

Inhaltsverzeichnis

1	Einleitung	2
1.1	Aufgabe und Motivation	2
1.2	Aufbau der Arbeit	2
2	Hintergrund	3
2.1	Terminologie	3
2.2	Domaine	6
3	Systemanalyse	7
3.1	Grammatik	7
3.1.1	Prüfungslogik	7
3.1.2	Datenverarbeitung	10
3.1.3	Typsystem	11
3.2	Ausführung	12
3.3	Codeanalyse	14
4	Planung	15
4.1	Anforderungen	15
4.2	Entwurf	15
5	Implementierungsphase	19
6	Evaluation	24
6.1	Testumgebung	24
7	Ausblick	25
8	Literaturverzeichnis	26

1 Einleitung

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, weil 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, weil 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.

In dieser Arbeit sollen mehrere Möglichkeiten konzipiert und implementiert werden, welche es ermöglichen sollen dynamische Daten innerhalb einer Prüfung zu verarbeiten. Bei dynamischen Daten handelt es sich um Daten, die während der Prüfung ihren Wert ändern. Dafür muss im Vorfeld das zugrunde liegende System analysiert und bearbeitet werden, sodass Zyklen vom System verarbeitet werden können. Die Implementierung wird in einer Evaluation auf Performance untersucht.

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 Hintergrund

Der Fokus dieses Kapitels liegt auf der Definition zentraler Fachbegriffe und der Beschreibung der Domäne 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.

2.1 Terminologie

TODO zitat zur rekursion nochmal genau anschauen

Schleife 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 Domäne 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 dann die externe DSLs in eine GPL übersetzt. Bei internen DSLs hingegen wird die Syntax einer GPL verwendet. Die interne

DSL kann dann oft über eine Programmierschnittstelle oder Bibliothek aufgerufen werden [15]. Ein Beispiel für eine externe DSL ist SQL. Ein Beispiel für eine interne DSL ist Rake. Die Vorteile von DSLs 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 Metaphan verwendet. Bilder eignen sich besonders gut zum Programmieren, weil diese ausdrucksstärker als Worte sind und einen höheren Wiedererkennungswert haben. Nachteile gegenüber Text-basierten Sprachen sind, dass VLs durch die eingeschränkte Syntax nicht so ausdrucksstark 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 Charntaweehkhun bieten Flussdiagramme einen didaktischen Vorteil, weil 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, visuelle 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 basieren überwiegend auf einem datenflussgesteuerten Modell, bei dem die Strukturierung von Programmen durch den Austausch von Informationen zwischen Operatoren erfolgt. [5]

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 text-basiert 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 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 ihren Arbeiten eine Menge von grundlegenden Eigenschaften. So sollen DLs frei von Seiteneffekten sein, den Lokalitätsprinzip folgen und keine Variablenüberschreiben zulassen [1]. Frei von Seiteneffekten bedeutet dabei, dass Ausführungen nur von den Eingaben abhängen. Das Lokalitätsprinzip besagt, dass bevorzugt auf Speicherbereiche zugegriffen wird, die entweder räumlich nahe beieinanderliegen (räumliche Lokalität) oder innerhalb kurzer Zeit mehrfach verwendet werden (zeitliche Lokalität).

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 vorneherein besteht [11].

Wasserfallmodell Das Wasserfallmodell ist ein Modell für Softwareentwicklung [?] und wurde 1970 von Winston W. Royce definiert [?]. Die Softwareent-

wicklung 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 [?].

2.2 Domaine

Bei dem Programm handelt es sich um einen graphischen Editor der Fahrzeug-systemdaten GmbH (FSD). Die FSD erstellt Prüfvorgaben für die Hauptuntersuchung. Der Benutzer hat die Möglichkeit per Drag-and-Drop seine Prüfung zusammen zu bauen. Dabei stehen dem Benutzer eine Vielzahl von vordefinierten Blöcken zur Verfügung. In Abbildung TODO ist ein Auszug der fertigen Software abgebildet. Zur besseren Verständlichkeit wurde die Abbildung in Teilbereiche unterteilt, welche im folgenden kurz erläutert werden. Bereich B umfasst das Canvas, das als interaktive Arbeitsfläche fungiert. TODO

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

TODO Grammatik definieren.

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 1 abgebildeten Regeln beschreiben 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 TODO.

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 Daten-

verarbeitung 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.

$$\begin{aligned}
\langle \textit{ActivityModel} \rangle &::= \langle \textit{Activity} \rangle^* \langle \textit{ActivityConnection} \rangle \\
\langle \textit{Activity} \rangle &::= \langle \textit{ActivityStart} \rangle \mid \langle \textit{ActivityAction} \rangle \mid \langle \textit{ActivityCondition} \rangle \mid \\
&\quad \langle \textit{ActivityDisplay} \rangle \\
\langle \textit{ActivityConnection} \rangle &::= \text{ref}(\textit{Activity source}) \langle \textit{string label} \rangle \text{ref}(\textit{Activity target}) \\
\langle \textit{ActivityStart} \rangle &::= \epsilon \\
\langle \textit{ActivityAction} \rangle &::= \langle \textit{ActivityFlowCall} \rangle \mid \langle \textit{ActivityPitaBuildInforRequest} \rangle \mid \\
&\quad \langle \textit{ActivityLoadExternalData} \rangle \\
\langle \textit{ActivityFlowCall} \rangle &::= \text{ref}(\textit{FlowTemplate}) \langle \textit{ActivityPortValue} \rangle^* \\
&\quad \langle \textit{TemplateParameterValue} \rangle^* \langle \textit{ValueTransformation} \rangle^* \\
\langle \textit{ActivityPitaBuildInforRequest} \rangle &::= \langle \textit{string abdFilename} \rangle \langle \textit{string requestAlias} \rangle \\
&\quad \langle \textit{string expectedSystems} \rangle^* \langle \textit{number timeout} \rangle \\
\langle \textit{ActivityLoadExternalData} \rangle &::= \langle \textit{Type dataType} \rangle \langle \textit{string dataSource} \rangle \\
\langle \textit{ActivityPortValue} \rangle &::= \langle \textit{FlowPortValue} \rangle \mid \langle \textit{ActivityPortRefernce} \rangle \\
\langle \textit{FlowPortValue} \rangle &::= \langle \textit{string} \rangle \mid \langle \textit{number} \rangle \mid \langle \textit{bool} \rangle \mid \langle \textit{date} \rangle \mid \langle \textit{FlowPortValue} \rangle^* \\
\langle \textit{ActivityPortRefernce} \rangle &::= \text{ref}(\textit{ActivityAction}) \langle \textit{ValueTransformation} \rangle^* \\
\langle \textit{ValueTransformation} \rangle &::= \langle \textit{string objectReference} \rangle \mid \langle \textit{number listIndex} \rangle \\
\langle \textit{ActivityCondition} \rangle &::= \langle \textit{ActivityBinaryCondition} \rangle \mid \\
&\quad \langle \textit{ActivityValidityCondition} \rangle \\
\langle \textit{ActivityBinaryCondition} \rangle &::= \text{ref}(\textit{FlowTemplate}) \\
&\quad \langle \textit{ActivityBinaryConditionOperator} \rangle \langle \textit{ActivityPortValue left} \rangle \\
&\quad \langle \textit{ActivityPortValue right} \rangle \\
\langle \textit{ActivityValidityCondition} \rangle &::= \langle \textit{ActivityPortValue} \rangle^* \\
\langle \textit{ActivityBinaryConditionOperator} \rangle &::= '=' \mid '\neq' \mid '<' \mid '\leq' \mid '>' \mid '\geq' \\
\langle \textit{ActivityDisplay} \rangle &::= \langle \textit{ActivityDisplayField} \rangle^* \\
\langle \textit{ActivityDisplayField} \rangle &::= \langle \textit{string label} \rangle \langle \textit{string color} \rangle \text{ref}(\textit{ActivityAction})
\end{aligned}$$

Abbildung 1: Aktivitätsmodell

3.1.2 Datenverarbeitung

Die in Abbildung 2 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.

$$\begin{aligned}
\langle FlowInstance \rangle &::= \langle FlowOutputPort\ lambdaArguments \rangle^* \\
&\quad \langle FlowInputPort\ lambdaArguments \rangle^* \langle FlowLambda \rangle^* \\
\langle FlowLambda \rangle &::= \langle FlowOutputPort\ lambdaArguments \rangle^* \\
&\quad \langle FlowInputPort\ lambdaArguments \rangle^* \\
\langle FlowInputPort \rangle &::= \langle string\ name \rangle \langle Type \rangle \langle bool\ acceptsError \rangle \\
\langle FlowOutputPort \rangle &::= \langle string\ name \rangle \langle Type \rangle \langle bool\ producesError \rangle
\end{aligned}$$

Abbildung 2: Flow-Instanz

Ein Flow-Template wird durch die in Abbildung 3 spezifizierten Regeln beschrieben. Das Flow-Template 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.

$$\begin{aligned}
\langle FlowTemplate \rangle &::= \langle Flow \rangle \langle TemplateParameter \rangle^* \\
\langle Flow \rangle &::= \langle LibraryFlow \rangle \mid \langle FlowModel \rangle \\
\langle LibraryFlow \rangle &::= \epsilon \\
\langle TemplateParameter \rangle &::= 'String' \mid 'Number' \mid 'Bool' \mid \\
&\quad \langle TemplateParameterList \rangle \\
\langle TemplateParameterList \rangle &::= \langle TemplateParameter \rangle
\end{aligned}$$

Abbildung 3: Flow-Template

Die Regeln des Flow-Model sind in Abbildung 4 formal dargestellt. Das Flow-Model entspricht dabei der Datenverarbeitung. [?] Durch die Regel *FlowModel* 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.

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

Abbildung 4: Flow-Modell

3.1.3 Typsystem

In Abbildung 5 werden die Regeln des Typsystem dargestellt. Das Typsystem unterstützt Primitive- und Generische-Datentypen, welche an einen Optional-, Listen- oder Objekt-Typen gekapselt werden können. Als Primitiven-Datentypen stehen **String**, **Number**, **Bool**, **Data** und **PtiaResponse** zur Auswahl. Ersteres ist eine Datumsanagbe. Hingegen PtiaResponse die Antwort von einem Hauptuntersuchungs-Anfrage ist.

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

Abbildung 5: Typ-Definition mit gernischen und Referenz-Typen

3.2 Ausführung

Im folgenden Unterkapitel wird gezeigt, wie die einzelnen Ebenen ausgeführt werden. Die folgende Zusammenfassung basiert auf der 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 soeben ausgeführte Aktivität 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 Referenzstack 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 zugehörigen Eingabeport 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 zuerst die Referenztypen durch konkrete Typen ersetzt. Nachdem 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 Typzuweisung gespeichert. Es wird probiert, über die anderen Typzuweisungen 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-Nodes 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 nicht das Schleifen im Graphen vorkommen und ausgeführt werden können. Konkret lassen sich zwei Probleme aus der aktuellen Umsetzung ableiten:

- P1 Die Modellanalyse erlaubt keine Zyklen im Graphen
- P2 Jeder Knoten kann nur einmal ausgeführt werden

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 zuerst 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 zunächst der Liste hinzugefügt. Dadurch erfolgt ein topologisches Sortieren der einzelnen Knoten. Auf der topologischen sortierten Liste wird dann eine erneute Tiefensuche durchgeführt, die für jeden erreichbaren Knoten einen Eintrag mit der gemeinsamen Wurzel (root) erstellt. Dabei werden die Einträge 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 des Dictionary Elements sein und der Value der Knoten. Hat ein Dictionary Element mehr als ein Element im Value ist ein Zyklus vorhanden und dieses Element wird dann in einem anderen Dictionary gespeichert, welches zum Erstellen von Fehlermeldungen verwendet wird. Aber dies ist auch das Problem. Es zeigt, wenn ein Zyklus erlaubt ist, darf das zugehörige Key-Value-Paar nicht in das Dictionary für die Fehlermeldung gespeichert werden, sondern dieser Schritt muss übersprungen werden.

P2 - Jeder Knoten kann nur einmal ausgeführt werden

Die Klasse *VirtualMachine* verhindert aktuell, dass Blöcke mehrmals aufgerufen werden. Dies geschieht indem für alle auszuführenden Blöcke ein Eintrag in einer Liste vom Datentyp Bool angelegt wird. Jeder Eintrag hat standardmäßig den Wert false. Wird ein Block nun ausgeführt, wird der dazugehörige Wert in der Liste auf true gesetzt. Bevor eine Funktion ein Block ausgeführt wird, wird geschaut, welcher Wert in der Liste ist. Ist dies der Fall, wird der Block übersprungen und mit der nächsten auszuführenden Funktion weitergemacht. Hingegen beim Wert false wird die Funktion der Klasse *VirtualMachineCounter* hinzugefügt. Dort liegt auch das Problem. TODO

4 Planung

In dem Kapitel wird das Design der geplanten Erweiterung beschrieben. Dafür werden zunächst die Anforderungen erklärt und diese im Anschluss spezifiziert. Die beiden Punkte ähneln dabei der Anforderungs- und Entwurfsphase des Wasserfallmodells.

4.1 Anforderungen

In diesem Unterkapitel werden die Anforderungen an die geplante Erweiterung konkretisiert.

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 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 wurde ist, kann der Benutzer mit dem Schleifenblock interagieren. Die Schleifenlogik soll durch den Benutzer konfigurierbar sein. Dafür wird auf die bereits vorhandene grafische Benutzeroberfläche des Systems zurück gegriffen.

4.2 Entwurf

Im folgenden Unterkapitel wird der Entwurf beschrieben. Dabei werden auf die Punkte aus dem Kapitel 4.1 eingegangen und im Anschluss die Algorithmen entworfen.

Schleifenblock

Für die Integration der neuen Schleifenblöcke ist eine Anpassung der Grammatik der Prüfungslogik erforderlich, konkret an der Regel *Activity*. Hier muss die Regel um eine Alternative erweitert werden, die ein neues Nichtterminale *ActivityLoop* einführt. *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 Nichtterminal wurde bereits in Kapitel 3.1.1 definiert. Dargestellt wird die Erweiterung in Abbildung 6. Auch in diesem Fall beinhaltet die Erweiterung nur die für die Ausführungssemantik benötigten Elemente.

$$\begin{aligned}
\langle Activity \rangle &::= \langle ActivityStart \rangle \mid \langle ActivityAction \rangle \mid \langle ActivityCondition \rangle \mid \\
&\quad \langle ActivityDisplay \rangle \mid \langle ActivityLoop \rangle \\
\langle ActivityLoop \rangle &::= \langle integer \ iteration \rangle \langle ActivityBinaryCondition \rangle \\
\langle ActivityBinaryCondition \rangle &::= \text{ref(FlowTemplate)} \\
&\quad \langle ActivityBinaryConditionOperator \rangle \quad \langle ActivityPortValue \ left \rangle \\
&\quad \langle ActivityPortValue \ right \rangle \\
\langle ActivityBinaryConditionOperator \rangle &::= '=' \mid '\neq' \mid '<' \mid '\leq' \mid '>' \mid '\geq'
\end{aligned}$$

Abbildung 6: Erweiterung Prüfungslogik

Der Schleifenblock soll dabei als Oval im Canvas dargestellt werden. Beim Klicken auf den Schleifenblock soll der Benutzer die Möglichkeit haben die Anzahl der Iterationen und die Abbruchbedingung für die Schleife festzulegen. Zusätzlich TODO. Die Abbruchbedingung setzt sich dabei aus zwei Feldern für die Argumente und einer Auswahlliste für den Operator zusammen. Am Block selbst kann der Benutzer die Verbindungen erstellen, indem er per Drag-and-Drop die einzelnen Interaktionspunkte verschiedener Aktivitäten verbindet. Rückverbindungen über den Schleifenblock sollen dabei erlaubt sein. Insgesamt 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 unten vom Interaktionspunkt (OnTrue) genommen. Wenn hingegen die Abbruchbedingung Falsch ist, wird die Verbindung rechts vom Interaktionspunkt (OnFalse) genommen.

Block A

Block A soll die Wiederholung von Aktivitäten durch explizite Ausführung ermöglichen und wird durch SchleifeB abgebildet. 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 Instruktionen in der Schleife mehrmals pro Iteration auszuführen werden, um dadurch die Häufigkeit der Iterationen zu verringern [9]. Um das Ziel zu erreichen, muss die Schleifenentfaltung abgeändert werden. Die Schleifenentfaltung soll nicht nur die 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 Schleifenkörpers 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 Aktivität einer Kopie eine Verbindung zur vorangegangenen Schleifenblock hergestellt wird. Graphisch wird dieser Ansatz in Abbildung 7 dargestellt.

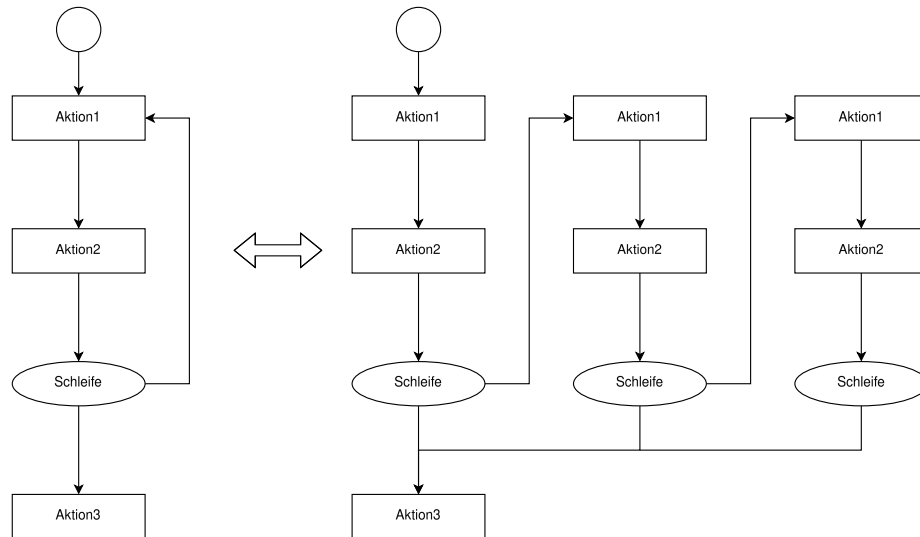


Abbildung 7: Schleifenentfaltung

Block B

Block B soll die Wiederholung von Aktivitäten durch iterative Schleifen ermöglichen und wird durch SchleifeA abgebildet. 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 Iterationen 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 letzten 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, weil 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 enthalten sein, wird die Berechnung der Differenz übersprungen und direkt mit der Wiederholung der Aktivitäten begonnen. Dadurch kann gewährleistet werden, dass das System einlaufen kann und die Mittelwerte erst gebildet werden, wenn eine aussagekräftige Datenbasis vorhanden ist.

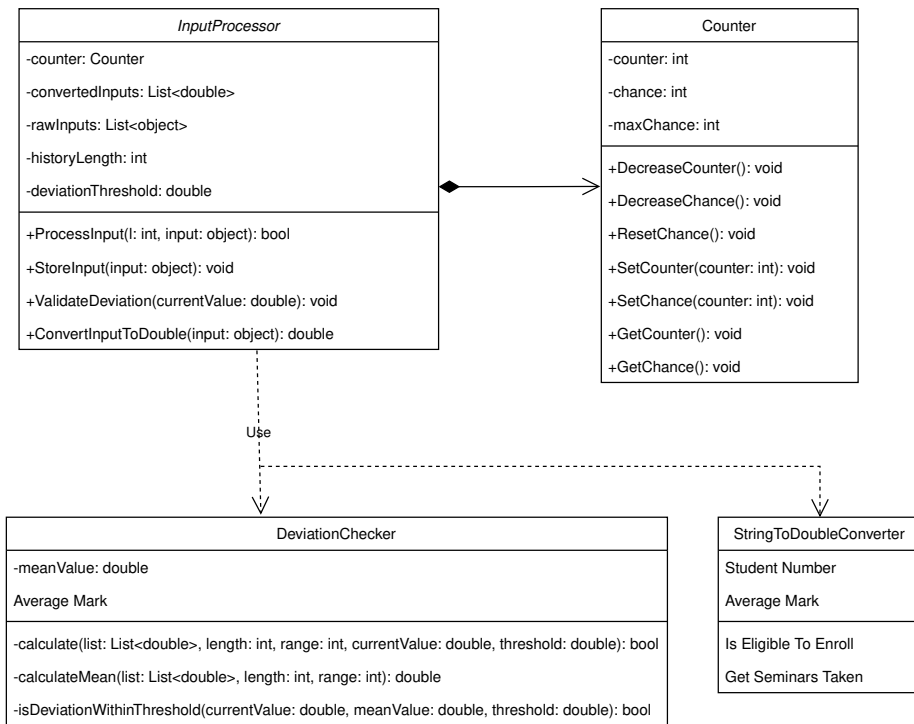


Abbildung 8: Klassendiagramm Block B

Auf die Vor- und Nachteile der einzelnen Blöcke wird im Kapitel 6 eingegangen.

5 Implementierungsphase

In diesem Kapitel wird die Implementierungsphase beschrieben. Dabei werden die in der Entwurfsphase erarbeiteten Konzepte in das System implementiert. [?] Im Folgenden werden die vorgenommenen Änderungen am Quellcode beschrieben und erläutert.

Zunächst wurde Klasse *DataModelActivityElement* um das Attribut *CreateLoop* 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 Benutzer die Möglichkeit hat von dem Block ausgehend eine Rückverbindung einzuzichnen. Aus dem Grund ist das Attribut für alle Aktivitäten bis auf die SchleifeA und SchleifeB *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, wurden durch die strings **OnLoop** und **OnFinish** erweitert. **OnLoop** ist dabei der *OnFalse* Interaktionspunkt und *Onfinish* der *OnTrue* Interaktionspunkt.

Eine weitere Änderung wurde an der Klasse *VirtualMachineCompilerActivity* vorgenommen. *VirtualMachineCompilerActivity* ist zuständig für die Kompilierung der Aktivitäten. Dafür wird von jeder Aktivität ein Objekt mit allen benötigten Informationen zur Ausführung (Statement) erstellt. Da für SchleifeA und SchleifeB noch keine Möglichkeit besteht Statements zu erstellen, müssen Änderungen an den bestehenden Methoden vorgenommen werden. Bei der Methode *CreateActivityFunctions()* muss die Switch-Anweisung um zwei weitere Fälle erweitert werden. Da SchleifeA und SchleifeB Verzweigungsblöcke sind, muss der dritte Parameter der Methode auf *true* gesetzt werden, damit für beiden Schleifen ein eigener Funktionsblock erstellt wird. Dieses Statements werden dann in der Klasse *VirtualMachineActivity* ausgeführt. Die in diesem Absatz beschriebenen Änderungen wurden durch Westermann implementiert.

Zur Umsetzung der Wiederholungslogik im Block A 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 bestimmen.

Außerdem leitet *UnrollLoop()* die Schleifenentfaltung ein. Dafür stehen der Methode die Liste aller Elemente und zwei Variablen für den Schleifenanfang und das Schleifenende, die jeweils das konkrete Element speichern, zur Verfügung. Um die beiden Variablen zu konkretisieren, wird zuerst über die Liste iteriert und nachdem Element gesucht mit dem Attribut `classType = "LoopB"`. Wurde ein Element gefunden, wird dieses in die Variable für das Schleifenende gespeichert. Um den Schleifenanfang zu finden, kann das Schleifenende verwendet werden. Mithilfe des Attributs `OnFinishedSuccessor` kann die Id des Schleifenanfangs herausgefunden werden und genutzt werden um das konkrete Element zu bekommen. Dafür wird erneut über die Liste der Elemente iteriert und jede Id mit der Id des Schleifenanfangs verglichen. Sobald eine Übereinstimmung gefunden wurde ist, wird dieses Element abgespeichert. Wenn die beiden Variablen bestimmt und nicht Null sind, kann die Methode *getLoopElements()* aufgerufen werden.

getLoopElements()

TODO Die Methode *visit()* führt eine rekursive Tiefensuche ab dem Schleifenanfang im Graphen durch. Dabei wird jeder Knoten aus einem HashSet hinzugefügt und für jeden nachfolgenden Knoten *visit()* erneut aufgerufen. Die Ausnahme dabei ist dabei das Schleifenende. Handelt es sich bei dem nachfolgenden Knoten um das Schleifenende, wird das Schleifenende dem HashSet hinzugefügt, aber *visit()* wird nicht für dieses Element aufgerufen. Das hat dem Grund, da die nachfolgenden Knoten ab dem Schleifenende irrelevant für den Algorithmus sind.

Die Hilfsmethode *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 Organisation des Klonvorgangs zuständig. Um dies zu realisieren steht der Methode ein HashSet für die geklonten Elemente zur Verfügung. Zunächst wird über das HashSet mit allen Elementen aus der Schleife iteriert. Für jedes Element wird mithilfe von *clone()* eine exakte Kopie erstellt. Die Kopie wird anschließend der Kopie des Datenmodells angehängt und daraufhin einer neuem HashSet hinzugefügt. Das neu erstellte HashSet dient als Parameter für den rekursiven Aufruf und beinhaltet nur die Kopien. Sobald die Iteration abgeschlossen ist, wird überprüft ob die Anzahl der Iteration bereits kleiner ist als zwei. Ist das der Fall wird Zähler für die Anzahl der Iterationen um eins verringert und *cloneAndModify()* erneut aufgerufen.

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 Schleifenrumpfs erhalten.

Zur Umsetzung der Wiederholungslogik im Block B wurden vier Klassen *DeviationChecker*, *Counter*, *StringToDoubleConverter* und *InputProcessor* definiert. Die Klasse *DeviationChecker*, die in Abbildung TODO dargestellt ist, hat das Ziel 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.

Mithilfe 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 *calculateMean()* aufgerufen und dessen Rückgabewert in die Mittelwert-Variable gespeichert. Im Anschluss erfolgt der Aufruf von *isDeviationWithinThreshold()*, dessen Rückgabewert von *calculate()* zurückgegeben wird.

Eine 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.

Abbildung 9: Beispielrechnung der Methode `StringToDouble` mit und ohne Index-Gewichtung

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 Unterkapitel 3.3 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 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

in diesen Kapitel werden die Ergebnisse der durchgeführten Tests beschrieben. Dafür wird am Anfang das zu auszuführende System definiert und im Anschluss die verschiedenen Szenarien mit welchen das System getestet wurden ist. Am Ende werden die Ergebnisse interpretiert und ein Fazit gezogen.

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 ist Windows 10 in der Version 10.0.19045 Build 19045 installiert. Als Laufzeitumgebung wurde .Net 8 eingesetzt.

Zum messen der Ausführungszeit wurde die Klasse Stopwatch verwendet, die eine Menge an Methoden und Eigenschaften bereitstellt um verstrichene Zeit zu messen. Die Klasse kommt aus der standard Bibliothek von .Net. 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. Auch diese Klasse stammt aus der standard Bibliothek von .Net. GC hat nicht die Möglichkeit sofort den Speicherverbrauch zu messen, aber mithilfe der Methode *GetTotalMemory()* kann der Speicherverbrauch indirekt berechnet werden. *GetTotalMemory()* gibt die Heapgröße ohne Fragmentierung zurück. Wird die Methode vor und nach einer Funktion aufgerufen und die Differenz aus den beiden Zahlen gebildet, dann ergibt sich der Speicherverbrauch der Funktion.

6.1 Testumgebung

Als Testumgebung wurde eine Prüfung erstellt, TODO Für den Test wurden insgesamt 1024 Werte definiert. Im jeden Schleifendurchlauf wird TODO Damit eine Aussage über das Verhalten der Algorithmen getroffen werden kann, wurden die Anzahl an Iterationen am Schleifenblock stückweise erhöht. Zunächst wurden die Anzahl um vierer Schritten erhöht. Hingegen ab 64 die Schritte auf 16 erhöht wurden sind. Insgesamt wurde der Speicherverbrauch und die Ausführungszeit an TODO Stellen gemessen. Argument1 wird dabei als dynamisches Datum festgelegt und Argument2 als konstanten Wert.

7 Ausblick

In diesem Kapitel werden mögliche Weiterentwicklung und zukünftige Forschungsarbeiten beschrieben. Die erst im späten Verlauf aufgetreten sind oder aus Zeitgründen nicht bearbeitet werden konnten.

Der Threshsold für den Algorithmus Block B wird aktuell fest in den Code implementiert werden. Das sollte in der Praxis so nicht sein. Hier würde es sich anbieten einen Algorithmu zu entwerfen der den Threshold dynmaisich anpasst jenachdem welche Eingabe er bekommt.

Ein anderer Punkt, ist dass derBneutzer aktuell kein Feedback bekommt, ob ein Ergebniss gefunden wurden ist oder nicht. Um das zu verbessern könnte eine Meldung ins System implementiert werden, wenn der Algorithmus nicht durchgeht, dass der Benutzer eine Meldung bekommt, dass keine Lösung gefunden werden konnte.

Ein anderer Punkt der verbessert werden kann, ist die Robustheit des Block A. Dieser führt aktuell bei einer großen Anzahl an Iterationen zu einen Overflow.
aus

8 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.