

folgende Begriffe sollen definiert werden:

Visual Programming Language

Grammatik

Domain Specific Language

Schleifen

Flowchart

(Fixpunktberechnung)

Contents

1	Einleitung	3
2	Terminologie	4
2.1	Schleifen	4
2.2	Datenfluss-basierte Systeme	4
2.3	Domain Specific Language	4
2.4	Visual Programming Language	5
3	Aufbau der domainspezifischen Sprache	6
3.1	Prüfungslogik	6
3.2	Datenverarbeitungs	8
3.3	Typsystem	10
4	Implementierung	11
4.1	1. Lösungsansatz	11
4.2	2. Lösungsansatz	12
5	Evaluation	14
5.1	14
5.1.1	1. Lösungsansatz	14
6	Literaturverzeichnis	15

1 Einleitung

2 Terminologie

2.1 Schleifen

Eine Schleife ist eine Kontrollstruktur, welche einen Code-Abschnitt mehrmals ausführt. [21] Sie stellt dabei oftmals die zeitintensivste Komponente eines Programms dar, weil die Ausführung der Schleife sehr viel Zeit in Anspruch nehmen kann. [6] Der Algorithmus der Kontrollstruktur kann bei iterativ oder rekursiv umgesetzt werden. Beide Ansätze haben dabei das gleiche Ziel, aber setzen die Schleife anders um. Bei der Iteration wird die Schleife mehrmals wiederholt. Bei der Rekursion ruft sich die zu wiederholende Funktion mehrmals selbst auf. Die Iteration verwendet dabei einen Akkumulativenansatz, welcher das zu lösende Problem schrittweise löst und diesen Vorgang solange wiederholt bis eine vordefinierte Abbruchbedingung erreicht wurde ist. Die Rekursion verwendet keinen Akkumulativenansatz, sondern reduziert das eigentlich zu lösende Problem auf mehrere (Teil-)Probleme. Für die Teilprobleme werden dann einzelne Lösungen gesucht, welche anschließend zusammengesetzt werden um das eigentliche Problem zu lösen. [2] Laut Chen L. spiegelt die Iteration das menschliche Denken wieder und ist daher besonders für lineare Probleme geeignet. Hingegen die Rekursion für lineare oder sequentielle Probleme geeignet sind, welche Zwischenergebnisse oder Teillösungen benötigen. [2] Die Iteration lässt sich dabei in zeitabhängige oder horizontale Iteration unterteilen. Bei der zeitabhängigen Iteration ist das Ergebnis des aktuellen Schleifendurchlaufes von dem Ergebnis des vorherigen Schleifendurchlaufes abhängig. Hingegen bei der horizontalen Iteration die Ergebnisse der Schleifendurchläufe unabhängig voneinander sind. [4]

2.2 Datenfluss-basierte Systeme

2.3 Domain Specific Language

Eine domänenspezifische Sprache (DSL) ist eine Programmiersprache, welche für einen bestimmten Anwendungsbereich entwickelt wurde und normalerweise auch auf diesen beschränkt ist. Das Ziel einer DSL ist es in dem klar abgegrenzten Anwendungsbereich die Probleme effizient zu lösen. [18] Dabei fallen DSL nicht in die Gruppe der General-Purpose Language (GPL) wie z.B. Java, C++ oder Python, sondern bilden das Gegenstück dazu. [15] Die Syntax ist dabei oftmals eingeschränkter und erlauben nur eine bestimmte Auswahl an Notationen und Befehlen. In manchen Fällen hat eine DSL aber auch eine GPL als Zweitsprache. [18] DSLs lassen sich in externe und interne unterteilen. Externe DSLs haben ihre eigene Syntax und TODO. Dadurch kann eine größere Flexibilität geschaffen werden, aber zeitlich ist der Aufwand für den Entwickler sehr hoch, weil alle relevanten Tools selbst implementieren muss. [7] Zur Laufzeit wird die externe DSL dann in eine GPL übersetzt. [15] Interne DSLs verwenden die Syntax einer GPL und kann über eine Programmierschnittstelle oder Bibliothek aufgerufen werden. [15] Allgemein sind DSLs kompakt, wiederver-

wendbar, effizient und domänenspezifisch. Aber auf der anderen Seite ist die Erstellung einer DSL kostspielig und haben einen hohen Lernaufwand für den Benutzer. Zudem haben sie nur einen eingeschränkten Anwendungsbereich und sind nur begrenzt verfügbar. [18]

2.4 Visual Programming Language

Das Ziel von VPLs ist es die Darstellung der Programmierlogik zu verbessern und den Ablauf des Programms zu verstehen. [14] Außerdem soll der Benutzer sich mehr auf die Implementierung des Algorithmus konzentrieren anstatt auf die Syntax, weil diese auf die IDE übertragen wird. [10] Bei VPLs stehen dem Programmierer nur ein bestimmter Satz an grafischen Elementen gegenüber statt dem ganzen Alphabet wie bei GPLs. Dadurch ist das Programm leichter zu verstehen und Fehler z.B. Semantik oder Syntax Fehler lassen sich bereits beim Erstellen des Programms vermeiden. [10] VPLs lassen sich dabei in Imperativ und Deklarativ unterteilen. In einer imperativen VPL wird vom Programm vorgegeben, in welcher Reihenfolge die Operationen ausgeführt werden. Bei einer deklativen VPL hingegen wird vom Programm nur die Abhängigkeit zwischen Daten vorgegeben und das System bestimmt die Reihenfolge selbst. [12]

3 Aufbau der domainspezifischen Sprache

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 (BNF) Notation. Der Aufbau einer BNF wird anhand der Grammatik 3 erklärt `symbol` sind nichtterminale `::=` bedeutet dass `symbol` 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 Softwareprüfung lässt sich visuell von zwei seiten betrachten.

$\langle symbol \rangle ::= _expression_$

Grammatik TODO Backus-Naur-Form

Grammatik lässt sich in 3 Ebenen unterteilen Prüfungslogik, Datenverarbeitung und Typsystem Prüfungslogik führt Entscheidung im Prüfungsablauf aus und bestimmt die Reihenfolge der Aktionen. außerdem datenerfassung Datenverarbeitung ist für die Datentransformation auswertung zuständig. Also Funktionen, welche keine Nebeneffekte besitzen, weil sie unabhängig von der restlichen Softwareprüfung stattfinden. Typsystem ermöglicht die statische analyse der ausführbarkeit Softwareprüfung lässt sich visuell von zwei seiten betrachten. Einmal als Datenflussgraphen, indem Teil-Funktionen als Blöcke dargestellt werden und Funktionsparameter/Ergebnisse als Ports. Einmal als Aktivitätsdiagramm, in dem nur Startzustand, Endzustände, Aktions- und Entscheidungsblöcke dargestellt

3.1 Prüfungslogik

Das Aktivitätsmodell $\langle ActivityModel \rangle$ besteht aus einer Reihe von Aktivitäten $\langle Activity \rangle$. Aktivitäten können dabei entweder eine Startmarkierung, eine Aktivitätsaktion, einem Vergleich oder visuelles 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 Funktion. Dabei werden beide Werte als Eingabe für die referenzierte Funktion verwendet. Als Vergleichoperatoren stehen $=$ und \neq sowie Relationale Operatoren zur Verfügung. Eine Aktivitätsaktion `ActivityAction` kann dabei einer der folgenden Aktionen ausführen: Bevor das Ergebnis aus der vorherigen Aktionsaktivität verwendet wird, kann eine Transformation auf dieses Ergebnis angewendet werden.

- Senden von Hauptuntersuchungs-Adapter-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 Zeitdauer einer Anfrage. A2 nimmt als Eingabe den Typ der zu ladenen Datei und die dazugehörige URI. A3

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

Grammatik TODO Aktivitätsmodell

3.2 Datenverarbeitungen

Eingabe $\langle \text{FlowInputPort} \rangle$ und Ausgabe $\langle \text{FlowOutputPort} \rangle$ Funktionen höherer Ordnung $\langle \text{FlowLambda} \rangle$ Eine Funktion höher Ordnung besteht aus zusätzliedn Eingabe- und Ausgabeports Eingabe- und Ausgabeports nehmen als Parameter einen Namen des Ports, den Typ und ob Fehlererlaub ist.

Ein Funktions Template $\langle \text{FlowTemplate} \rangle$ besteht aus einer Funktion Flow und belieg vielen Parametern $\langle \text{TemplateParameter} \rangle$ Die Parameter generieren Port- und Lambda-Defintion Funktionen welche vom Autorensystem $\langle \text{LibraryFlow} \rangle$ und selbst definierte Funktionen $\langle \text{FlowModel} \rangle$

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

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

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

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

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

Grammatik TODO Flow-Instanz

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

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

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

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

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

Grammatik TODO Flow-Template

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

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

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

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

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

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

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

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

$\langle TemplateParameterValueList \rangle ::= \langle TemplateParameterValue \rangle^*$

Grammatik TODO Flow-Modell

3.3 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-Defintion mit generischen und Referenz-Typen

4 Implementierung

Bei der Implementierung muss nicht nur auf des Design des Schleifenkonstrukt 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-

4.1 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 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 einen 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.

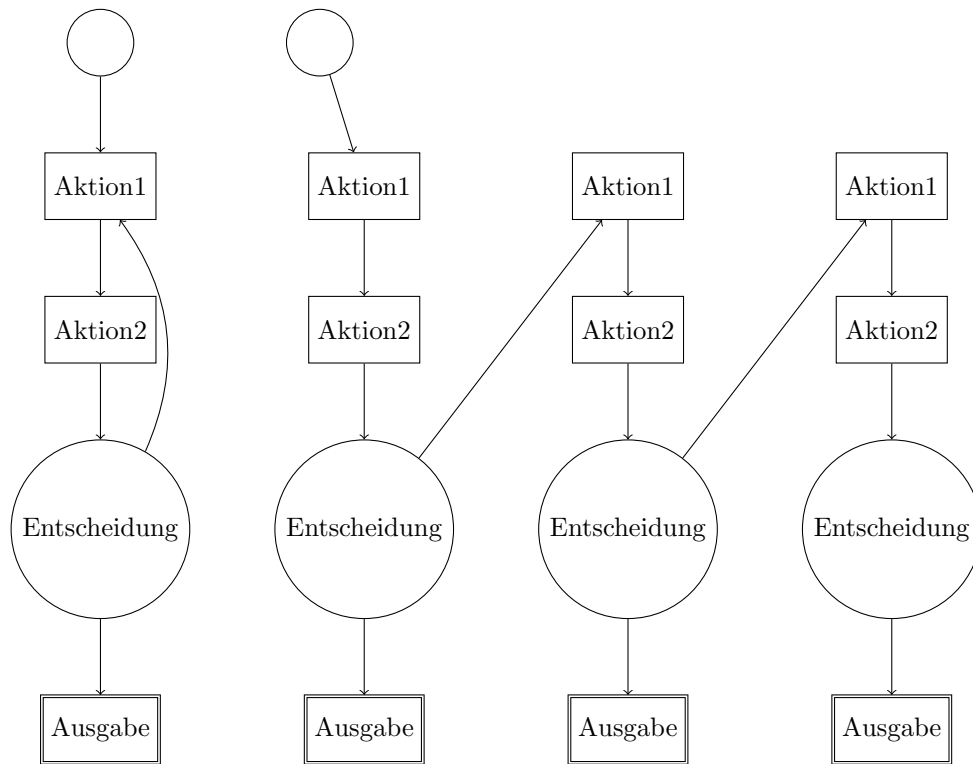


Abbildung TODO Algorithmus Schleifenentfaltung

Auf die Vor- und Nachteile der Implementierung wird im Kapitel 5 eingegangen.

4.2 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 sich im Verhältnis zu den Mittelwerten verhält. Es wird die Differenz zwischen Mittelwert und aktuellen Sensorwert gebildet. 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 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.

5 Evaluation

5.1

5.1.1 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 implemtniert 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

6 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*.