überprüft, ob die anderen Methoden aufgerufen werden soll. Der Algorithmus wird lediglich bei Vorhandensein von mindestens drei Werten in der Eingabeliste gestartet. Ist dies der Fall wird die Methode calculateMean aufgerufen und dessen Rückgabewert in die Mittelwert-Variable gespeichert. Im Anschluss erfolgt der Aufruf von isDeviationWithinThreshold, dessen Rückgabewert von calculate zurückgegeben wird.

Ein andere Klasse ist Counter. Diese dient der Verwaltung zweier Zählerwerte: chance und counter. Der Zugriff auf diese Variablen erfolgt nach dem Prinzip der Datenkapslung. Zur Funktionalität stehen Methoden zum Verrigern (decreaseChance und decreaseCounter), Zurücksetzen (resetChance) sowie Setzen und Auslesen (setCounter, getCounter, getChance) der Zählerwerte zur Verfügung.

Mit der Klasse StringToDoubleConverter wird ein String in einen Double-Wert umgewandelt. Um dies zu ermöglichen stellt die Klasse verschiedene Variable zur verfügung, darunter eine Variable für den Zähler der Schleife, einen für die Bytes und einen für den umgewandelten Wert. Die Methode StringToDouble

konviert die Zeichenkette, indem sie zunächst die Zeichenkette in UTF-8 kodierte Bytes umgewandelt. Anschließend werden die Bytes mit einem fortlaufenden Index multipliziert und auf einen Akkumulator addiert. Der akkumulierte Wert wird im Anschluss als Rückgabewert zurückgegeben.

Die letzte der vier Klassen ist InputProcessor. Diese ist für die Verarbeitung der Eingabewerte zuständig. Zur Verarbeitung stehen mehrere interne Datenstrukturen und konstanten zur verfügung, darunter zwei Listen zur Speicherung der Roh- und Konvertierungswerte sowie Parameter zur Konfiguration (HistoryLength und deviationThreshold) und eine Instanz der Klasse Counter. Die Methode processInput übernimmt TODO und überprüft zunächst ob eine weitere Verarbeitung zulässig ist. Als Kriterium dafür wird der aktuelle Stand der Klasse Counter herangezogen. Ist eine weiter verarbeitung zulässig wird zuerst TODO und der Eingabewert mithilfe der Methode storeInput in eine Liste gespeichert. Anschließend wird der Eigabewert mit der Methode convertInputToDouble in einen Double-Wert konvertiert. Dieser Wert wird nach der Konvertierung dann in eine Liste gespeichert. Im Anschluss daran wird mithilfe der Methode validateDeviation der konvertierte Wert auf eine mögliche Abweichung hin überprüft. Als letzter Schritt der Verarbeitung wird der Zähler dekrementiert.

Die Methode convertInputToDouble TODO. Ist dies nicht möglich, wird der Eingabewert mit der Methode StringToDouble aus der Klasse StringToDoubleConverter in einen Double-Wert konviert. Als Rückgabewert wird der Double zurückgeben.

Mithilfe der Methode validateDeviation wird überprüft, ob der Eingabewert eine signifikate Abweichung im Vergleich zur Historie aufweist. Dafür wird zunächst der Index des letzten Elements aus der Liste bestimmt und anschließend die Abweichungsberechnung über die Methode calcute der Klasse DeviationChecker durchgeführt. Liegt eine signifikante Abweichung vor wird der Zähler resetet. Ist hingegen keine Abweichung vorhanden wird der Zähler dekremntiert.

Damit Zyklen übern den Schleifenblock erlaubt sind und es zu keinen Fehler kommt, muss die Zylkenüberprüfung angepasst werden. Wie die Zylkenüberprüfung funktioniert wurde bereits in Kapitel TODO beschrieben. Die Änderung am diesem Algorithmus muss an der Stelle stattfinden an der die TODO. Hier muss bevor TODO geschaut werden ob der Root der Schleifenblock ist. Ist dies der Fall Fall wird der Funktionsaufruf übersprungen. Dadurch wird für die Rückverbindung kein ComponentEntry erstellt bei dem root und element ungleich snd. So eine Kombinantion führt nämlich im weiteren Verlauf dazu, dass ein Eintrag im Fehler Dicentario erstellt wird und dadurch Rückführung ungültig ist. Da der Algorithmus die einzelnen Knoten über IDs identifiziert, muss eine Liste erstellt werden, welche alle IDs der Schleifenblöcke beeinhaltet. Bei der überprüfung wird nun geschaut ob die ID des roots in der Liste ist. Ist

der Root in der Liste handelt es sich um einen Schleifenblock. Um eine Liste mit den IDs zu erstellen, kann eine Kombination aus den Methoden where und select und kommen aus der statischen Klasse Enumerable im Namensraum System.Linq. Durch where wird TODO gefildert und mit select wird die gefilderte TODO projektiert.

6 Evaluation

6.1 1. Lösungsansatz

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

6.2 2. Lösungsansatz

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

7 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.