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. [**?**] Häufig ist diese dabei die zeitintensivste Komponente eines Programms, da ihre die Ausführung sehr viel Zeit in anspruch nehmen kann. [**?**] Der Algrorithmus dieser Kontrollstruktur kann dabei iterativ oder rekursiv implementiert werden. Ersteres wiederholt die Schleife mehrmals. Hingegen die Rekursion sich mehrmals selbst aufruft. [**?**] 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. [**?**] 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. [**?**] 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. [**?**] 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. Hingegen sind die Ergebnisse der Schleifendurchläufe bei der horizontalen Schleife unabhängig voneinander. [**?**]

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. [**?**] DSLs bilden das Gegenstück zu General-Purpose Languages (GPL) wie Java, C++ oder Python. [**?**] Anders als bei GPLs verfügen DSLs oftmals über eine reduzierte Syntax, die ausschließlich für die jeweilige Domaine relevant ist. Teilweise wird DSLs durch GPL ergänzt. [**?**] Bei DSLs 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 werden müssen. Außerdem braucht der Benutzer länger Zeit um die Syntax zu lernen. [**?**] Zur Laufzeit wird die externe DSLs dann in eine GPL übersetzt. [**?**] Interne DSLs hingegen verwenden die Syntax einer GPL und können über eine Programmierschnittstelle oder Biblio-

thek aufgerufen werden. [?] 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. [?]

Visual Programming Language

Visuelle Sprachen (VL) sind Sprachen, bei denen die Informationsdarstellung primär über grafische Elemente und nicht über textuelle Komponenten erfolgt. [?] Dabei werden hauptsächlich grafische Tools und visuelle Metaphoren verwendet. Bilder eignen sich besonders gut zum Programmieren, weil diese ausdruckstärker als Worte sind und einen höheren Wiedererkennungswert haben. Nachteile gegenüber Text-basierten Sprachen sind, dass VLs durch die eigeschränkte Syntax nicht so ausdruckstark und flexibel sind. [?]

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 von VPLs besteht in der Verbesserung der Darstellung der Programmierlogik sowie in der Erleichterung des Verständnisses von Programmabläufen. [?] Dadurch soll der Fokus bei Programmieren stärker auf die konzeptuellen statt symtaktischen Aspekte verlagert werden. Die syntaktischen Aspekte werden von der Entwicklungsumgebung übernommen. [?] Die Umsetzung von Programmen erfolgt durch Flussdiagrammen, die vom Benutzer erstellt werden können. Die erstellten Flussdiagramme werden dann vom System interpretiert und ausgeführt. [?] Nach Charntaweekhun bieten Flussdiagramme einen didaktischen Vorteil, da diese Programmieranfängern ermöglichen, komplexe Abläufe visuell zu erfassen und zu strukturieren. [?] Ein weiterer Vorteil von VPLs besteht in der erhöhten Lesbarkeit und der geringeren Anfälligkeit für syntaktische Fehler, was auf die Verwendung einer begrenzten Menge vordefinierter grafischer Elemente zurückzuführen ist.[?] Die stärken von VPLs zeigen sich darüber hinaus in ihrer Einfachheit, visuelle Darstellbarkeit, Transparenz und Interaktivität. [?] 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.[?] Visuelle Programmiersprachen (VPLs) basieren überwiegend auf einem datenflussgesteuerten Modell, bei dem die Strukturierung von Programmen durch den Austausch von Informationen zwischen Operatoren erfolgt. [?] Zusammengefasst kann man sagen, dass VPLs die Vorteile von Flussdiagrammen und nicht die Nachteile der klassischen Programmierung kombiniert. [?]

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. [?] Dieser wird als gerichteter Graph

(DG) definiert, in dem Funktionen als Knoten dargestellt werden. Die Knoten können dabei durch gerichtete Kanten verbunden werden, welche die Datenabhängigkeiten zwischen zwei Knoten beschreiben. [**?**] 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.[**?**] Neben der Granularität lässt sich ein Datenflussgraph auch in Zyklenstrukturen unterteilen. Dabei wird zwischen zyklisch und azyklisch unterschieden. [**?**] DLs sind überwiegend funktional geprägt, aber können auch textbasiert sein. [**?**] Der Vorteil einer DLs ist, dass diese durch einen Graphen dargestellt werden können [**?**] und dadurch die Programme einfach zur verstehen sind. [**?**] 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. [**?**] DLs führen Instruktionen nicht in einer festen Sequenz aus. Dadurch können unabhängig voneinander ausführbare Instruktion parallel verarbeitet werden. [**?**] Diese Form der Ausführung ermöglicht eine Effizienzsteigerung, da der Ablauf nicht mehr durch einen zentralen Programmzähler gesteuert wird. [**?**]

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.[**?**]

Eine Computerarchitketur, die auf DLs basiert, wird Datenfluss-basiertes System (DBS) genannt. DBSs wurde eingeführt um den Flaschenhals der von-Neumann-Architektur zu vermeiden. [**?**] Vorteile von DBSs sind eine hohe Effizienz, flexible Strukturen und leistungsstarke Ausführungsmechanismen. [**?**] Darüber hinaus besteht ein weiterer Vorteil im möglichen Paralellismus, da bei fehlender Datenabhängigkeit mehrere Insturktionen gleichzeitig ausgeführt werden können. [**?**] Dies wird durch die direkte Weitergabe von Daten zwischen Funktionen unterstützt, wobei die Verarbeitung Der Daten, wie Transformation und Filterung, innerhalb der Funktionen erfolgt. [**?**] Innerhalb von DBSs wird zwischen daten- und bedarfgetriebener Ausführung unterschieden. Ersteres führt die Funktionen aus, sobald alle Operanden vorhanden sind und ein Signal vorliegt. Hingegen bei der bedarfgetriebenen Ausführung die Funktion ausgeführt wird, sobald alle Operanden vorhanden sind. [**?**] Aus dem Grund kann die datengesteuerte Ausführung als Spezialfall einer bedarfsgesteuerten Ausführung angesehen werden, bei der ein Bedarf an allen Ergebnissen von vorneherein besteht. [**?**]

Wasserfall Modell

Das Wasserfallmodell ist ein Modell für Softwareentwicklung [**?**] und wurde 1970 eingeführt. [**?**] Die Softwareentwicklung wird dabei in verschiedene voneinander getrennte Phasen unterteilt. [**?**] Es handelt sich beim Wasserfallmodell um ein

statisches Modell, da die einzelnen Phasen linear und sequentiell durchlaufen werden. [**?**] Sobald eine Phase abgeschlossen ist, kann diese nicht wieder besucht werden. [**?**] Aus dem Grund ist das Modell besonders gut für Systeme geeignet, welche nach der Implementierung keine Änderungen mehr zulassen [**?**]. Die Vorteile eines Wasserfallmodells sind, dass es einfach verständlich ist [**?**], die Anforderungen an das System bereits vor der Entwicklung bekannt sind und es einfach umzusetzen ist. [**?**] Außerdem trägt die Dokumentation am Ende jeder Phase zur Verbesserung der Projektqualität bei. [**?**] Hingegen ein großer Nachteil von Wasserfallmodellen ist, dass sich ändernde Anforderungen oder auftretende Probleme während der Entwicklung nicht berücksichtigt werden können. [**?**] Um den Nachteilen des klassischen Wasserfallmodells etwas entgegenzuwirken haben sich die letzten Jahre einige Varianten entwickelt. So beschreibt TODO (24) in seinem Artikel eine Variante, bei der die einzelnen Phasen einer sorgfältigen Validierung und Bestätigung seitens des Clients benötigen um die Phase abzuschließen. Sollte der Client mit dem Ergebnissen nicht zufrieden sein, so kann das Modell wieder von vorne beginnen. [**?**] In einer anderen Variante wird die Test-Phase komplett weggelassen, da das Testen während der gesamten Entwicklung stattfinden soll. [**?**]

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

Die Grammatik der domainspezifischen Sprache lässt sich Formal in drei Ebenen unterteilen. Die oberste Ebene ist die Prüfungslogik, die mittlerste Ebene ist die Datenverarbeitung und die letzte Ebene ist das Typsystem. [**?**] Ersteres beschreibt die benötigten Aktionen und Entscheidungen für eine Prüfung. [**?**] Hingegen die Datenverarbeitung für die Definition von Datentranformationen zuständig ist und eine von mehreren Aktionstypen darstellt. [**?**] Im gegensatz dazu kümmert sich das Typsystem um die statische Analyse der Ausführbarkeit der vorangegangen Ebenen. [**?**]

3.1.1 Prüfungslogik

Die in Abbildung TODO abgebildeten Regeln beschreibt das Aktivitätsmodell, die der Prüfungslogik entspricht. [**?**] Zentral ist dabei die Regel *ActivityModel*, welche festlegt, dass ein Aktivitätsmodell aus mehreren Aktivitäten und Verbindungen besteht. Die Verbindung referenziert dabei zwei Aktivitäten und hat auch eine Bezeichnung, die als TODO. Die Regel *Activity* beschreibt die möglichen Aktivitäten innerhalb des Aktivitätsmodells. Dabei kann eine Aktivität entweder eine Startmarkierung, eine Entscheidung, eine Aktion oder ein Label sein.

Eine Entscheidung kann dabei ein Binärentscheidung oder ein Validierungsentscheidung sein. Ersteres besteht dabei aus einer Referenz auf ein Flowtemplate, das eine Datenverarbetung ist, gefolgt von einem Operator und zwei Argumenten. Als Operatoren stehen $=$ und $\neq$ sowie Relationale Operatoren zur Verfügung. Die Validierungsentscheidung hingegen besteht nur aus einer Menge von werten, die entweder.

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 dabei aus den Symbolen ActivityPortValue und TemplateParameter-Value 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*⟩*

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

⟨*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 kann eine Flow-Instanz als konkreten Aufruf einer Funktion interpretiert werden. Die Regel *FlowInstance* legt dabei fest, dass eine Flow-Instanz aus mehreren Eingabe- und Ausgabeports sowie einer Menge von 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. Das FlowTemplate wird dabei durch eine Funktion *Flow* und eine Menge von TemplateParametern definiert. Bei den TemplateParameter wird zwischen einer Zeichenkette, Zahlen oder einem Wahrheitswert unterschieden. Diese Parameter beeinflussen die automatische Generierung der Port-Struktur sowie der internen Verarbeitungslogik eines Flows. [**?**] Die Funktionen lassen sich in vordefinierte *LibraryFlow* und eigene erstellten Funktionen *FlowModel* unterteilen.

⟨*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. Das FlowModel entspricht dabei der Datenverarbeitung. [**?**] Durch die Regel *FlowModel* 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. Flow-Nodes sind Funktionen die innerhalb eines Flow-Modells genutzt werden.

⟨*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

In Abbildung TODO werden die Regeln des Typsystem dargestellt. Das Typsystem unterstützt Primitive- und Generische-Datentypen, welche an einen Optional-Typen, Listen-Typen oder Objekt-Typen gekapselt werden können. Als Primtiven-DatenTypen stehen String, Number und Bool zur Auswahl. Zusätzlich noch Data und PtiaResponse. Ersteres ist eine Datumsanagbe. Hingegen PtiaResponse die Antwort von einem Hauptunsersuchungs-Anfrage ist.

⟨*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.

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 vorkommen darf. Die Reihenfolge der auszuführenden Aktivitäten wird durch die grade ausgeführte Aktivitäte vorgegeben, da jede Aktivität das Label der zu folgende Kante als Rückgabewert 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 die gespeicherten Ergebnisse referenzieren, um diese als Parameter für ihre Funktionen zu verwenden.

Datenverarbeitung

Die Ausführung der Datenverarbeitung basiert auf einer Execute-Funktion, die zur ausführung die Template-Parameter und eine Evaluate-Funktion benötigt. Für die vom System bereitgestellten Funktionen wird die Execute-Funktion direkt in den Code implentiert. Die Implementierung nutzt dabei eine Hilfsklasse, die die Evaluate-Funktion bereitstellt und gleichzeitig die Werte der Ausgabeports speichert. Zudem muss die Implementierung sicherstellen, dass der Datenfluss zwischen Eingabe- und Ausgabeports korrekt hergestellt wird. Um dies zu gewährleisten, stellt die Klasse RuntimeContext gewisse Hilfsmethoden zur Verfügung.

Eine wichtige Komponente der Datenverarbeitung ist der Ergebniscache. Im Ergebniscache werden die Werte der Ausgabeports gespeichert. Wird ein Wert für einen Eingabeport benötigt, so wird zunächst nach dem zugehörigem InputNode sowie der entsprechenden eingehen Kante gesucht. Anschließend wird geprüft, ob für den Ausgabeport, der mit der Kante verbunden ist, ein Wert im Ergebniscache vorliegt. Ist dies nicht der Fall, wird die Funktion des Knotens 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. Wenn ein Wert für einen Eingabeport einer Lambda-Funktion benötigt wird, wird zunächst der Wert des Ausgabeports, falls vorhanden, in dem Ergebniscache gespeichert. Im Anschluss daran wird für alle Funktionen, die im Kind-verhältnis zur Lambda-Funktion stehen, das Ergebnis im Ergebniscache invalidiert, indem das Ergenis aus dem Ergebniscache gelöscht wird. Anschließend wird der oben beschriebene Algortihmus verwendet, um die Werte für die Eingabeport der Lambda-Funktion zu berechnen.

TypSystem

Bevor die Typvalidierung bei Flow-Modells vorgenommen werden kann, muss bei dem Modell zunächst die Flow-Templates in die konkrete Flow-Instanz überführt werden. Anschließend werden die Verbindungen als Objektreferenzen geändert. Das daraus entstandene Modell wird Graph-Form genannt und wird für die Analyse verwendet. Bei der Graph-Form werden als erstes die Referenztypen durch konrekte Typen ersetzt. Nach dem die ersetzung Abgeschlossen ist, wird sich um die generischen Typen gekümmert. Dafür werden zunächst die Flow-Modelle topoloisch sortiert und pro Flow-Modelle über alle Verbindungen iteriert. Kommt bei einem Port ein Generisch Typ vor wird die Typ-Zuweisung gespeichert. Es wird probiert über die anderen Typ-Zuweisungen die generischen Typen aufzulösen. Sobald dies erledigt ist, wird die Zuweisungskompatibilität der einzelnen Verbindungen überprüft. Als letztes kann das Modell dann auf Port-Fehler überprüft werden. Hierfür werden zunächst die Flow-Node topologisch sortiert, sodass Abhängigkeiten einer Flow-Node vor der Flow-Node validiert werden. Bei der Validierung für für jede Flow-Node jedes Argument und dessen Verbindung untersucht. Bei der untersuchung wird geschaut, ob das Attribut acceptsError des dazugehörigen Eingabeports den gleichen Wert hat wie der Ausgabeport. Ist dies nicht der Fall wird eine Fehlermeldung für diesen Flow-Modell ausgegeben. Dabei wird die Fehlermeldung durch alle Flow-Nodes weitergeleitet bis sie am Ausgabeport des Flow-Modells anliegt.

Zur Validierung des Aktivitätsmodell wird ein Typen-Referenzstack verwendet. Bei der Validierung wird durch alle Aktivitäten iteriert und für jede Aktivität der Typen-Referenzstack berechnet. Berechnet wird der Referenzstack durch den Schnitt aller Typen-Referenzstack vorhergehenden Aktivitäten. Je nachdem welche Aktivität gerade ausgeführt wird, wird der Referenzstack bearbeitet. Bei einer Aktionsaktivität wird der Referenzstack durch einen Eintrag erweitert. Hingegen bei einer Entscheidungsaktivität Typen im Referenzstack bearbeitet werden können.

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

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 -

Beim betrachen der Klasse VirtualMachineCompilerActivity, die für das kompilieren der Aktivitäten zuständig ist, ist aufgefallen, dass dort ein Fehler vorliegt.

4 TODO

4.1 Anforderungsphase

Die geplante Erweiterung sieht die Einführung von zwei Schleifenblöcken vor, die die Wiederholung von Aktivitäten ermöglichen sollen. Dadurch soll es möglich sein Prüfungen mit dynamsichen Daten verarbeiten zu können. Der einzige Unterschied ist, wie der Algorithmus die Schleife ausführen will. Der Benutzer soll die Möglichkeit haben per drag-and-drop den Schleifenblock aus einer Liste auszuwählen und diesen frei im Canvas platzieren zu können. Sobald der Block platziert wurden ist, kann der Benutzer mit dem Schleifenblock interargieren. Die Schleifenlogik soll durch den Benutzer konfigurierbar sein. Dafür wird auf die bereits vorhandene grafische Benutzeroberfläche des Systems zurück gegriffen.

4.2 Entwursphase

Schleifenblock

Für die Integration der neuen Schleifenblöcke ist eine Anpassung der Grammatik der Prüfungslogik erforderlich, konkret an der Regel *Activity*. Hier muss die Regel eine Alternative erweitert werden, die ein neues Nichterminale einführt. Das Nichtterminal *ActivityLoop* kann dabei SchleifeA oder SchleifeB sein. Beide Varianten setzen sich dabei aus einem ganzzahligen Wert für die Anzahl der Iterationen und einer Abbruchbedingung zusammen. Die Abbruchbedingung wird durch das Nichtterminalsymbol *ActivityBinaryCondition* abgebildet. Die entsprechende Regel für das nichterminal wurde bereits definiert. Dargestellt wird die Erweiterung in Abbildung TODO. Auch hier beinhaltet die Erweiterung nur die für die Ausführungssemantik benötigten Elemente.

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

⟨*ActivityLoop*⟩ ::= ⟨*integer iteration*⟩ ⟨*ActivityBinaryCondition*⟩

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

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

Grammatik TODO Erweiterung Prüfungslogik

Der Schleifenblock soll dabei als Oval im Canvas dargestellt werden. Beim Klicken auf den Schleifenblock soll der Benutzer die Möglichkeit haben die Anazahl der Iterationen und die Abbruchbedingung für die Schleife festzulegen.

Zusätzlich TODO. Die Abbruchbedingung setzt sich dabei aus zwei Feldern für die Argumente und einer Auswahlsliste für den Operator zusammen. Am Block selbst kann der Benutzer die Verbindungen erstellen, indem er per drag-and-drop die einzelnen Interaktionspunkte verschiedener Aktvitäten verbindet. Rückverbindungen über den Schleifenblock sollen dabei erlaubt sein. Ingesammt stehen dem Schleifenblock vier Interaktionspunkte zur Verfügung. Welcher Pfad genommen wird, wird durch den Wahrheitswert der Abbruchbedingung bestimmt. Ist die Abbruchbedingung Wahr, wird die Verbindung vom Interaktionspunkt unten genommen. Wenn hingegen die Abbruchbedingung Falsch ist, wird die Verbindung vom Interaktionspunkt rechts genommen.

Block A

Block A soll die Wiederholung von Aktivitäten durch explizite Ausführung ermöglichen. Das soll geschehen indem die Schleife durch eine verschachtelte if-Anweisung ersetzt wird. Zur umsetzung dieses Ziels wird die Schleifenentfaltung als Grundlage verwendet. Unter Schleifenentfaltung versteht man, dass die Instruktione in der Schleife mehrmals pro Iteration auszuausgeführt werden, um dadurch die häufigkeit der Iterationen zu verringern. [**?**] Um das Ziel zu erreichen, muss die Schleifenentfaltung abgeändert werden. Die Schleifenentfaltung soll nicht nur häufigkeit der Iterationen reduzieren, sondern diese vollständig eliminieren. Da die maximale Anzahl an Iterationen von vorneherein bekannt ist, kann für jede potenzielle Iteration eine Kopie des Schleifenkörper erstellt werden. Am Ende jeder Kopie muss dann noch die Abbruchbedingung drangehangen werden, da diese TODO. Als nächstes müssen die einzelnen Kopien miteinander verbunden werden. Dies wird erreicht indem von der ersten Aktvität einer Kopie eine Verbindung zur vorangegangenen Schleifenblock hergestellt wird. Graphisch wird dieser Ansatz in Abbildung TODO dargestellt.

Block B

Block B soll die Wiederholung von Aktivitäten durch iterative Schleifen ermöglichen. Zur umsetzung dieses Ziels wird ein Mittelwertansatz als Grundlage verwendet. Wird die OnFalse Verbindung des Schleifenblocks ausgewählt wird als erstes überprüft, ob eine Wiederholung stattfinden darf oder nicht. Dafür wird geschaut, ob die Anzahl an Interationen bereits das maximum erreicht haben oder der Chancen-Zähler den Wert 0 erreicht hat. Sind beide Bedingungen nicht erfüllt, ist eine Wiederholung erlaubt wird TODO. Dabei wird das erste Argument aus der Abbruchbedingung genommen und versucht in einen Gleitkommawert umgewandelt. Dafür wird das Argument als erstes in eine Zeichenkette umgewandelt und anschließend versucht als Gleitkommazahl zu interpretieren. Ist dies Möglich, wird die die Gleitkommazahl in eine Liste mit allen bisherigen umgewandelten Argumenten gespeichert. Schlägt die Interpretation fehl, ist das Argument keine Zahl gewesen und die Zeichenkette muss mithilfe von UTF-8 in eine Dezimalzahl umgewandelt werden. Bei der Umwandlung

wird die Zeichenkette zeichenweise in die entsprechende Dezimalzahl umgewandelt und anschließend mit ihren Index multipliziert. Im Anschluss werden die einzelnen Zahlen addiert, sodass am Ende nur noch eine Summe übrig bleibt. Die Summe wird anschließend in eine Gleitkommazahl umgewandelt und der Liste hinzugefügt. In Abbildung TODO wird die Rechnung für ein Beispiel durchgegangen. Basierend auf der Liste mit allen bisherigen umgewandelten Argumenten wird dann ein Mittelwert über alle Werte gebildet und ein gleitender Mittelwert über die letzen 3 Werte. Anschließend wird die Differenz zwischen dem aktuellsten Wert und den Mittelwerten berechnet. Ist die Differenz kleiner als ein Threshold wird der Chancen-Zähler dekremntiert. Anderfalls wird der Chancen-Zähler zurückgesetzt. Der Chancen-Zähler ist erforderlich, da dieser als Steuermechanismus für die Wiederholungslogik dient und eine Begrenzung der Iterationen bei ausbleibender signifikanter Veränderung sicherstellt. Sollten weniger als 3 Werte in der Liste drin sein, wird die Berechnung der Differenz übersprungen und direkt mit der Wiederholung der Aktivitäten begonnen. 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

Zunächst wurde Klasse DataModelActivityElement um das Attribute *CanCreateLoop* vom Datentyp bool erweitert. Bei dieser Klasse handelt es sich um die Basisklasse aller Aktivitäten. Das Attribut soll als Kennzeichnung dienen, um anzuzeigen ob der User die Möglichkeit hat von dem Block ausgehend eine Rückverbindung einzuzeichnen. Aus dem Grund war das Attribut für alle Aktivitäten bis auf die Schleife false. Da das System keine Schleifen am Anfang unterstützt hat, musste für die Schleife eine Klasse erstellt werden. Bei der Klasse handelt es sich um DataModelActivityElementLoop, die als Grundlage für die Datenspeicherung und für die Ausführung genutzt wird. Der Klasse stehen zwei Variablen für die Ausgänge und eine Variable für die Anzahl der Iterationen. Für die Variablen stehen jeweils noch getter und setter als Methoden zur verfügung, damit die Werte gesetzt und ausgelesen werden können. Zusätzlich stehen der Klasse durch die Veerbung die Methoden und Variablen der Basisklasse DataModelActivityElement zur verfgügung. Die Klasse ActivityPortLabels, die die Label für die Ports beinhaltet, wurde durch die strings OnLoop und OnFinish erweitert.

Klasse VirtualMachineCompilerActivity zuständig für die Kompilierung der Aktivitäten. Es müsste ein Statement für die Schleife erstellt werden. Dieses Statement wird in der Klasse VirtualMachineActivity ausgeführt. Damit dies Funktioniert

Die in diesem Absatz beschriebenen Änderungen wurden durch Westermann implementiert.

Zur umsetzung der Wiederholungslogik im Block B wurde eine Klasse definiert. Dabei handelt es sich um die Klasse ActivityLoopUnrollingResolver. Ziel der Klasse ist es Schleifen zu identifizieren und diese durch den in Kapitel TODO beschriebenen Algorithmus aufzulösen. Zur Umsetzung dieser Funktionalität hat die Klasse sieben Methoden UnrollLoop, getLoopActivities, visited, getLoopElements, cloneAndModify, DeepClone und clone.

Der Startpunkt für den Algorithmus ist die Methode UnrollLoop. Diese Methode hat das Ziel, den Schleifenanfang und das Schleifenende zu erkennen. Außerdem leitet die Methode UnrollLoop die Schleifenentfaltung ein. Dafür stehen der Methode die Liste aller Elemente, ein Set für die Verbindungen und zwei Variablen für den Schleifenanfang und das Schleifenende zur verfügung. Um die beiden Variablen zu konkretisieren, wird als erstes über die Liste iteratiert und überprüft ob das Element die Eigenschaft CanCreateLoop auf wahr hat. Ist das der Fall, ist dieses Element das Schleifenende und wir in die zugehörige Variable gespeichert. Um den Schleifenanfang zu finden, wird bei jeden Element überprüft, wie viele Vorgänger das Element hat. Hat das Element mehr als zwei Vorgänger, handelt es sich um den Schleifenanfang. Sobald die beiden Variablen bestimmt sind, kann die Methode getLoopActivities aufgerufen werden.

Das Ziel der Methode getLoopActivities ist es die Rahmenbedingung für die

Tiefensuche zu schaffen. Die Methode stellt dafür ein Set für die gefunden Verbindungen, einem Tupel, welches die Rückverbindung darstellt und einer Variable, die die Id des Schleifenanfangs beinhaltet bereit.

Die Methode visited führt eine rekursive Tiefensuche ab dem Schleifenanfang im Graphen durch. Dabei wird vom jeden Knoten aus geprüft, ob es eine ausgehende Verbindung gibt. Wurde eine Verbindung gefunden, wird geschaut ob diese der Rückverbindung der Schleife ist. Ist das der Fall wird die Verbindung übersprungen. Anderfalls wird die Verbindung dem Set mit allen Verbindungen hinzugefügt. Im Anschluss wird die Funktion rekursiv für den Nachfolgenden Knoten aufgerufen.

In der Methode getLoopElements werden die Ids in konkrete TODO übersetzt. Dafür wird zunächst über das Set, welches in der Methode visitied erstellt wurden ist, iteratiert und für jedes Element aus dem Tupel das dazugehörige Object aus der Liste aller Elementen selektiert. Die zwei selektieren Elemente werden dann wieder als Tupel in ein Set gespeichert, welches am Ende auch als Rückgabewert der Methode dient.

Die Hilfemethode clone TODO ist für das Klonen der Elemente zustädnig. Dazu wird das übermittelte TODO in serialisiert und anschließend deserialisiert. Die beiden Methoden werden von der Klasse JsonSerializer bereitgestellt Durch die Deserialiserung entsteht eine exakte Kopie vom übermittelten TODO. Als Rückgabewert der Hilfsmethode wird das kopierte Objekt zurückgegeben.

Die Methode cloneAndModify ist für die organsierung des Klonvorgangs und bereitet die Rekursion vor. Um dies zu realisieren steht der Methode ein Set für die geklonten Elemente zur verfügung. Zunächst wird über das Set der TODO iteratiert und dort für Item aus dem Tupel eine Kopie mithilfe der Methode clone erstellt. Anschließend werden die Kopien wieder als Tupel in das Set für die geklonten Elemente hinzugefügt.

Die letzte Methode ist clone. Das Ziel dieser Methode ist die Erstellung der Kopie und änderungen an dieser vorzunehmen. Dafür wird als erstes überprüft, um welche Aktivität es sich bei dem übergebenen Element genau handelt. Sobald dies feststeht, wird eine Kopie mithilfe der Methode DeepClone erstellt. Anschließend wird die Id des kopierten Objekts um einen Suffix erweitert. Als nächstes werden die Verbindungen überprüft. Existiert eine Verbindung wird diese auch um den gleichen Suffix erweitert. Die Ausnahme dafür bilden die Verbindungen des Schleifenblocks. Hier wird an der Verbindung, die aus der Schleife führt, keine Änderung vorgenommen, weil an den Verbindungen nachdem Schleifenrumpf keine Veränderungen vorgenommen werden. Und die eigentliche Rückverbindung muss auf das erste Elemente der Kopie verweisen. Dafür muss die Verbindung des Urbilds geändert werden. Anders gesagt, das Suffix muss an die Verbindung des Urbilds drangehangen werden statt an der Kopie. Durch das anhängen des Suffix wird eine eindeutige TODO geschaffen,

außerdem bleibt die ursprüngliche Struktur des Schleifenrumpf erhalten.

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.

Um Zyklen in der Prüfung zu erlauben, ist eine Anpassung der Zyklenprüfung erforderlich. Die Funktionsweise der Zyklenprüfung wurde bereits im Kapitel TODO beschrieben. Eine Änderung muss am Anfang der Methode *ComponentAnalysis()* vorgenommen werden. Hier muss bei der Iteration über alle Elemente jede Id in ein Set gespeichert werden, das ein Schleifenblock ist. Die andere Änderung muss in der Methode *AssignToComponent()* erfolgen. Konkret an der Stelle an der über die Vorgängerelemente iteriert wird. Dort müssen vor dem reksuriven Methoden Aufruf zwei Bedingungen überprüft werden. Erstens ob das Vorgängerelement ein Schleifenblock ist. Dies geschieht, indem geschaut wird, ob die Id des Vorgängerelements in dem Set vorkommt. Wenn dies der Fall ist, handelt es sich bei dem Vorgängerelement um einen Schleifenblock. Zweitens ob das Element und das Vorgängerelement die gleichen Elemente sind. Dies kann überprüft werden indem die beiden Elemente auf gleichheit überprüft werden. Wenn beide Bedingungen erfüllt sind, handelt es sich um die Rückverbindung der Schleife. In diesem Fall darf die Methode *AssignToComponent()* nicht aufgerufen werden, sondern muss übersprungen werden. Wenn die Methode nicht übersprungen wird, wird die Verbindung in das Dictionary aufgenommen, was im späteren Verlauf des Algorithmus zu einem Fehler

führen würde. Durch diese beiden Änderungen ist es nun möglich, ausgehende Schleifen vom Schleifenblock aus zu erzeugen, ohne dass ein Fehler entsteht. Ausgehende Schleifen von anderen Aktivitäten aus sind weiterhin nicht erlaubt.

6 Evaluation

Das Programm wurde auf einem System mit einem AMD Ryzen 5 2600 mit 6 Kernen und einer Taktrate 3400MHz ausgeführt. Als Arbeitsspeicher waren 2-Mal 8GB DDR4 mit einer DRAM Frequenz von 1065 MHz eingebaut. Als Betriebssystem war Windows 10 in der Version 10.0.19045 BUild 19045 installiert. Als Laufzeitumgebung wurde dotnet 8 einesetzt.

Zur messen der Ausführungszeit wurde die Klasse Stopwatch verwendet, die eine Menge an Methoden und Eigenschaften bereitstellt um verstrichene Zeit zu messen. Mithilfe der Methoden *Start()* und *Stop()* lässt sich die Stopuhr starten beziehungsweise beenden. Mit der Eigenschaft *ElapsedMilliseconds* lässt sich dann die gemessene Zeit der aktuellen Instanz als long ausgeben.

Hingegen zum messen des Speicherverbrauchs die Klasse GC verwendet wird, die eine Menge von Methoden und Eigenschaften bereitstellt um Speicher zu verwalten. GC hat nicht die Möglichkeit sofort den Speicherverbrauch zu messen, aber mithilfe der Methode *GetTotalMemory()* kann der Speicherverbraucht indirekt berechnet werden. *GetTotalMemory()* gibt nämlich die Heapgrößen ohne Fragmentierung zurück. Wird die Methode vor und nach einer Funktion aufgerufen und die Differenz aus den beiden Zahlen gebildet, dann hat man Speicherverbrauch der Funktion.

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.