

folgende Begriffe sollen definiert werden:

Visual Programming Language

Grammatik

Domain Specific Language

Schleifen

Flowchart

(Fixpunktberechnung)

Contents

1	Einleitung	3
2	Grammatikanalyse	4
2.1	Prüfungslogik	4
2.2	Datenverarbeitungs	6
2.3	Typsystem	8
3	Implementierung	9
3.1	1. Lösungsansatz	9
3.2	2. Lösungsansatz	9
4	Evaluation	11
4.1	11
4.1.1	1. Lösungsansatz	11

1 Einleitung

test

2 Grammatikanalyse

Im nachfolgenden Kapitel möchte ich die zugrundeliegende Grammatik beschreiben. Am Anfang möchte ich auf die Notation eingehen.

Die zugrundeliegende Grammatik basiert auf der Backus-Naur-Form Notation.

$\langle symbol \rangle ::= _expression_$

$\langle symbol \rangle$ sind nichtterminale $::=$ bedeutet dass $\langle symbol \rangle$ durch $_expression_$ ersetzt wird $_expression_$ ist eine sequenze von nichtterminalen und terminale Kleene-Stern $*$ wiederholung Alternation $|$ oder Sequenz erlaubt auch Klammern um die Reihenfolge der Regel zu definieren

2.1 Prüfungslogik

$\langle ActivityModel \rangle ::= \langle Activity \rangle^* \langle ActivityConnection \rangle$

$\langle Activity \rangle ::= \langle ActivityStart \rangle | \langle ActivityAction \rangle | \langle ActivityCondition \rangle | \langle ActivityDisplay \rangle$

$\langle ActivityStart \rangle ::= \epsilon$

$\langle ActivityAction \rangle ::= \langle ActivityFlowCall \rangle | \langle ActivityPitaBuildInforRequest \rangle | \langle ActivityLoadExternalData \rangle$

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

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

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

$\langle ActivityPortValue \rangle ::= \langle FlowPortValue \rangle | \langle ActivityPortRefernce \rangle$

$\langle FlowPortValue \rangle ::= \langle string \rangle | \langle number \rangle | \langle bool \rangle | \langle date \rangle | \langle FlowPortValue \rangle^*$

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

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

$\langle ActivityCondition \rangle ::= \langle ActivityBinaryCondition \rangle | \langle ActivityValidityCondition \rangle$

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

$\langle ActivityValidityCondition \rangle ::= \langle ActivityPortValue \rangle^*$

$\langle ActivityBinaryCondition \rangle ::= '=' | '\neq' | '<' | '\leq' | '>' | '\geq'$

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

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

Grammatik TODO Aktivitätsmodell

Das Aktivitätsmodell `¡ActivityModel¿` besteht aus einer Reihe von Aktivitäten `¡Activity¿`. Aktivitäten können dabei entweder eine Startmarkierung, eine Aktivitätsaktion, einem Vergleich oder visualles Label sein. Die Startmarkierung muss pro Prüfungslogik genau einmal vorkommen. Ein vergleich kann dabei entweder eine Binärentscheidung oder eine Validierungsentscheidung sein. Die Validierungsentscheidung nimmt als Eingabe einen Wert und überprüft ob diese Werte vorhanden sind. Die Binärentscheidung nimmt als Parameter zweite Werte, einen Vergleichoperator und eine referenz zu einer Funtkion. Dabei werden beide Werte als Eingabe für die referenzierte Funktion verwendet. Als Vergleichoperatoren stehen `=` und `≠` sowie Relationale Operatoren zur verfügung. Eine Aktivitätsaktion `¡ActivityAction¿` kann dabei einer der folgenden Aktionen ausführen: Bevor das Ergbeniss aus der vorherigen Aktionsaktivität verwendet wird, kann eine Transformation auf dieses Ergebniss angewendet werden.

- Senden von Hauptuntersuchungs-Adatper-Anfragen (A1)
- Lesen einer JSON Datei (A2)
- Ausführung einer Datenverarbeitung (A3)

A1 nimmt als Parameter den Namen der auszuführenden Anfrage, eine Beschreibung für den debugger, eine Liste von anzusprechenden System im Fahrzeug und die maximale Zeitdrauer einer Anfrage. A2 nimmt als Eingabe den Typ der zu ladenenden Datei und die dazugehörige URI. A3

2.2 Datenverarbeitungen

Eingabe `¡FlowInputPort¿` und Ausgabe `¡FlowOutputPort¿` Funktionen höherer Ordnung `¡FlowLambda¿`. Eine Funktion höher Ordnung besteht aus zusätzlicehn Eingabe- und Ausgabeports Eingabe- und Ausgabeports nehmen als Parameter einen Namen des Ports, den Typ und ob Fehlererlaub ist.

Ein Funktions Template `¡FlowTemplate¿` besteht aus einer Funktion `¡Flow¿` und belieg vielen Parametern `¡TemplateParameter¿`. Die Parameter generieren Port- und Lambda-Defintion Funktionen welche vom Autorensystem `¡LibraryFlow¿` und selbst definierte Funktionen `¡FlowModel¿`.

Ein Flow-Modell ist ein DAG bei dem mehrere Funktionen mitienander verbunden werden. Einzelne Funktionen werden Nodes `¡FlowNode¿` genannt. Das Flow-Modell wird durch eine `¡FlowInstance¿`, Reihe von Funktionen und Verbindungen definiert. Die Funktion kann dabei eine Eingabe `¡FlowNodeInput¿`, eine Ausgabe `¡FlowNodeOutput¿`, einer Lambda Referenz `¡FlowNodeLambda¿` oder eine Funtkions Referenz `¡FlowNodeFlowCall¿` Konstante Werte `¡FlowPortValue¿`.

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

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

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

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

Grammatik TODO Flow-Instanz

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

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

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

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

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

Grammatik TODO Flow-Template

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

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

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

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

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

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

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

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

$\langle \textit{TemplateParameterValueList} \rangle ::= \langle \textit{TemplateParameterValue} \rangle^*$

Grammatik TODO Flow-Modell

2.3 Typsystem

unterstützt die gleichen Primitiv-Typen wie JSON-Format String, Number und Bool zusätzlich Date und PtiaResponse. Außerdem werden auch generische Typen unterstützt, weil nicht immer von vorneherein der Typ bekannt ist. Date ist eine Datumsangabe PtiaResponse ist eine Antwort einer Hauptuntersuchungs-Anfrage Diese Typen lassen sich an optionalen, Listen oder Objekt-Typen kapseln

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

Grammatik TODO Typ-Definition mit generischen und Referenz-Typen

3 Implementierung

Bei der Implementierung muss nicht nur auf das Design des Schleifenkonstrukts geachtet werden, sondern auch auf neue Sachen, welche durch die Implementierung entstanden sind. Bei den Schleifendurchläufen wird nicht auf die Ergebnisse des letzten Durchlaufs zugegriffen werden, sondern der Schleifenkörper soll die Entscheidungen auf Grundlage des aktuellen Sensorwertes treffen. Das Auslesen des aktuellen Sensorwertes ist bereits möglich. Aktuell unterstützt die zugrundeliegende Implementierung noch keine Variablen. Um das zu ändern muss die Grammatik bearbeitet werden-

3.1 1. Lösungsansatz

Der Benutzer gibt von vorneherein eine Zahl a an, welche die maximale Anzahl von Schleifendurchläufen beschränkt. Für die Zahl muss dafür folgendes gelten: TODO. Die Idee des Ansatzes ist es, den Schleifenkörper nicht iterativ oder rekursiv ausführen, sondern a -mal auszurollen. Dafür wird der Schleifenkörper und die nachfolgenden Anweisungen a -mal kopiert. Die Schleife wird dadurch nicht dynamisch ausgeführt, sondern statisch in den Code implementiert. Dadurch entstehen $a+1$ Graphen. Jeder dieser Graphen repräsentiert eine ursprüngliche Iteration. Dabei werden die einzelnen Graphen mit ihren direkten Nachbarn verbunden. Da die aktuell zugrunde liegende Implementierung deterministisch ist und aktuell nur auf die gleiche Eingabe zugegriffen werden kann, muss ein Mechanismus implementiert werden, welcher den aktuellen Sensorwert ausliest und diesen an die nachfolgenden Anweisungen weitergibt. Dieser Vorgang muss für jede neu eingefügte Verbindung wiederholt werden. Dieser Ansatz wird auch von Ye et al. im Konferenz-Paper "Loop Unrolling Based on SLP and Register Pressure Awareness" beschrieben.

Auf die Vor- und Nachteile der Implementierung wird im Kapitel 4 eingegangen.

3.2 2. Lösungsansatz

Wie auch schon beim 1. Lösungsansatz gibt der Benutzer von vorneherein eine Zahl a an, welche die Anzahl an Schleifendurchläufen beschränkt. Für die Zahl a gelten die gleichen Bedingungen wie im 1. Lösungsansatz beschrieben. Zusätzlich wird noch eine Zahl TODO benötigt, welche auch der Benutzer angeben muss. TODO soll dabei die Funktionen eines Grenzwertes übernehmen. Bei diesem Ansatz werden mehrere Mittelwerte gebildet und geschaut, wie sich der neu ausgelesene Sensorwert im Verhältnis zu den Mittelwerten verhält. Es wird die Differenz zwischen Mittelwert und aktuellen Sensorwert gebildet. Anschließend wird geschaut, ob die Differenz größer als TODO ist. Die Mittelwerte bilden wir einmal über alle bisherigen Sensorwerte und einmal über die letzten b Sensorwerte. Dadurch haben wir die Mittelwerte für einen kurzen und längeren Zeitraum. Sollte das der Fall sein, wissen wir, dass die Sensorwerte sich noch nicht stabilisiert haben und wir können den Vorgang wiederholen. Da wir

nicht bereits nachdem ersten stabilisierten Wert aufhören wollen, sondern erst wenn der Wert über einen längeren Zeitraum stabil ist, führen wir folgenden n-Chance-Mechanismus ein:

- Sollte der Grenzwert unterschritten werden, wird der Counter um 1 erhöht.
- Sollte der Grenzwert überschritten werden, wird der Counter wieder auf 0 gesetzt.
- Sollte der Counter irgendwann n erreichen, wissen wir das sich die Werte stabilisiert haben und wir davon ausgehen können dass das zu erwartende Ergebniss nicht mehr rauskommt.

4 Evaluation

4.1

4.1.1 1. Lösungsansatz

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