folgende Begriffe sollen definiert werden:
Visual Programming Language
Grammatik
Domain Specific Language
Schleifen
Flowchart
(Fixpunktberechnung)

Contents

1 Einleitung

test

2 Grammatikanalyse

2.1 Prüfungslogik

⟨*ActivityModel*⟩ ::= ⟨*Activity*⟩* ⟨*ActivityConnection*⟩

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

⟨*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

2.2 Datenverarbeitungs

⟨*FlowInstance*⟩ ::= ⟨*FlowOutputPort lambdaArguments*⟩* ⟨*FlowInputPort lambdaArguments*⟩*

⟨*FlowLambda*⟩ ::= ⟨*FlowOutputPort lambdaArguments*⟩* ⟨*FlowInputPort lambdaArguments*⟩*

⟨*FlowInputPort*⟩ ::= ⟨*string name*⟩ ⟨*Type*⟩ ⟨*bool acceptsError*⟩

⟨*FlowOutputPort*⟩ ::= ⟨*string name*⟩ ⟨*Type*⟩ ⟨*bool producesError*⟩

Grammatik TODO Flow-Instanz

⟨*FlowTemplate*⟩ ::= ⟨*Flow*⟩ ⟨*TemplateParameter*⟩*

⟨*Flow*⟩ ::= ⟨*LibraryFlow*⟩ | ⟨*FlowModel*⟩

⟨*LibraryFlow*⟩ ::= ϵ

⟨*TemplateParameter*⟩ ::= 'String' | 'Number' | 'Bool' | ⟨*TemplateParameterList*⟩

⟨*TemplateParameterList*⟩ ::= ⟨*TemplateParameter*⟩

Grammatik TODO Flow-Template

⟨*FlowModel*⟩ ::= ⟨*FlowInstance*⟩ ⟨*FlowNode*⟩* ⟨*FlowConnection*⟩*

⟨*FlowNode*⟩ ::= ⟨*FlowNodeOutput*⟩ | ⟨*FlowNodeInput*⟩ | ⟨*FlowNodeLambda*⟩ | ⟨*FlowNodeFlowCall*⟩

⟨*FlowNodeOutput*⟩ ::= ref(FlowOutputPort) ⟨*FlowPortValue*⟩

⟨*FlowNodeLambda*⟩ ::= ref(FlowLambda) ⟨*FlowPortValue*⟩*

⟨*FlowNodeFlowCall*⟩ ::= ref(FlowTemplate) ⟨*FlowPortValue*⟩* ⟨*TemplateParameterValue*⟩*

⟨*FlowConnection*⟩ ::= ref(FlowOutputPort source) ref (FlowOutputPort target)

⟨*FlowConnection*⟩ ::= ref(FlowOutputPort source) ref (FlowOutputPort target)

⟨*TemplateParameterValue*⟩ ::= ⟨*string*⟩ | ⟨*number*⟩ | ⟨*bool*⟩ | ⟨*TemplateParameterValueList*⟩

⟨*TemplateParameterValueList*⟩ ::= ⟨*TemplateParameterValue*⟩*

Grammatik TODO Flow-Modell

2.3 Typsystem

⟨*Type*⟩ ::= ⟨*TypePrimtive*⟩ | ⟨*TypeOptional*⟩ | ⟨*TypeList*⟩ | ⟨*TypeObject*⟩

⟨*TypePrimtive*⟩ ::= 'String' | 'Number' | 'Bool' | 'Data' | 'PtiaResponse'

⟨*TypeOptional*⟩ ::= ⟨*Type*⟩ '?'

⟨*TypeList*⟩ ::= ⟨*Type*⟩ '[]'

⟨*TypeObject*⟩ ::= '{' (⟨*string key*⟩ ':' ⟨*Type*⟩)* '}'

⟨*TypeGeneric*⟩ ::= '$' ⟨*string genericName*⟩

⟨*TypeReference*⟩ ::= ref(Type)

Grammatik TODO Typ-Defintion mit generischen und Referenz-Typen

3 Implementierung

3.1 1. Lösungsansatz

Der Benutzer gibt von vorneherein eine Zahl a an, welche die maximale Anzahl von Schleifendurchläufe beschränkt. Für die Zahl muss dafür folgendes gelten TODO. Die Idee des Ansatzes ist es, den Schleifenköper nicht Iterativ oder Rekursiv ausführen, sondern a-mal auszurollen. Dafür wird der Schleifenköper und die nachfolgenden Anweisungen a-mal kopiert. Die Schleife wird dadurch nicht dynmasisch ausgeführt, sondern statisch in den Code implementiert. Dadurch entstehen $a+1$ Graphen. Jeder dieser Graphen repräsentiert einen ursprüngliche Iteration. Dabei werden die einzelnen Graphen mit ihren direkten Nachbaran verbunden. Da die aktuell zugrunde liegende Implementierung determenistisch 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 wiederolt 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 beschrnänkt. 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 sich im Verhältnis zu den Mittelwerten verhält. Es wird die Differenz zwischen Mittelwert und aktuellen Sensorwert gebildert. Anschließend wird geschaut auf 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 das 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 einen n-Chance 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 +keine Endlosschleife +keine Zyklen +weniger Sprünge +möglicher Performance gewinn -größerer Codeumfang -höherer verbraucht an ressourcen zB 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