

Contents

1	Einleitung	2
1.1	Aufgabe und Motivation	2
1.2	Aufbau der Arbeit	2
2	Terminologie	3
3	Systemanalyse	7
3.1	Grammatik	7
3.1.1	Prüfungslogik	7
3.1.2	Datenverarbeitungs	9
3.1.3	Typsystem	10
3.2	Ausführung	11
3.3	Codeanalyse	12
4	TODO	14
4.1	Anforderungsphase	14
4.2	Entwursphase	14
5	Implementierung	17
6	Evaluation	22
7	Literaturverzeichnis	23

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 konzipiert und implementiert werden, welche es ermöglichen sollen dynamischen 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 vorliegende Arbeit setzt sich aus TODO nachfolgenden Kapiteln zusammen. Im zweiten Kapitel TODO Anschließend daran wird im dritten Kapitel TODO Darauf aufbauen wird im vierten und fünften Kapitel TODO Der Fokus im letzten Kapitel liegt auf TODO

2 Terminologie

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 Algorithmus 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 Problem 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. Hingegen 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 die jeweilige Domaine relevant ist. Teilweise wird DSLs durch GPL ergänzt. [18] Bei DSLs wird zwischen externen und internen DSLs unterschieden. Externe DSLs haben ihre eigene Syntax. Dadurch kann eine größere flexibilität geschaffen werden, aber zeitlich 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. [7] Zur Laufzeit wird die externe DSLs dann in eine GPL übersetzt. [15] Interne DSLs hingegen verwenden die Syntax einer GPL und können 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 diese ausdrückstärker als Worte sind und einen höheren Wiedererkennungswert haben. Nachteile gegenüber Text-basierten Sprachen sind, dass VLs durch die eingeschränkte Syntax nicht so ausdrückstark und flexibel sind. [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 von VPLs besteht in der Verbesserung der Darstellung der Programmierlogik sowie in der Erleichterung des Verständnisses von Programmabläufen. [14] Dadurch soll der Fokus bei Programmieren stärker auf die konzeptuellen statt syntaktischen Aspekte verlagert werden. Die syntaktischen Aspekte werden von der Entwicklungsumgebung übernommen. [10] Die Umsetzung von Programmen erfolgt durch Flussdiagrammen, die vom Benutzer erstellt werden können. Die erstellten Flussdiagramme werden dann vom System interpretiert und ausgeführt. [13] Nach Charntaweekhun bieten Flussdiagramme einen didaktischen Vorteil, da diese Programmieranfängern ermöglichen, komplexe Abläufe visuell zu erfassen und zu strukturieren. [13] 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. [10] Die Stärken von VPLs zeigen sich darüber hinaus in ihrer Einfachheit, visueller Darstellbarkeit, Transparenz und Interaktivität. [14] Die Klassifikation visueller Programmiersprachen unterscheidet zwischen imperativen und deklarativen Modellen. Ersteres gibt die exakte Reihenfolge der Operationen vor, während letzteres lediglich Datenabhängigkeiten spezifiziert und die Ausführungsreihenfolge dem System überlässt. [12] 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 Funktionen weitergeleitet werden. Die Programme werden dabei als Graphen dargestellt. [11] 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. [1] Datenflussgraphen lassen sich hinsichtlich ihrer Granularität in feinkörnig und grobkörnig unterteilen. In einem feinkörnigen Graphen führt jeder Knoten exakt eine Instruktion 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 textbasiert sein. [1] Der Vorteil einer DLs ist, dass diese durch einen Graphen dargestellt werden können [11] und dadurch die Programme einfach zu verstehen sind. [5] Für komplexe Programme kann eine reine Graphendarstellung schnell unübersichtlich werden. Zur Strukturierung komplexer Programme werden Mikrofunktionen eingesetzt, bei denen einzelne Knoten auf untergeordnete Teilgraphen verweisen. Dadurch lassen sich dann auch rekursive Abläufe modellieren. [11] DLs führen Instruktionen nicht in einer festen Sequenz aus. Dadurch können unabhängig voneinander ausführbare Instruktionen 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 wissenschaftlichen Arbeit eine Menge von Eigenschaften. So sollen DLs frei von Seiteneffekten sein, den Lokalitätsprinzip folgen und keine Variablen überschreiben.[1]

Eine Computerarchitektur, die auf DLs basiert, wird Datenfluss-basiertes System (DBS) genannt. DBSs wurden eingeführt, um den Flaschenhals der von-Neumann-Architektur zu vermeiden. [8] Vorteile von DBSs sind eine hohe Effizienz, flexible Strukturen und leistungsstarke Ausführungsmechanismen. [6] Darüber hinaus besteht ein weiterer Vorteil im möglichen Parallelismus, da bei fehlender Datenabhängigkeit mehrere Instruktionen gleichzeitig ausgeführt werden können. [6] Dies wird durch die direkte Weitergabe von Daten zwischen Funktionen unterstützt, wobei die Verarbeitung der Daten, wie Transformation und Filterung, innerhalb der Funktionen erfolgt. [16] 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 wird die Funktion ausgeführt, sobald alle 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 vornherein besteht. [11]

Wasserfall Modell

Das Wasserfallmodell ist ein Modell für Softwareentwicklung [?] und wurde 1970 eingeführt. [?] Die Softwareentwicklung wird dabei in verschiedene voneinander getrennte Phasen unterteilt. [21] Es handelt sich beim Wasserfallmodell um ein

statisches Modell, da die einzelnen Phasen linear und sequentiell durchlaufen werden. [21] Sobald eine Phase abgeschlossen ist, kann diese nicht wieder besucht werden. [21] Aus dem Grund ist das Modell besonders gut für Systeme geeignet, welche nach der Implementierung keine Änderungen mehr zulassen [21]. Die Vorteile eines Wasserfallmodells sind, dass es einfach verständlich ist [21], 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 mittlere 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 Datentransformationen zuständig ist und eine von mehreren Aktionstypen darstellt. [?] Im Gegensatz dazu kümmert sich das Typsystem um die statische Analyse der Ausführbarkeit der vorangegangenen 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 eine Binärentscheidung oder eine Validierungsentscheidung sein. Ersteres besteht dabei aus einer Referenz auf ein Flowtemplate, das eine Datenverarbeitung 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 Hauptuntersuchungs-Adapter 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 Aktionsform, die Ausführung einer Datenverarbeitung, besteht aus einem Verweis auf ein 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 `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 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.

$$\langle \text{FlowInstance} \rangle ::= \langle \text{FlowOutputPort } \lambda \text{Arguments} \rangle^* \langle \text{FlowInputPort } \lambda \text{Arguments} \rangle^* \langle \text{FlowLambda} \rangle^*$$

$$\langle \text{FlowLambda} \rangle ::= \langle \text{FlowOutputPort } \lambda \text{Arguments} \rangle^* \langle \text{FlowInputPort } \lambda \text{Arguments} \rangle^*$$

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

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

Grammatik TODO Flow-Instanz

Ein Flowtemplate wird durch die in Abbildung TODO spezifizierten 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.

$$\langle \text{FlowTemplate} \rangle ::= \langle \text{Flow} \rangle \langle \text{TemplateParameter} \rangle^*$$

$$\langle \text{Flow} \rangle ::= \langle \text{LibraryFlow} \rangle \mid \langle \text{FlowModel} \rangle$$

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

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

$$\langle \text{TemplateParameterList} \rangle ::= \langle \text{TemplateParameter} \rangle$$

Grammatik TODO Flow-Template

Die Regeln des FlowModel sind in Abbildung TODO formal dargestellt. Das FlowModel entspricht dabei der Datenverarbeitung. [?] Durch die Regel *Flow-Model* wird bestimmt, dass ein FlowModel aus einer FlowInstance, einer Auflistung aller genutzten 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.

$$\begin{aligned}
\langle FlowModel \rangle &::= \langle FlowInstance \rangle \langle FlowNode \rangle^* \langle FlowConnection \rangle^* \\
\langle FlowNode \rangle &::= \langle FlowNodeOutput \rangle \mid \langle FlowNodeInput \rangle \mid \langle FlowNodeLambda \rangle \mid \\
&\quad \langle FlowNodeFlowCall \rangle \\
\langle FlowNodeOutput \rangle &::= \text{ref}(\text{FlowOutputPort}) \\
\langle FlowNodeInput \rangle &::= \text{ref}(\text{FlowInputPort}) \langle FlowPortValue \rangle \\
\langle FlowNodeLambda \rangle &::= \text{ref}(\text{FlowLambda}) \langle FlowPortValue \rangle^* \\
\langle FlowNodeFlowCall \rangle &::= \text{ref}(\text{FlowTemplate}) \langle FlowPortValue \rangle^* \langle TemplateParameterValue \rangle^* \\
\langle FlowConnection \rangle &::= \text{ref}(\text{FlowOutputPort source}) \text{ref}(\text{FlowOutputPort target}) \\
\langle TemplateParameterValue \rangle &::= \langle string \rangle \mid \langle number \rangle \mid \langle bool \rangle \mid \langle TemplateParameterValueList \rangle \\
\langle TemplateParameterValueList \rangle &::= \langle TemplateParameterValue \rangle^*
\end{aligned}$$

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 Primitiven-Datentypen stehen String, Number und Bool zur Auswahl. Zusätzlich noch Data und PtiaResponse. Ersteres ist eine Datumsanagbe. Hingegen PtiaResponse die Antwort von einem Hauptuntersuchungs-Anfrage ist.

$$\begin{aligned}
\langle Type \rangle &::= \langle TypePrimitive \rangle \mid \langle TypeOptional \rangle \mid \langle TypeList \rangle \mid \langle TypeObject \rangle \\
\langle TypePrimitive \rangle &::= \text{'String'} \mid \text{'Number'} \mid \text{'Bool'} \mid \text{'Data'} \mid \text{'PtiaResponse'} \\
\langle TypeOptional \rangle &::= \langle Type \rangle \text{'?' } \\
\langle TypeList \rangle &::= \langle Type \rangle \text{'[]'} \\
\langle TypeObject \rangle &::= \text{'{' } (\langle string key \rangle \text{' ':' } \langle Type \rangle)^* \text{' '}}
\end{aligned}$$

$\langle TypeGeneric \rangle ::= '\$' \langle string genericName \rangle$

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

Grammatik TODO Typ-Definition 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 unveröffentlichten 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 Aktivität vorgegeben, da jede Aktivität das Label der zu folgenden 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 implementiert. 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 zugehörigen InputNode sowie der entsprechenden eingehenden 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 Funktionen 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 Ergebnis aus dem Ergebniscache gelöscht wird. Anschließend wird der oben beschriebene Algorithmus verwendet, um die Werte für die Eingabeport der Lambda-Funktion zu berechnen.

TypSystem

Bevor die Typvalidierung bei Flow-Modellen 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 konkrete Typen ersetzt. Nach dem die Ersetzung abgeschlossen ist, wird sich um die generischen Typen gekümmert. Dafür werden zunächst die Flow-Modelle topologisch sortiert und pro Flow-Modell über alle Verbindungen iteriert. Kommt bei einem Port ein generischer 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 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 Aktivitätsmodells 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 Zyklen

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 Dictionary 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 topologischen sortierten Liste wird eine erneute Tiefensuche durchgeführt, die für jeden erreichbaren Knoten einen Komponent mit der gemeinsamen Wurzel (root) erstellt. Dabei werden Komponenten als erstes für alle Vorgänger Knoten erstellt. Nun wurden alle zusammenhängende Elemente des Graphen gefunden und die zusammenhängende Elemente können in einem Dictionary übertragen werden. Dabei wird der root der Schlüssel eines Dictionary Elements sein und der Value der Knoten. Hat ein Dictionary Element mehr als ein Element im Value ist ein Zyklus vorhanden und diese Dictionary Element wird in einem anderen Dictionary gespeichert, welches zum erstellen von Fehlermeldungen verwendet wird

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

P2 -

Beim betrachten 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 dynamischen Daten zu erstellen. Die Interaktion erfolgt über die bereits vorhandene grafische Benutzeroberfläche, in dieser kann der Benutzer die Anzahl der Iterationen und die Abbruchbedingung festlegen. Beide Blöcke sind von der Benutzeroberfläche gleich 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 Abbruchbedingung 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 Entwurfsphase

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 Verhalten 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 Instruktionen im Schleifenkörper mehrmals pro Iteration auszuführen wird, um dadurch die Häufigkeit der Iterationen zu verringern. [9] Anstatt rekursive oder iterative Wiederholungen zu verwenden, wird jede potenzielle Iteration als eigenständiger und bedingter Codeblock realisiert. Dazu muss eine Anpassung an der klassischen Schleifenentfaltungsvorgehen 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 Verschaltungen ohne Iterationen abzubilden. Da die maximale Anzahl an Schleifendurchläufen von vornherein bekannt ist, kann für jede potenzielle Ausführung 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 ausgeführt 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 Dezimalzahl 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 Addition 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 letzten 3 Werte. Anschließend wird die Differenz zwischen dem aktuellen Wert und den Mittelwerten gebildet. Ist die Differenz kleiner als ein Threshold wird der Counter dekrementiert. 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 Ergebnisliste drin sein, wird die Berechnung übersprungen. Dadurch kann gewährleistet 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 vorangegangenen Kapitel dienen insbesondere Kapitel TODO. Das Ziel ist es die Implementierung verständlich und nachvollziehbar zu machen.

Zunächst wurde Klasse `DataModelActivityElement` um das Attribut *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 einzuzichnen. 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 Vererbung die Methoden und Variablen der Basisklasse `DataModelActivityElement` zur Verfügung. 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 iteriert 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 jedem 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 gefundenen 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 `visited` erstellt wurden ist, iteriert und für jedes Element aus dem Tupel das dazugehörige Object aus der Liste aller Elementen selektiert. Die zwei selektierten 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 zuständig. Dazu wird das übermittelte TODO in serialisiert und anschließend deserialisiert. Die beiden Methoden werden von der Klasse `JsonSerializer` bereitgestellt. Durch die Deserialisierung 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 iteriert 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 Element 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 überprüfen. Zur Umsetzung dieser Funktionalität hat DeviationChecker eine Mittelwert-Variable vom Datentyp Double sowie die drei Methoden `calculate`, `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`

konvertiert 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 weitere Verarbeitung zulässig wird zuerst `TODO` und der Eingabewert mithilfe der Methode `storeInput` in eine Liste gespeichert. Anschließend wird der Eingabewert 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 konvertiert. Als Rückgabewert wird der `Double` zurückgegeben.

Mithilfe der Methode `validateDeviation` wird überprüft, ob der Eingabewert eine signifikante 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 resettet. Ist hingegen keine Abweichung vorhanden wird der Zähler dekrementiert.

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 rekursiven 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 Ele-

mente 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 eingesetzt.

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 Speicherverbrauch 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] Lućanin, D., & Fabek, I. (2011). *A visual programming language for drawing and executing flowcharts*. In 2011 Proceedings of the 34th International Convention MIPRO (pp. 1679–1684).
- [11] Davis, A., & Keller, R. (1982). *Data Flow Program Graphs*. All HMC Faculty Publications and Research.
- [12] Boshernitsan, M., & Downes, M. (2004). *Visual Programming Languages: A Survey*. EECS University of California, Berkeley.
- [13] Charntaweekhun, K., & Wangsiripitak, S. (2006). *Visual Programming using Flowchart*. In 2006 International Symposium on Communications and Information Technologies (pp. 1062–1065).

- [14] Burnett, M., Baker, M., Bohus, C., Carlson, P., Yang, S., & Van Zee, P. (1995). *Scaling up visual programming languages*. *Computer*, 28(3), 45–54.
- [15] Kurihara, A., Sasaki, A., Wakita, K., & Hosobe, H. (2015). *A Programming Environment for Visual Block-Based Domain-Specific Languages*. *Procedia Computer Science*, 62, 287–296.
- [16] Hils, D. (1992). *Visual languages and computing survey: Data flow visual programming languages*. *Journal of Visual Languages & Computing*, 3(1), 69–101.
- [17] Sousa, T. (2012). *Dataflow Programming Concept, Languages and Applications*. *Doctoral Symposium on Informatics Engineering*, 7.
- [18] Van Deursen, A., Klint, P., & Visser, J. (2000). *Domain-specific languages: an annotated bibliography*. *ACM SIGPLAN Notices*, 35(6), 26–36.
- [19] Roy, G., Kelso, J., & Standing, C. (1998). *Towards a visual programming environment for software development*. In *Proceedings. 1998 International Conference Software Engineering: Education and Practice* (Cat. No.98EX220) (pp. 381–388). *IEEE Comput. Soc.*
- [20] Weintrop, D. (2019). *Block-based programming in computer science education*. *Communications of the ACM*, 62(8), 22–25.
- [21] Gumm, H.P., & Sommer, M. (2016). *Band 1 Programmierung, Algorithmen und Datenstrukturen*. *De Gruyter Oldenbourg*.