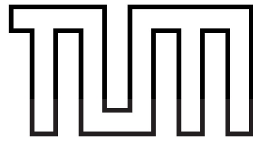


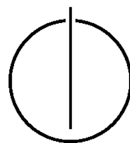
FAKULTÄT FÜR INFORMATIK

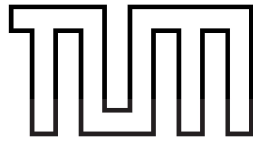
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Masterarbeit in Informatik

Messgrößenbasierte Typanalyse im Anlagenbau

Ann-Sophie Kracker





FAKULTÄT FÜR INFORMATIK

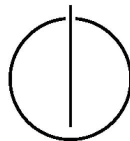
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Masterarbeit in Informatik

Messgrößenbasierte Typanalyse im Anlagenbau

Variable-type Analysis Based on Measurement Categories in Plant Engineering

Bearbeiter:	Ann-Sophie Kracker
Betreuer:	Dr. Benjamin Hummel, Dr. Elmar Jürgens, Dr. Alexander von Rhein
Aufgabensteller:	Prof. Dr. Dr. h.c. Manfred Broy
Abgabedatum:	15. Dezember 2020



Ich versichere, dass ich diese Masterarbeit selbstständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Sonthofen, den 9. Dezember 2020 Ann-Sophie Kracker

Danksagung

Bedanken will ich mich bei allen, die mich bei der Durchführung und dem Gelingen dieser Masterarbeit unterstützt haben.
Mein besonderer Dank gilt meinen Betreuern Dr. Alexander von Rhein, Dr. Benjamin Hummel und Dr. Elmar Jürgens, die auf Fragen jeglicher Art stets eine kompetente und zielführende Antwort wussten. Sie halfen mir, das Projekt wissenschaftlich zu bearbeiten, das Thema in verschiedene Richtungen zu hinterfragen und die Schwerpunkte richtig zu setzen.
Ebenso danke ich Herrn Prof. Dr. Dr. h.c. Manfred Broy, der mir die Umsetzung dieses spannenden Projektes ermöglicht hat.
Eine große Unterstützung habe auch durch das Maschinenbauunternehmen erfahren, das mir den Test auf ihrem eigenen System ermöglicht hat. Fragen im Laufe des Projektes wurden jederzeit kompetent beantwortet und die Ergebnisse der Analyse schnell und umfassend bewertet.

Inhaltsverzeichnis

Danksagung **v**

1 Einleitung **1**
1.1 Einführung in das Thema 1
1.2 Umsetzung der Typanalyse 2
1.3 Forschungsfragen 3
1.4 Strukturierung der Arbeit 3

2 Verwandte Arbeiten **4**

3 Hintergrund **7**
3.1 Statische Codeanalyse 7
3.2 Teamscale 8
3.3 Der IEC 61131-3 Standard 8
3.3.1 Speicherprogrammierbare Steuerungen (SPS) 9
3.3.2 Programmeinheiten 9
3.3.3 Structured Text (ST) 10
3.3.4 Projektstruktur 10

4 Messgrößenbasierte Typanalyse im Anlagenbau **13**
4.1 Systemvariablen 13
4.1.1 Aufbau von Systemvariablen 13
4.1.2 Parsen der Systemvariablendatei 15
4.2 Statische Analyse von Einheiten 16
4.2.1 Einheiten von Systemvariablen 16
4.2.2 Physikalischer Hintergrund und Vereinfachungsregeln 18
4.2.3 Umgang mit Konstanten 21
4.2.4 Regeln für Operationen 21
4.2.5 Genaue Einheiten 24
4.3 Parsen von *Structured Text* Statements 26
4.3.1 Der *CUP* Parser Generator 26
4.3.2 Grammatik für Anweisungen in *Structured Text* 26
4.3.3 Operatorpräzedenz 27

4.3.4 Erstellen des AST 28
4.3.5 Der Scanner 33
4.3.6 IF, CASE, FOR, REPEAT und WHILE 34
4.3.7 Generieren des Parsers 35
4.4 Die Typanalyse 35
4.5 Ausführen der Typanalyse in *Teamscale* 37

5 Evaluierung **46**
5.1 Forschungsfragen 46
5.2 Studienobjekt 46
5.3 Studienaufbau 47
5.4 Ergebnisse 47
5.5 Diskussion 49
5.6 Fazit des Maschinenbauunternehmens 51

6 Zusammenfassung und weiterführende Arbeiten **52**
6.1 Zusammenfassung 52
6.2 Weiterführende Arbeiten 53

Abbildungsverzeichnis **55**

Tabellenverzeichnis **56**

Listings **57**

Literatur **58**

1 Einleitung

1.1 Einführung in das Thema

Statische Analysen sind ein wichtiger Bestandteil der modernen Softwareentwicklung. Eine immer größere Rolle spielt der Begriff der kontinuierlichen Integration (Continuous Integration). Dazu zählt auch, dass der Entwickler eine möglichst schnelle Rückmeldung über die Qualität seines Programmcodes erhält. Die Idee ist es, Analysen und Tests möglichst oft und schnell auszuführen, um die Funktionalität kontinuierlich überprüfen zu können. Neben der korrekten Funktionalität einer Software gibt es auch andere Indikatoren für die Qualität, wie beispielsweise Effizienz, Bedienungsfreundlichkeit und Wartbarkeit. Aber genauso wichtig ist es, den Code selbst zu betrachten. Hält sich der Entwickler an bestimmte Richtlinien, können oftmals bekannte Fehlermuster vermieden werden. Um die Qualität von Software in Bezug auf solche Indikatoren zu prüfen, kann man unter anderem eine Statische Codeanalyse anwenden. In der vorliegenden Arbeit wird der neue Indikator der messgrößenbasierten Typanalyse eingeführt in die Analyse aufgenommen.

Im Anlagenbau besitzen Variablen oftmals eine physikalische Bedeutung. In diesem Fall können einer Variable im Programmcode eine physikalische Größe sowie eine Einheit zugeordnet werden. Auf diese Weise kann beurteilt werden, ob eine Operation zulässig ist. Findet beispielsweise eine Addition eines Druckwertes mit der Einheit *bar* mit einem Geschwindigkeitswert in $\frac{km}{h}$ statt, ist diese Operation vermutlich nicht gewollt. Sie ist physikalisch nicht korrekt, da nur Werte mit identischen Einheiten sinnvoll addiert werden können. Die Typanalyse deckt solche Fehler auf und weist den Entwickler darauf hin.

Die Situation im Bereich des Anlagenbaus unterscheidet sich in einigen Punkten von der Softwareentwicklung im Informatikbereich. Die Maschinen werden von dedizierten Rechnern gesteuert oder geregelt. Diese echtzeitfähigen Controller besitzen meist Hardware-Ein- und Ausgänge und die Software wird in speziellen Sprachen entwickelt. Der Standard für sog. Speicherprogrammierbare Steuerungen (SPS) wird von der *International Electrotechnical Commission* festgelegt. Im *IEC 61131-3* Standard sind fünf Programmiersprachen definiert. Die Typanalyse wird in dieser Arbeit auf die textuelle

Programmiersprache *Structured Text* beschränkt.

Aufgrund der hardwarenahen Programmierung können Fehler im Anlagenbau leicht zu kostenintensiven Hardwareschäden oder im schlimmeren Fall zu Personenschäden führen. Gerade deshalb sind in diesem Feld kontinuierliches Testen sowie eine Codebasis, die den Qualitätsrichtlinien entspricht, von großer Bedeutung. Um frühzeitig Fehler zu erkennen, die sich direkt auf physikalische Komponenten beziehen, oder diese sogar ganz zu vermeiden, wurde im Rahmen dieser Arbeit eine messgrößenbasierte Typanalyse für *IEC 61131-3 ST (Structured Text)* Code entwickelt.

1.2 Umsetzung der Typanalyse

Die Typanalyse wird in das Codeanalysetool *Teamscale* integriert. Dabei handelt es sich um eine Software, die sowohl statische als auch dynamische Analysen durchführt und u.a. Testabdeckungen und deren Änderung bestimmen kann. *Teamscale* arbeitet inkrementell, d.h. die Versionsverwaltung eines Projektes wird für die Analysen berücksichtigt. *Teamscale* besitzt bereits einige statische Analysen für *Structured Text*. Diese werden um die beschriebene Typanalyse erweitert, sodass die Ergebnisse gemeinsam auf der Benutzeroberfläche von *Teamscale* angezeigt werden können.

Um die Analyse umzusetzen, muss die physikalische Bedeutung von Variablen extrahiert werden. Dafür stehen Dateien mit Definitionen von sog. Systemvariablen zur Verfügung. Hier kann einzelnen Variablen eine physikalische Größe oder eine Einheit zugewiesen werden. Anschließend wird ein Parser für einzelne *Structured Text* Anweisungen entwickelt, um die Operationen und Variablen im Programmcode zu identifizieren. Während der Analyse werden Operationen auf die Konsistenz von Einheiten überprüft. Beim Entdecken eines Fehlers wird in *Teamscale* ein Finding generiert, d.h. in der grafischen Benutzeroberfläche wird die Zeile der entsprechenden Datei markiert und der Entwickler wird mit einer passenden Beschreibung auf den Fehler hingewiesen. Bei korrekten Zuweisungen werden Einheiten weitergegeben und fließen in die folgende Analyse mit ein.

Schlussendlich wird die Analyse auf der Codebasis eines großen Maschinenbauunternehmens für Spritzgussmaschinen ausgeführt.

Aus der beschriebenen Problemstellung und dem Lösungsansatz ergeben sich die folgenden Forschungsfragen.

1.3 Forschungsfragen

1. Ist eine statische messgrößenbasierte Typanalyse in *Structured Text* auf Basis von Systemvariablen möglich?
2. Findet die Analyse potenzielle Fehler auf echtem Projektcode?
3. Kann eine solche Typanalyse effizient umgesetzt werden?
4. Sind die Ergebnisse der Analyse für Entwickler in der Praxis nützlich?

1.4 Strukturierung der Arbeit

Zu Beginn wird die Arbeit in einen größeren Kontext eingeordnet, indem in Kapitel 2 verwandte Arbeiten vorgestellt werden. in Kapitel 3 wird ein kurzer Überblick über statische Codeanalyse, *Teamscale* sowie den *IEC 61131-3* Standard gegeben. In Kapitel 4 wird die Umsetzung der messgrößenbasierten statischen Analyse erläutert. Dazu zählen das Konzept der Systemvariablen, die Typanalyse sowie das Generieren des Parser und der Findings in *Teamscale*. Es folgt eine Evaluation der Ergebnisse in Kapitel 5. Zum Schluss wird in Kapitel 6 ein kurzer Überblick darüber gegeben, welche Verbesserungen und Erweiterungen in Zukunft umgesetzt werden können, die über den Umfang der Arbeit hinausgehen.

2 Verwandte Arbeiten

Die Annahme, dass semantische Informationen in Form von physikalischen Größen oder Einheiten eine wichtige Rolle im Entwicklungsprozess spielen können, existiert schon seit einiger Zeit. Bereits im Jahr 1995 veröffentlichten Ian J. Hayes und Brendan P. Mahony ihre Forschungsarbeit zum Thema *Using Units of Measurement in Formal Specifications* [HM95]. Sie betonen, dass die Berücksichtigung von Einheiten bei der Programmierung besonders im Bereich der Physik das Verständnis und die Darstellung des Systems deutlich verbessern. Die beiden Wissenschaftler wollen physikalische Einheiten als ein Typsystem einführen, vergleichbar mit der statischen Typisierung von Variablen. Auf diese Weise könnten bereits vor oder während des Kompiliervorgangs Typisierungsfehler in Bezug auf physikalische Einheiten erkannt werden, analog zu den statischen Variablentypen der Programmiersprache. In der Arbeit wird mithilfe einer Erweiterung der Z-Notation[1] formal bewiesen, dass eine statische Dimensionsanalyse dieser Art machbar ist. Umgesetzt und getestet wurde der Ansatz jedoch nicht.
Die Ergebnisse zeigen, dass das Typisieren von Variablen mit physikalischen Einheiten realisierbar ist und zur frühzeitigen Fehlererkennung beitragen kann. In der vorliegenden Arbeit wird ein ähnlicher Ansatz verwendet, um die Typanalyse umzusetzen. Auch hier werden physikalische Einheiten in einer statischen Analyse berücksichtigt, um so Fehler frühzeitig zu erkennen. Im Gegensatz zu [HM95] besitzen jedoch nicht alle Variablen eine direkte physikalische Bedeutung, womit das Typisierungsschema nicht vollständig ist.

Ein allgemeinerer Ansatz der semantischen Analyse wird in dem Paper *Automatic Dimension Inference and Checking for Object-Oriented Programs* aus dem Jahr 2009 verfolgt [HL09]. Dabei geht es nicht mehr nur um die Anwendung im Umfeld der Physik. Hierbei wird einer Variable keine konkrete Bedeutung zugewiesen, stattdessen werden Interferenzen zwischen Variablen bestimmt und somit ein semantischer Zusammenhang hergestellt. Die Autoren heben hervor, dass für Klassenobjekte in der objektorientierten Programmierung meist ausgereifte Mechanismen zum Prüfen der Typeigenschaften bereitstehen. Für primitive Datentypen und Strings dagegen ist dies nicht der Fall, obwohl Variablen des gleichen Typs verschiedene Bedeutungen haben können. Alleine auf der Grundlage der Verwendung von Variablen im Programmcode

[1] Notation zur formalen Spezifikation von Software-Systemen

werden automatisiert Zusammenhänge zwischen diesen Variablen hergestellt. Um das Konzept umzusetzen, wurde das Tool *UniFi* entwickelt und erfolgreich an einem Java Projekt mit ca. 19.000 Zeilen getestet.
Für diese Masterarbeit liegen die semantischen Informationen bereits in Form von Systemvariablen und ihren zugehörigen Attributen vor. Durch den physikalischen Kontext sind Zusammenhänge zwischen Einheiten bereits festgelegt und müssen nicht ermittelt werden. Jedoch werden auch in dieser Arbeit Regeln für einzelne Operationen festgelegt, aus denen Inkonsistenzen und falsche Zusammenhänge ermittelt werden können.

Im Folgenden kam die Idee auf, die semantischen Informationen in Form von Annotationen bereitzustellen, um Ansatzpunkte für eine Analyse zu schaffen. Ben Lickly verfolgt in seiner Doktorarbeit mit dem Titel *Static Model Analysis with Lattice-based Ontologies* [Lic12] aus dem Jahr 2012 diesen Ansatz. Er entwickelt eine Methode, um mithilfe von wenigen manuellen Annotationen im Code Beziehungen innerhalb und zwischen Programmelementen herzustellen und zu verstehen. Nicht explizit definierte Konzepte werden dabei automatisch von bestehenden Informationen abgeleitet. Das Ziel der Arbeit ist es, ein Framework zu entwickeln, mit dessen Hilfe Annotationen mit semantischen Informationen bei der modellbasierten Entwicklung angegeben werden können. Das Konzept soll dabei helfen, parametrisierte Werte und strukturierte Datentypen besser verständlich darzustellen. Auf diese Weise sollen Fehler früher im Entwicklungsprozess ausfindig gemacht und v.a. die Fehlerursache schneller erkannt werden.
Das gleiche Ziel verfolgt die vorliegende Masterarbeit. Die hier entwickelten Typanalyse wird dazu verwendet, um Fehler und deren Ursache schneller und einfacher zu identifizieren. Dazu sind auch hier einige Variablen mit semantischen Informationen versehen, allerdings nicht im Umfeld der modellbasierten Entwicklung. Außerdem werden die Informationen nicht per Annotationen vergeben, sondern anhand des Konzeptes der Systemvariablen in dedizierten Dateien.

Tools zur statischen Analyse stehen für viele Programmiersprachen wie Java, C/C++, C# usw. in großer Anzahl zur Verfügung. Im Bereich der SPS-Programmierung existieren solche Analysetools jedoch nur sporadisch. In ihrem Forschungsbericht mit dem Titel *Opportunities and Challenges of Static Code Analysis of IEC 61131-3 Programs* [Prä+12] heben die Autoren diese Tatsache hervor. Im Rahmen ihrer Arbeit wird ein statisches Analysewerkzeug entwickelt, das eine regelbasierte Analyse für einen Dialekt von zwei der fünf *IEC 61131-3* Sprachen durchführt. Dafür wird im ersten Schritt ein Parser erstellt, der aus dem *IEC 61131-3* Code in einen Syntaxbaum erstellt. Mit dessen Hilfe wird anschließend nach bestimmten Sprachkonstrukten gefiltert, um diese genauer zu

untersuchen. Berücksichtigte Regeln sind bspw. Namenskonventionen, inkompatible Konfigurationseinstellungen, Programmkomplexität, aber auch das Erkennen von Race Conditions verschiedener Tasks und mögliche Performance-Probleme. Für das Tool werden insgesamt sieben Kriterien festgelegt und erläutert, das Tool ist aber generell erweiterbar.
Ein sehr umfangreiches Werkzeug zur statischen Codeanalyse stellt CODESYS[2] für alle Sprachen des *IEC 61131-3* Standards zur Verfügung. Dabei werden mehr als 100 Regeln berücksichtigt, die teilweise parametrisiert werden können. Kriterien, die hier überprüft werden, sind bspw. Schreibzugriffe auf Eingangsvariablen, nutzlose Deklarationen und nichterreichbarer Code. Die Durchführung der Analyse ist jedoch auf die CODESYS-Umgebung beschränkt.
Da das Konzept der Systemvariablen zum Typisieren von Variablen nicht Teil des *IEC 61131-3* Standards ist, sondern nur auf seinen Sprachen basiert und diese erweitert, sind Analysen in diesem Zusammenhang kein Teil der bestehenden Analysetools. Die Existenz eines Typanalysetools für die Sprachen des *IEC 61131-3* Standard ist zu diesem Zeitpunkt nicht bekannt.

[2]https://de.codesys.com/

3 Hintergrund

3.1 Statische Codeanalyse

Die Codeanalyse ist eine Methode, um Qualitätsmerkmale von Software zu überprüfen. Die statische Codeanalyse steht der dynamischen gegenüber. Während bei der dynamischen Analyse der Code ausgeführt wird, um die vorgegebenen Kriterien zu prüfen, wird bei der statischen Analyse lediglich der Text selbst inspiziert. Generell kann die statische Codeanalyse manuell oder automatisch durchgeführt werden. [Hof13] Soll eine automatische Analyse erfolgen, muss im Normalfall ein Parser bzw. mindestens ein Lexer für die zu untersuchende Programmiersprache zur Verfügung stehen. Nur so können bestimmte Sprachkonstrukte identifiziert werden, auf die hin der Code untersucht wird.

Bei der statischen Codeanalyse wird der Code anhand bestimmter Metriken untersucht und bewertet. Beispiele solcher allgemeinen Metriken sind die Methodenlänge, die Verschachtelungstiefe und die Vollständigkeit von Kommentaren. Es existieren auch sprachspezifische Analysen, wie bspw. eine Überprüfung in Python, ob private Methoden importiert werden.

Natürlich weist die statische Codeanalyse Grenzen auf. Für die Analyse von Metriken wie Performance, Funktionalität, Benutzbarkeit und Zuverlässigkeit muss auf den dynamischen Ansatz bzw. Softwaretests zurückgegriffen werden. [Hof13]. Die in dieser Arbeit durchgeführte Typanalyse wird als statische Codeanalyse realisiert. Syntaktische Fehlerquellen können durch statische Analysen verhältnismäßig einfach identifiziert werden [Hof13]. Schwieriger gestaltet sich im Allgemeinen das Erkennen von semantischen Inkonsistenzen, was in der Typanalyse jedoch notwendig ist. Hier müssen erst semantische Informationen aus dem statischen Kontext extrahiert werden.

3.2 Teamscale

Die in dieser Arbeit entwickelte statische Typanalyse wird in *Teamscale* integriert. *Teamscale* ist ein Tool der Firma *CQSE*[1], mit dessen Hilfe bestimmte Qualitätsanforderungen in Softwareprojekten fortlaufend überprüft werden können. Projektbegleitend werden ausgewählte Qualitätsindikatoren analysiert und den Entwicklern eine anschauliche Übersicht über die durchgeführten Analysen gegeben.
Teamscale bietet statische Analysen für ein breites Spektrum an Programmiersprachen. Einige Analysen wie **Clone Detection** sind sprachübergreifend, viele sind aber auch sprachspezifisch. Neben der Weboberfläche existieren Integrationen für IDEs, sodass der Entwickler direkt auf einen Verstoß gegen die festgelegten Richtlinien aufmerksam gemacht wird. Neben den statischen bietet *Teamscale* auch dynamische Analysen. Ein wichtiger Anwendungsfall ist z.B. die Test-Gap Analyse. Hier wird bei einem Commit geprüft, ob alle Änderungen getestet wurden. Außerdem gibt es Hinweise, welche Tests noch ausgeführt werden müssen, um eine vollständige Testabdeckung zu erreichen.
Teamscale stellt bereits einige statischen Analysen für *Structured Text* bereit, wie bspw. Namenskonventionen für Variablen, Verschachtelungstiefe, Methodenlänge und Vollständigkeit von Kommentaren. Die in dieser Arbeit entwickelte messgrößenbasierte Typanalyse wird neben diesen Checks eingeordnet.

3.3 Der IEC 61131-3 Standard

Die *International Electrotechnical Commission* (IEC) ist ein Gremium, das Normen für die Elektrotechnik erstellt. Eine dieser Normen ist *IEC 61131-3*. Sie befasst sich mit der Programmierung von Speicherprogrammierbaren Steuerungen (SPS, engl. Programmable Logic Controller (PLC)). In diesem Standard sind fünf Sprachen beschrieben:

- Instruction List (IL)
- Structured Text (ST)
- Sequential Function Chart (SFC)
- Function Block Diagram (FBD)
- Ladder Diagram (LD)

Bei der *Instruction List* handelt es sich um eine hardwarenahe textuelle Programmiersprache ähnlich zu Assembler. *Structured Text* hingegen zählt zu den höheren

[1] https://www.cqse.eu

Programmiersprachen. Er wird auch in textueller Form entwickelt. *Sequential Function Chart*, *Function Block Diagram* und *Ladder Diagram* sind grafische Programmiersprachen. Manche Sprachen sind für bestimmte Anwendungsfälle geeigneter als andere. Sie können innerhalb eines Projektes variieren bzw. kombiniert werden. In dieser Arbeit wird nur *Structured Text* berücksichtigt.

3.3.1 Speicherprogrammierbare Steuerungen (SPS)

Im Bereich des Maschinenanlagenbaus werden im Feld in den meisten Fällen SPSen eingesetzt. Sie zeichnen sich dadurch aus, dass sie echtzeitfähige Controller sind. Auf dem Markt gibt es viele verschiedene Steuerungen von unterschiedlichen Herstellern mit jeweils eigenen Entwicklungsumgebungen, die meist die Sprachen des *IEC 61131-3* Standard unterstützen. Manche SPSen basieren auf einem echtzeitfähigen Betriebssystem, andere besitzen eine eigene Laufzeitumgebung.

Die auf den SPSen ausgeführten Programme verhalten sich im Normalfall zyklisch. In diesen Punkt unterscheidet sich die Maschinenprogrammierung stark von der imperativen Programmierung. Statt einer festen Abarbeitungsreihenfolge von einzelnen Anweisungen existieren ein oder mehrere Tasks, für die jeweils unabhängige Zykluszeiten bestehen können. Diese Tasks werden dann beim Aufspielen des Programms gestartet und zu regelmäßigen Zeiten wiederholt. Innerhalb der Tasks können wiederum imperative Sprachen wie *Instruction List* oder *Structured Text* zum Einsatz kommen.

3.3.2 Programmeinheiten

Die *IEC 61131-3* Norm sieht verschiedene Bausteine zum Aufbau eines Programmes vor. Die grundlegenden Bausteine eines Projektes sind *POUs* (Program Organization Units). Es existieren drei Arten von *POUs*[JT09]:

- **Funktionen** (*FUN*) können mehrere Eingänge, jedoch nur einen Ausgang besitzen. *FUN*s erzeugen bei gleichen Eingaben immer das gleiche Ergebnis, da sie Werte nicht über einen Zyklus hinaus speichern.
- Für einen **Funktionsbaustein** (*FB*) können mehrere Ein- und Ausgänge definiert sein. Im Gegensatz zu *FUN*s, behalten *FB*s ihren Zustand, also die Werte der Variablen, über einen Zyklus hinweg. Damit können gleiche Eingabewerte zu verschiedenen Ausgangswerten führen. Mit diesem Baustein können auch Funktionalitäten realisiert werden, die abhängig von früheren Ereignissen und Ergebnissen unterschiedlich reagieren. *FB*s können instanziiert werden.
- Ein **Programm** (*PROG*) ist eine Art Hauptprogramm in der SPS-Programmierung. Es wird einem Task zugeordnet, der die Ausführung veranlasst. Während ein

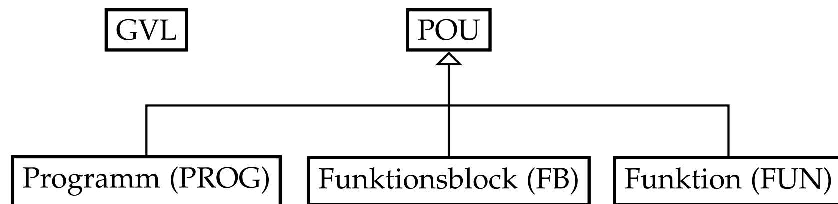


Abbildung 3.1: Struktur der Programm-Organisationseinheiten

Programm Funktionsblöcke und Funktionen instantiieren bzw. aufrufen kann, ist es umgekehrt nicht der Fall.

Jede *POU* wird in einen Deklarations- und einen Implementierungsteil untergliedert. Im Deklarationsteil werden die Variablen definiert. Dabei wird zwischen Eingangs-, Ausgangs- und lokalen Variablen unterschieden. Im Implementierungsteil wird die Funktionalität realisiert. Dies kann in jeder der fünf *IEC 61131-3* Sprachen geschehen. In einer sog. globalen Variablenliste (*GVL*) können projektweite Variablen definiert werden. Diese ist unabhängig von den *POU*s. Die Struktur der Grundbausteine ist in Abbildung 3.1 zu sehen.

3.3.3 Structured Text (ST)

Bei *Structured Text* handelt es sich um eine textuelle Programmiersprache, die zu den höheren Programmiersprachen gezählt wird. Sie ist der Sprache *PASCAL* am ähnlichsten. [JT09]
In Abbildung 3.2 ist ein kurzes Beispiel eines *FB* mit dem Namen **CalcPressureDiff** zu sehen, der die Differenz von zwei Druckwerten berechnet. Im Deklarationsteil werden zwei Eingangs- sowie zwei Ausgangsvariablen deklariert. In dem Codeauschnitt werden keine lokalen Variablen benötigt, deshalb bleibt der **VAR** Bereich leer. Im Implementierungsteil werden die zwei Eingangsparameter verglichen. Die Boolsche Indikatorvariable **bResult** gibt an, ob **iPressureLeft** größer oder gleich **iPressureRight** ist. Die folgende Subtraktion ist vom Ergebnis des Vergleichs abhängig.

Neben einer IF/ELSE Anweisung stehen in *Structured Text* auch FOR-, REPEAT- und WHILE-Schleifen sowie ein CASE-Konstrukt zur Verfügung.

3.3.4 Projektstruktur

Die Projektstruktur und die Benamung sind stark vom Hersteller abhängig. Im Allgemeinen können mehrere Steuerungen in ein Projekt eingebunden werden. Auf jeder

Abbildung 3.2: *Structured Text*

```
CalcPressureDiff
 1  // Deklarationsteil
 2  FUNCTION_BLOCK CalcPressureDiff
 3  VAR_INPUT
 4      iPressureLeft : INT;
 5      iPressureRight : INT;
 6  END_VAR
 7  VAR_OUTPUT
 8      bResult : BOOL;
 9      iDiff : INT;
10  END_VAR
11  VAR
12  END_VAR

 1  // Implementierungsteil
 2  bResult := iPressureLeft >= iPressureRight;
 3  IF bResult THEN
 4      iDiff := iPressureLeft - iPressureRight;
 5  ELSE
 6      iDiff := iPressureRight - iPressureLeft;
 7  END_IF
```

Steuerung können mehrere Applikationen laufen, die wiederum verschiedene Tasks parallel ausführen können.

Wie das Projekt im Dateisystem des Programmierrechners abgelegt wird, ist wiederum herstellerspezifisch. In der vorliegenden Arbeit wird mit Code gearbeitet, der in einer Entwicklungsumgebung geschrieben wurde, die das Projekt in verschiedene Projekteinheiten (PUs) unterteilt. Für jede dieser Einheiten gibt es einen Unterordner mit einer *.pu* Datei und beliebig vielen Systemvariablendateien mit der Endung *.sv*. Die hier definierten Systemvariablen sind innerhalb der Projekteinheit gültig. Darüber hinaus existiert eine *.sv* Datei auf der obersten Projektebene, die für das gesamte Projekt sichtbar ist. (s. Abbildung 3.3)
Der Aufbau einer PU-Datei ist wie folgt: Nach einer Importsektion folgen ein oder mehrere **ALGORITHM_BLOCK**s, die wiederum aus **ALGORITHM**s bestehen. Ein **ALGORITHM** beinhaltet den Deklarations- und den Implementierungsteil eine *POE*. Für jeden Funktionsblock, jede Funktion und jedes Programm gibt es dementsprechend einen **ALGORITHM**-Abschnitt in einer PU-Datei.

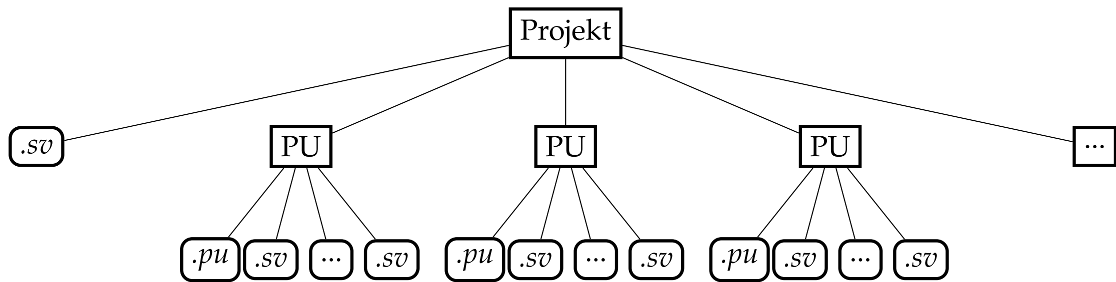


Abbildung 3.3: Projektstruktur

4 Messgrößenbasierte Typanalyse im Anlagenbau

Die Typanalyse wird auf der Basis von Systemvariablen realisiert. Die Idee dabei ist, dass ausgewählten Variablen eine physikalische Bedeutung zugeschrieben wird. Im ersten Schritt müssen die Systemvariablen gemeinsam mit der zugewiesenen physikalischen Größe oder Einheit aus den entsprechenden Dateien extrahiert werden. Im zweiten Schritt werden Vereinfachungsregeln für die physikalischen Größen festgelegt und implementiert, damit sie später vergleichbar sind. Anschließend wird für jeden Operator bestimmt, welche Kombinationen von Größen zulässig sind. Um die Operationen aus dem Code zu extrahieren, wird ein Parser für *Structured Text* Anweisungen entwickelt. Für jede dieser Anweisungen entsteht beim Parsen ein Syntaxbaum. Mit dessen Hilfe kann letztendlich die Typanalyse durchgeführt werden. Die identifizierten Regelverstöße können anschließend in *Teamscale* angezeigt und evaluiert werden.

4.1 Systemvariablen

Im Maschinenanlagenbau stehen Variablen in vielen Fällen für einen physikalischen Mess-, Steuer- oder Rechenwert. Um diese Informationen im Projektcode verwenden und darstellen zu können, hat die *KEBA AG*[1] das Konzept der Systemvariablen entwickelt. Die Typanalyse in dieser Arbeit wird auf der Basis von Systemvariablen und den dadurch gegebenen Zusatzinformationen realisiert.

4.1.1 Aufbau von Systemvariablen

Zum Definieren von Systemvariablen dienen dedizierte Dateien mit der Dateiendung *.sv* (Abk. für Systemvariable). Eine Systemvariablendatei ist in verschiedene Sektionen untergliedert:

- Import-Sektion
- Deklarationsteil für Systemvariablen

[1] https://www.keba.com/de/home

- Deklarationsteil für Systemevents
- Deklarationsteil für Systemalarme

Für die Arbeit relevant sind die Import-Sektion sowie der Deklarationsteil für die Systemvariablen. Systemevents und Systemalarme sind für die Typanalyse ohne Bedeutung und werden deshalb ignoriert. In der Import-Sektion werden Attributierungsoptionen importiert, also z.B. die physikalischen Größen, Formate und Variablengruppen. Im Deklarationsteil werden die Systemvariablen und ihre Attribute festgelegt. Einzelne Deklarationen werden durch ein Semikolon getrennt. Ein Beispiel für eine einfache *.sv* Datei ist in Listing 4.1 ausgeführt.

Listing 4.1: Deklaration von Systemvariablen

```
%IMPORT_OVER_LISTFILE_SOURCE
Force
, Length
, Pressure
END_IMPORT

%SYSTEMVAR_DECL

sv_xPosition : REAL := 0.1
%PLAUSIBILITY 0.0..9.0 %UNIT Length
;

sv_fb : TOtherFB := (a:=5,b:=@system.sv_pressure1)
%ELEMENT sv_fb.a
%UNIT Force
%ELEMENT sb_fb.b
%UNIT Pressure
;

%END
```

Attribute, die festgelegt sein müssen, sind der Variablenname sowie der Datentyp. Der Typ kann dabei einer der *IEC 61131-3* Basisdatentypen (s. Tabelle 4.1) oder ein instantiierbarer Funktionsblock sein. Optional können ein Initialwert und neben vielen anderen Eigenschaften auch eine Einheit (*%UNIT*) sowie ein Plausibilitätsintervall (*%PLAUSIBILITY*) zugewiesen werden.

Tabelle 4.1: Auswahl an Basisdatentypen in IEC 61131-3

Name	Beschreibung	Anzahl an Bits
BOOL	Boolean	1
SINT	Short Integer	8
INT	Integer	16
DINT	Double Integer	32
LINT	Long Integer	64
UINT	Unsigned Integer	16
REAL	Rationale Zahlen	32
STRING	Text	variabel
TIME	Zeit oder Dauer	variabel
...		

Als grundlegendes Attribut für die in der Arbeit durchgeführte Typanalyse dient die Einheit einer Variable, die durch das Attribut *%UNIT* festgelegt wird. Diese ist optional, muss also nicht für jede Variable vergeben werden.

4.1.2 Parsen der Systemvariablendatei

Die Systemvariablendateien werden direkt nach der Erstellung eines Projektes in Teamscale geparst. Dabei wird jede Systemvariable mit der entsprechenden Einheit bzw. physikalischen Größe in einer lokalen Liste abgelegt. Ist keine Einheit definiert, wird die Variable in dieser Liste als **UNDEFINED** markiert.

Der Aufbau einer Systemvariablendatei ist in Listing 4.1 zu sehen. Die Definitionen können in zwei Kategorien unterteilt werden. Die Variable **sv_xPosition** ist eine einfache Systemvariable. Sie führt einen der Basisdatentypen. In diesem Fall muss beim Parsen lediglich die passende Einheit zum Variablennamen extrahiert werden. Aufwendiger ist hingegen das Parsen bei der Verwendung von komplexen Datentypen. Im Beispielcode ist die Systemvariable **sv_fb** von Typ **TOtherFB** mit den Elementen **a** und **b**. In diesem Fall wird nicht der Systemvariable selbst eine Einheit zugewiesen, sondern ihren Subvariablen. Diese werden in den folgenden Zeilen nach den Schlüsselwort **%ELEMENT** definiert. Eine Systemvariable kann beliebig viele Subvariablen besitzen, die wiederum Subvariablen aufweisen können. Beim Parsen wird der Systemvariable **sv_fb** selbst keine Einheit zugewiesen. Sie besitzt aber eine Liste mit Subvariablen (**sv_fb.a** und **sv_fb.b**), denen wiederum die entsprechenden Einheiten (*Force* und *Pressure*) zugeordnet werden.
Als Alternative zu einem festen Wert kann einer Subvariablen eine Referenz auf eine

andere Systemvariable zugewiesen werden. Das geschieht, wie im Beispiel zu sehen, durch einen **@system.<Systemvariable>** oder einen **%FU.<Systemvariable>** Verweis. Die **@system** Referenz bezieht sich dabei auf die systemweite *.sv* Datei mit den projektweit gültigen Variablen. Wird eine Systemvariable mit **%FU** referenziert, handelt es sich um eine Variale innerhalb der Projekteinheit. Dabei ist die Reihenfolge nicht von Bedeutung. Es kann auf Systemvariablen verwiesen werden, die erst später in der gleichen Datei definiert werden. Außerdem ist in *Teamscale* nicht festgelegt, dass die *.sv* Datei der systemweiten Variablen vor den *.sv* Dateien der Projekteinheiten bearbeitet werden. Dies führt dazu, dass nicht alle Referenzierungen aufgelöst werden können. Die Lösung hierfür wäre ein zweimaliges Überprüfen jeder *.sv* Datei des gesamten Projektes. Diese Maßnahme wird aus Gründen der Performance und des Aufwandes im Rahmen der Masterarbeit vernachlässigt. Die Referenzierungen werden eher sporadisch eingesetzt und die meisten Referenzen können auch durch einmaliges Parsen einer Datei aufgelöst werden. Deshalb verändert die Einschränkung die vorgenommenen Analysen vermutlich nicht signifikant.

4.2 Statische Analyse von Einheiten

4.2.1 Einheiten von Systemvariablen

Die für das Projekt relevanten Einheiten sind in projektzugehörigen *.at* Dateien definiert. Listing 4.2 zeigt einen Ausschnitt einer solchen Datei. Hier werden jeder physikalischen Größe zwei Einheiten zugewiesen. Die erste Einheit ist Teil des metrischen Maßsystems, während die zweitgenannten Einheiten Vertreter des angloamerikanischen Systems sind. So kann die Größe **Stroke** (Längenmaß) in der Einheit *Millimeter* oder *Inch* angegeben sein. Bei einigen Größen sind die beiden Einheiten identisch. So wird die Leistung (*Power*) in beiden Fällen in *Kilowatt* angegeben. Bei anderen Einheiten wie **VoltageV** oder **VoltagemV** ist die Einheit bereits in der Namensgebung erkennbar. Das Einheitensystem lässt sich in der Projektoberfläche gesamtheitlich umstellen. Die Typanalyse wird in dieser Arbeit auf den allgemeinen physikalischen Größen durchgeführt, nicht auf den expliziten Einheiten. Der Grund dafür ist, dass auf der Basis einer statischen Analyse die Umrechnung zwischen Einheiten nicht vollständig nachvollzogen werden kann. Was für eine Analyse mit Einheiten geändert werden müsste und wo die Grenzen liegen, ist in Kapitel 4.2.5 beschrieben.

Listing 4.2: Definition der Einheiten im Projekt

```
%UNIT_DECL Stroke
%TRANS_FUNC system.hmi.MmInch // mm in
%END
```

```
%UNIT_DECL VoltagemV
// mV mV
%END

%UNIT_DECL VoltageV
// V V
%END

%UNIT_DECL Power
// kW kW
%END

%UNIT_DECL Force
%TRANS_FUNC system.hmi.KnTn // kN US ton
%END

%UNIT_DECL PressureAngle
%TRANS_FUNC system.hmi.BarPsi // bar/° psi/°
%END
```

In Teamscale werden sämtliche physikalische Größen in dem Enum **EIec61131SystemVariableUnit** zusammengefasst. Der Anfang der Klasse ist in Listing 4.3 zu sehen. Das Testprojekt in dieser Arbeit umfasst 128 Größen dieser Art.

Listing 4.3: Ausschnitt des Enums mit den physikalischen Größen

```
public enum EIec61131SystemVariableUnit {

        /* Acceleration in mm/s^2 or in/s^2 */
        Acceleration("Acceleration"),

        /* Acceleration_SIUnit in m/s^2 or m/s^2 */
        Acceleration_SIUnit("Acceleration_SIUnit"),

        /* AccelarationStroke in 1/s^2 or 1/s^2 */
        AccelarationStroke("AccelarationStroke"),

        /* Angle in ° or ° */
        Angle("Angle"),
```

```
/* AngleAccelaration in °/s² or °/s² */
AngleAccelaration("AngleAccelaration"),

...
```

4.2.2 Physikalischer Hintergrund und Vereinfachungsregeln

Die Einheiten müssen in Teamscale bekannt sein, bevor ein Projekt analysiert wird. Nur so können die Variablen und Operationen mit einem physikalischen Kontext versehen werden. Die Systemvariableneinheiten werden in zwei Kategorien aufgeteilt:

- **Basisgrößen**: Größen, die nicht weiter aufgeteilt werden können, bspw. die SI-Größen Länge, Masse und Zeit.
- **Abgeleitete Größen**: Größen, die sich aus den Basisgrößen zusammensetzen, bspw. die Geschwindigkeit, die sich von Länge und Zeit ableiten lässt.

Für jede abgeleitete Größe muss es eine Regel geben, mit welchen Größen sie korreliert bzw. aus welchen Größen sie sich zusammensetzt. Das ist notwendig, um Einheiten später sinnvoll miteinander vergleichen zu können.
Um Größen auf ihre Basisgrößen zurückzuführen, wurde im Rahmen dieser Arbeit die Klasse **UnitDeducer** implementiert. Hier ist definiert, welche Größen als Basisgrößen behandelt werden. Darunter zählen zum einen alle SI-Größen des internationalen Einheitensystems:

- Länge (Length)
- Masse (Mass)
- Zeit (Time)
- Elektrische Stromstärke (Electric Current)
- Absolute Temperatur (Temperature)
- Stoffmenge (Amount of substance)
- Lichtstärke (Luminous intensity)

Zum anderen werden einige projektspezifische Größen als Basisgrößen betrachtet, die nicht sinnvoll auf die SI-Einheiten zurückgeführt werden können. In diese Kategorie fallen:

- Counter
- Percent
- Points
- Shot
- u.v.m.

Einheiten bzw. Größen werden im Folgenden als Bruch betrachtet. Auf diese Art können alle Größen und ihre Folgerungen dargestellt werden. Wenn eine Systemvariable mit einer Einheit initialisiert wird, wird diese Einheit symbolisch auf den Zähler geschrieben und der Nenner bleibt leer. Wenn nun eine Größe rekursiv auf ihre Basisgrößen zurückgeführt wird, kann es sein, dass ein vollständiger Bruch entsteht. Im Folgenden wird dieser Ablauf anhand der Größe Druck (Pressure) demonstriert, wobei die Basisgrößen fett markiert sind.

Druck ist als Kraft pro Fläche definiert. Kraft wiederum kann als Masse mal Beschleunigung betrachtet werden. Masse ist eine Basisgröße. Die Beschleunigung setzt sich aus Geschwindigkeit und Zeit zusammen usw.

$$Pressure \rightarrow \frac{Force}{Area}$$

$$Force \rightarrow \mathbf{Mass} \cdot Acceleration$$

$$Acceleration \rightarrow \frac{Velocity}{\mathbf{Time}}$$

$$Velocity \rightarrow \frac{\mathbf{Length}}{\mathbf{Time}}$$

$$Area \rightarrow \mathbf{Length} \cdot \mathbf{Length}$$

Am Ende entsteht ein Bruch aus Einheiten. Für den Druck ergibt sich die Folgerung

$$Pressure \rightarrow \frac{Force}{Area} \rightarrow \frac{Mass \cdot Acceleration}{Area} \rightarrow \frac{Mass \cdot Velocity}{Area \cdot Time} \rightarrow \frac{Mass \cdot Length}{Area \cdot Time \cdot Time} \rightarrow \frac{Mass \cdot Length}{Length \cdot Length \cdot Time \cdot Time}$$

Anschließend wird der Bruch gekürzt.

$$\frac{Mass \cdot \cancel{Length}}{Length \cdot \cancel{Length} \cdot Time \cdot Time} \rightarrow \frac{Mass}{Length \cdot Time \cdot Time}$$

Diese Folgerung und das Kürzen sind notwendig, um Einheiten sinnvoll vergleichen zu können. Ohne Vereinfachung bzw. ohne die Rückführung auf Basiseinheiten wäre beispielsweise nicht erkennbar, dass die folgenden Addition gültig ist. Es muss erst geschlussfolgert werden, dass Druck eine Kraft pro Fläche ist:

```
sv_pressure + sv_force / sv_area;
```

In Listing 4.4 ist zu sehen, wie die Folgerungen umgesetzt sind. Die Vorgehensweise ist hierbei rekursiv. So wird bspw. die Beschleunigung im ersten Schritt in Geschwindigkeit und Zeit aufgeteilt. Erst im zweiten Schritt wird die Geschwindigkeit wiederum auf ihre Basiseinheiten zurückgeführt.
Sobald während der Typanalyse eine neue Größe erstellt oder eine neue Einheit zu einer bestehenden **UnitDivision** hinzugefügt wird, wird die neue Einheit rekursiv auf ihre Basiseinheiten zurückgeführt und anschließend mit der aktuell bestehenden Einheit zusammengeführt. Der Ablauf des Zusammenführens ist im Flussdiagramm 4.1 zu sehen. Der eigentliche rekursive Schritt erfolgt dadurch, dass in der Funktion **splitUnitUp** im **UnitDeducer** die Methoden **addToNumerator** und **addToDenominator** aufgerufen werden (s. Listing 4.4). Bei diesem Aufruf wird die neu hinzugefügte Einheit direkt weiter abgeleitet.

Listing 4.4: Ausschnitt aus der *UnitDeducer* Klasse

```
private static UnitDivision splitUnitUp(EIec61131SystemVariableUnit unit) {
        UnitDivision unitDivision = new UnitDivision();
        switch (unit) {
                /* Acceleration -> Velocity / Time */
                case Acceleration:
                        unitDivision.addToNumerator(Velocity);
                        unitDivision.addToDenominator(Time);
                        break;
                /* Area -> Length * Length */
                case Area:
                        unitDivision.addToNumerator(Length);
                        unitDivision.addToNumerator(Length);
                        break;
                /* Pressure -> Force / Area */
```

```
            case Pressure:
                  unitDivision.addToNumerator(Force);
                  unitDivision.addToDenominator(Area);
                  break;
            /* Force -> Mass * Acceleration */
            case Force:
                  unitDivision.addToNumerator(Mass);
                  unitDivision.addToNumerator(Acceleration);
                  break;
            /* Velocity -> Length / Time */
            case Velocity:
                  unitDivision.addToNumerator(Length);
                  unitDivision.addToDenominator(Time);
                  break;
            ...
      }
      return unitDivision;
}
```

4.2.3 Umgang mit Konstanten

Als Konstanten werden in diesem Kontext Zahlenwerte betrachtet, die direkt im Code vorkommen. Darunter fallen Integer-, Floating-Point-, Boolean-, Zeit- und String-Literale. Ihnen kann keine Einheit zugewiesen werden und sie werden dementsprechend als **UNDEFINED** behandelt. D.h. sie verändern bspw. bei einer Multiplikation oder Division die bestehende Einheit nicht.

4.2.4 Regeln für Operationen

Um eine Typanalyse durchführen zu können, muss erst für einzelne Operationen festgelegt sein, welche Kombinationen von Einheiten gültig sind. Bei unären Operatoren wie einer Dereferenzierung, einer Negation oder einer @-Referenz ist jeweils nur eine Variable mit einer Einheit beteiligt. Hier kann es keinen Verstoß gegen die Regeln geben. Anders ist die Lage bei binären Operatoren. Im Folgenden wird die Vorgehensweise für ausgewählte Operatoren genauer betrachtet. Allgemein werden einheitenlose Variablen als eine Art neutrales Element angesehen. D.h. sie verändern bei Multiplikationen und Divisionen die bestehende Einheit nicht und dürfen mit Variablen jeder Einheit verglichen, addiert oder ihr zugewiesen werden.

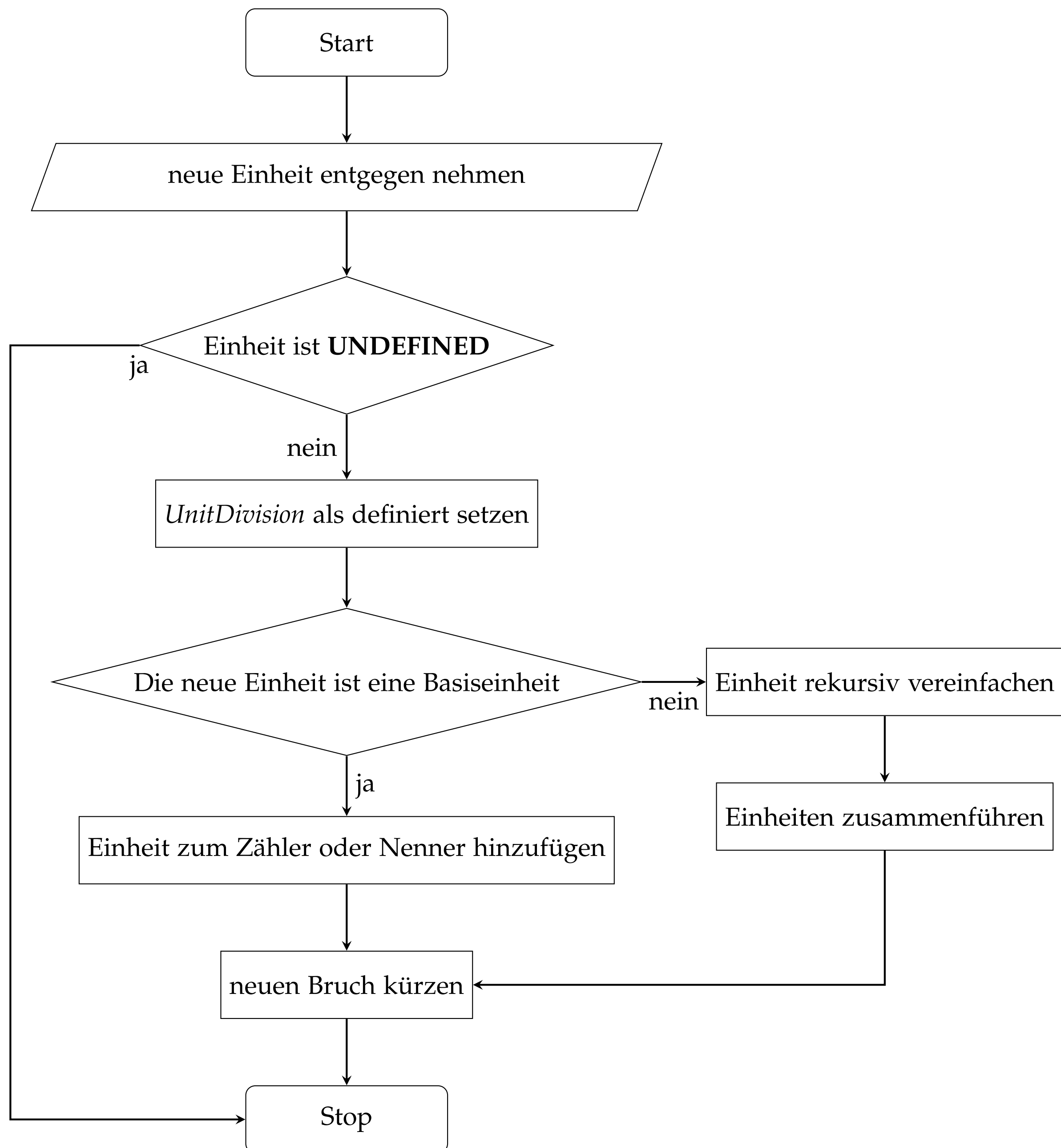


Abbildung 4.1: Vereinfachungsprozess bei der Kombination von Einheiten

Addition, Subtraktion Bei diesen Operationen dürfen die beiden Operanden keine unterschiedlichen Einheiten aufweisen. Es wird ein Fehler angezeigt, wenn zwei unterschiedliche Einheiten in der Operation verwendet werden. Ist mindestens eine Seite undefiniert oder sind die Einheiten identisch, ist die Operation zulässig. Ist die Operation zulässig und besitzt mindestens ein Operand eine Einheit, wird der gesamten Operation diese Einheit zugewiesen.

Vergleich Bei Vergleichsoperationen gelten die gleichen Regeln wie bei einer Addition oder Subtraktion. Das Ergebnis ist hierbei ist jedoch immer ein boolscher Wert ohne Einheit. Deshalb wird für die gesamte Operation im Gegensatz zu dem Punktrechnungen auch bei einer gültigen Verwendung von Einheiten der Wert *UNDEFINED* gesetzt.

Zuweisung Als erstes wird überprüft, ob die Variable, der ein Wert zugewiesen werden soll, bereits eine Einheit besitzt. Dieser Fall tritt ein, wenn die Variable eine Systemvariable mit festgelegter Einheit ist oder der Variable im Programmablauf bereits eine Einheit zugewiesen wurde. Sind die Einheiten der Variable und die Ergebniseinheit der Operation auf der rechten Seite des Zuweisungszeichens nicht identisch, wird die Operation als fehlerhaft markiert, anderenfalls ist die Operation zulässig. Besitzt die Variable auf der linken Seite der Zuweisung keine Einheit, wird ihr die Einheit des Ergebnisses der rechten Seite zugeordnet.

Multiplikation und Division Bei Punktrechnungen existieren keine ungültigen Kombinationen von Einheiten, es kann jedoch eine neue Einheit entstehen. Für diesen Fall wird die Einheit einer Variable als Bruch angesehen, der beliebig erweitert werden kann. Dies passiert nach dem in Kapitel 4.2.2 beschriebenen Konzept.

Modulo Modulo Operationen werden als gültig betrachtet, wenn beide Operanden die gleiche Einheit haben oder wenn der rechte Operand einheitenlos ist. Damit fällt die Modulo-Operation automatisch in die gleiche Klasse wie Subtraktion und Addition.

Boolsche Operatoren *AND*, *OR* und *XOR* können nicht nur auf Boolean-Werte angewendet werden. Vielmehr werden die Operanden bitweise logisch verknüpft. Damit ist es möglich, dass die Operanden typisiert sind. Ebenso wie bei den Vergleichsoperatoren können hier nur zwei identische Einheiten sinnvoll verrechnet werden, sodass das Ergebnis eine eindeutige Einheit besitzt.

Funktionsaufrufe Eine Ausnahme bilden Funktionsaufrufe. Hier wäre die Rückgabeeinheit von Bedeutung, sie lässt sich jedoch nicht einfach erkennen. Die Schwierigkeiten und mögliche Lösungsansätze sind in Kapitel 6.2 aufgeführt.

4.2.5 Genaue Einheiten

Wie bereits angedeutet stecken hinter jeder physikalischen Größe zwei mögliche Einheiten, eine aus dem metrischen und eine aus dem angloamerikanischen Maßsystem. Es liegt also nahe, die Analyse auf Einheiten statt auf physikalische Größen anzuwenden. Auf diese Weise könnten Fehler aufgedeckt werden, die sonst nicht gefunden werden. Würden bspw. ein Druckwert in *bar* mit einem in *Pascal* addiert, könnte diese Regelverletzung angezeigt werden.

Um in der Typanalyse die genauen Einheiten statt der allgemeinen physikalischen Größen zu berücksichtigen, müssen zwei Schritte durchgeführt werden:

1. Es müssen zusätzliche Basiseinheiten definiert werden.
2. Die Regeln im *UnitDeducer* müssen angepasst werden.

Die Menge der Basiseinheiten bezieht sich jetzt nicht mehr auf die physikalischen Größen, sondern auf die genauen Einheiten. Während bisher *TimeDay* (Zeit in Tagen), *TimeMin* (Zeit in Minuten) und *Timeys* (Zeit in Mikrosekunden) alle der Kategorie *Time* (Zeit in Sekunden) zugeordnet wurden, bilden sie jetzt eigene Basiseinheiten, die ohne Umrechnungsfaktor nicht aufeinander zurückgeführt werden können. Listing 4.5 zeigt einen Ausschnitt aus dem angepassten *UnitDeducer*. In dem Beispiel ist zu sehen, dass nun zwei verschiedene Klassen von Geschwindigkeiten berücksichtigt werden müssen. *Velocity_SIUnit* hat die Einheit $\frac{m}{s}$ während *Velocity* in $\frac{mm}{s}$ angegeben wird. An unterschiedlichen Stellen im Projekt können verschiedene Einheiten für die gleiche physikalische Größe nötig sein, um einen sinnvollen Wertebereich zu erhalten. Es ist aber semantisch nicht sinnvoll, $\frac{m}{s}$ und $\frac{mm}{s}$ zu vergleichen, ohne vorher die Einheiten anzupassen.

Das Problem bei dem Ansatz ist, dass mithilfe einer statischen Codeanalyse die Umrechnungsfaktoren nicht zuverlässig bestimmt werden können. Ist bei der Analyse die Umrechnung zwischen Einheiten nicht vollständig nachvollziehbar, generiert sie an einigen Stellen False Positives[2]. Listing 4.6 zeigt vier Beispiele von zulässigen Operationen. Bei der ersten Anweisung könnte die Konstante **10** auch aus dem statischen Kontext als Umrechnungsfaktor identifiziert werden. In der zweiten Zeile gestaltet es

[2]False Positive: Es wird fälschlicherweise ein Fehler angezeigt.

Listing 4.5: Ausschnitt aus der Klasse *UnitDeducer*

```
private static UnitDivision splitExactUnitUp(EIec61131SystemVariableUnit unit) {
        UnitDivision unitDivision = new UnitDivision();
        switch (unit) {
                /* Velocity_SIUnit -> Length(m) / Time(s) */
                case Velocity_SIUnit:
                        unitDivision.addToNumerator(Length);
                        unitDivision.addToDenominator(Time);
                        break;
                /* Velocity -> Stroke(mm) / Time(s) */
                case Velocity:
                        unitDivision.addToNumerator(Stroke);
                        unitDivision.addToDenominator(Time);
                        break;
                ...
                }
        return unitDivision;
}
```

sich schwieriger. Auch hier soll die **10** vermutlich einen Umrechnungsfaktor darstellen, der *cm* in *mm* konvertiert. Aufgrund der Vielzahl an Parametern ist die Zuordnung nicht eindeutig und es kann nicht sicher festgestellt werden, ob die Addition fehlerhaft ist. In der dritten Anweisung dient die Variablen **conversion_factor** als Umrechnungsfaktor. In diesem Fall kann jedoch bei einer statischen Analyse der Wert der Variable nicht sicher bestimmt werden, da der Wert selbst erst bei der Ausführung des Programms berechnet werden könnte. Hier stößt die statische Analyse an ihre Grenzen. Ein weiteres Problem sind Einheitenkonvertierungen mit der Hilfe von Funktionsaufrufen. Auch hier kann statisch die Rückgabeeinheit nicht ermittelt werden.

Listing 4.6: Gültige Operationen mit Umrechnungen zwischen Einheiten

```
position = sv_mm + sv_cm * 10;
position = sv_mm + sv_cm * 250.45 * 10 / 2;
position = sv_mm + sv_cm * coversion_factor;
sv_pressure_in_bar > pascalToBar(sv_pressure_in_pascal);
```

Aus diesem Grund wird die Typanalyse in dieser Arbeit auf physikalische Größen beschränkt und nicht mit den genauen Einheiten durchgeführt.

4.3 Parsen von *Structured Text* Statements

Teamscale besitzt bereits einen Parser für *Structured Text* Code. Dieser parst die Eingabe bis zur Anweisungsebene. Dabei wird ein Syntaxbaum (AST, Abk. abstract syntax tree) produziert, dessen Blätter einzelne Anweisungen sind.
Für die Typanalyse müssen die vorliegenden Anweisungen jedoch weiter untergliedert werden, um Operationen herausfiltern und auswerten zu können. Für diesen Zweck wird ein Statement-Parser implementiert. Er nimmt eine Anweisung als Liste von Token entgegen und gibt einen AST zurück, der die Operationen und ihre Reihenfolge widerspiegelt und dessen Blätter Variablen oder Literale sind. Anhand des entstandenen AST kann später ermittelt werden, ob eine Operation zulässig ist oder ob eine Meldung an den Programmierer generiert werden soll.

4.3.1 Der *CUP* Parser Generator

Der Statement-Parser wird mithilfe des *CUP* LALR Parser Generators[3] erstellt. *CUP* unterliegt der Open Source License und darf für die Arbeit frei verwendet werden. Die Grammatik wird in *.cup* Dateien definiert, aus denen der *CUP* Parser Generator einen Parser in Java erzeugt.

4.3.2 Grammatik für Anweisungen in *Structured Text*

Für die Grammatik können terminale und nichtterminale Symbole erstellt werden. Terminale sind in diesem Fall Symbole, die den einzelnen Token im Code entsprechen. Dazu zählen

- Variablen und Literale
- Zeichen wie Punkt, Komma, Semikolon und Klammern
- Operatoren wie $+, -, \geq, =, :=, MODULO, AND, XOR$ und
- Schlüsselwörter wie `RETURN, EXIT, CONTINUE, EMPTY`.

Zur leichteren Unterscheidung sind sie in der Grammatik in Großbuchstaben geschrieben.

Nichtterminale dienen als Hilfszustände, um den Parser rekursiv zu gestalten. In Listing 4.7 ist ein Ausschnitt der Grammatikregeln zu sehen. Ein Funktionsaufruf besteht aus einem *IDENTIFIER*, also dem Funktionsnamen, und einem Klammernpaar

[3] http://www2.cs.tum.edu/projects/cup/docs.php

(*LPAREN, RPAREN*). Optional kann zwischen den Klammern eine Variablenliste stehen, also die Parameter, die der Funktion übergeben werden. Eine Variablenliste wiederum besteht aus einer Aneinanderreihung von Variablen, Literalen oder Zuweisungen und Kommata. In dem Grammatikausschnitt stellen *IDENTIFIER, LPAREN, RPAREN* und *COMMA* Terminale dar, während *Function, VariableList, Variable, Literal* und *Assignment* nichtterminale Symbole sind. Mit dem aus der Grammatik erzeugten Parser können bspw. folgende Ausdrücke evaluiert werden:

```
fun()
add(a, 10)
calcNewPosition(sv_postion_x, sv_position_y, sv_position_z)
```

Hingegen nicht akzeptiert werden z.B. folgende Zeilen:

```
fun(a, b,)
add(a, b
```

Listing 4.7: *CUP* Grammatik

```
Function ::=
  IDENTIFIER LPAREN VariableList RPAREN
| IDENTIFIER LPAREN RPAREN
;
VariableList ::=
       VariableList COMMA VariableList
| Variable
| Literal
| Assignment
;
```

4.3.3 Operatorpräzedenz

In der Grammatik können Regeln für die Operatorpräzedenz festgelegt werden. In Listing 4.8 ist ein Beispiel zu sehen, in dem den Terminalen zugeordnet wird, ob sie links- oder rechtsbindend sind. Dabei sind die binären Operatoren linksbindend, d.h. der Operand auf der linken Seite des Operators wird vor dem rechten ausgewertet. Der unäre Operator *NOT* hingegen bindet das Symbol auf seiner rechten Seite. Auch die Reihenfolge ist dabei entscheidend. Je weiter unten ein Eintrag steht, desto größer ist die Operatorpräzedenz. Im Beispiel bindet *NOT* also stärker als *MULT*, *DIV* und *MOD*, die wiederum der gleichen Präzedenzklasse angehören. Alle Symbole binden stärker als *PLUS* und *MINUS*. Jeder Operator ist in einer solchen Regel aufgeführt.

Tabelle 4.2: Operatorpräzedenz in *Structured Text*

Klasse	Operation	Symbol
1	Klammern	()
2	Negation	$-$
2	Komplement	*NOT*
3	Potenzierung	**
4	Multiplikation	*
4	Division	/
4	Modulo	*MOD*
5	Addition	$+$
5	Subtraktion	$-$
6	Vergleich	$=, <>, >, <, <=, >=$
7	Boolsches Und	&, *AND*
8	Boolsches Xor	*XOR*
9	Boolsches Oder	*OR*

Listing 4.8: Regeln für die Operatorpräzedenz

```
precedence left PLUS, MINUS;
precedence left MULT, DIV, MOD;
precedence right NOT;
```

Das Festlegen dieser Regeln führt zu den folgenden gedachten Klammerungen:

```
a + b - c = (a + b) - c
NOT a OR b = (NOT a) OR b
a + b * c + NOT d * e = (a + (b * c)) + ((NOT d) * e)
```

Die Präzedenz für Operatoren in *Structured Text* sind in Tabelle 4.2 aufgeführt. Dabei sind Operatoren mit niedriger Klasse stärker bindend als Operatoren aus einer höheren Klasse. Operatoren mit der gleichen Präzedenzklasse binden gleich stark.[13]

4.3.4 Erstellen des AST

Zu jeder Regel in der *CUP*-Grammatik kann Java Code geschrieben werden, der ausgeführt wird, wenn diese Regel angewendet wird. Terminale werden in der Arbeit als *IToken* repräsentiert. Für den AST wurde eigens die Java-Klasse *AstNode* enwickelt. Jeder Knoten im AST ist eine Instanz dieser Klasse. Hier ist gespeichert, von welchem Typ der Knoten ist und welche *IToken* er beinhaltet. Tabelle 4.3 gibt einen Überblick über die möglichen Typen. Ein Knoten besitzt außerdem eine Einheit, die aber erst im

späteren Verlauf der Analyse gesetzt wird. Jeder Knoten besitzt eine Liste mit Kindern, die wiederum *AstNode*s sind. Auf diese Weise ist es möglich, eine Baumstruktur zu realiseren.

Ein Ausschnitt aus der *CUP*-Grammatik mit Java Anweisungen ist in Listing 4.9 zu sehen. Es wird beispielhaft an der *Expression*-Regel gezeigt, wie der AST beim Parsen einer Anweisung erstellt wird. Ein Ausdruck (*Expression*) kann bspw. eine Addition sein. Diese besteht aus einem *Plus*-Symbol und einem weiteren Ausdruck auf jeder Seite des Operators. Die Operanden können bspw. eine weitere Addition sein oder aber eine Variable oder ein Literal, also ein fester Zahlenwert. Steht auf einer Seite der Addition eine Variable, wird sie in der Regel `Variable` identifiziert und an die Ebene darüber weitergegeben (`RESULT = var;`). Bei der Variable *var* handelt es sich um einen *AstNode*, also einen Knoten im AST. In der `Expression`-Regel können somit *ex1* bzw. *ex2* als *AstNode* und das *Plus*-Symbol als *IToken* interpretiert werden. Es wird ein neuer Knoten vom Typ *ADDITION* erzeugt und *ex1* und *ex2* werden als seine Kinder gesetzt. Die Tokenliste des neuen Knotens besteht aus allen Token der Kindknoten sowie des *Plus*-Operators.
Alle Regeln des Parsers folgen diesem Schema. Auf diese Weise entsteht für eine Anweisung am Ende ein Syntaxbaum mit einer neuen Ebene für jede Regelanwendung.

Das Erzeugen des AST wird beispielhaft an der Anweisung

```
a := (b + c) * d >= e;
```

demonstriert. Abbildung 4.2 zeigt den AST, der nach dem Parsen entstanden ist. Dabei ist die gesamte Zeile als Statement zu betrachten, das im AST den Wurzelknoten bildet. Die Zuweisung ist die Operation, die als letztes ausgeführt wird. Sie befindet sich also direkt in der Ebene darunter. Das linke Kind der Zuweisung ist eine Variable, hier endet also der Ast. Das rechte Kind ist ein Vergleich mittels größer gleich, das wiederum zwei Kinder hat usw. Jedem Knoten ist ein Typ aus Tabelle 4.3 zugeordnet.

Neben den einfachen Operationen wie Additionen, Zuweisungen und Vergleichen müssen auch kompliziertere Anweisungen vom Parser berücksichtigt werden. Im Folgenden ist beschrieben, wie die Syntaxbäume für mehrdimensionale Arrayzugriffe und für Aufrufe von Funktionsblöcken entstehen.

Mehrdimensionaler Arrayzugriff Der Zugriff auf ein bestimmtes Feld eines mehrdimensionalen Arrays wird als Indexliste innerhalb einer Referenzierung realisiert (s. Listing 4.10)

Tabelle 4.3: Typen von *AstNode*s

Kategorie	Typ
Operationen	ADDITION SUBTRACTION MULTIPLICATION DIVISION MODULO EXPONENTIATION ASSIGNMENT BOOL_OPERATION COMPLEMENT COMPARISON
Literale	INTEGER_LITERAL BOOLEAN_LITERAL STRING_LITERAL FLOATING_POINT_LITERAL TIME_LITERAL
Schlüsselwörter	RETURN EXIT CONTINUE EMPTY
Sonstiges	STATEMENT VARIABLE FUNCTION VARIABLE_LIST ARRAY PARENTHESIS

Listing 4.9: *CUP*-Grammatik mit Java Code

```
Expression ::=
    Expression:ex1 PLUS:plus Expression:ex2
        {:
          List<IToken> token = new ArrayList();
          token.addAll(((AstNode) ex1).getTokens());
          token.add((IToken) plus);
          token.addAll(((AstNode) ex2).getTokens());
          AstNode node = new AstNode(EShallowEntityType.ADDITION, token);
          node.addChild((AstNode) ex1);
          node.addChild((AstNode) ex2);
          RESULT = node;
        :}
...
 | Variable:var
        {:
          RESULT = var;
        :}
  | Literal:lit
        {:
          RESULT = lit;
        :}
;
Variable ::=
        IDENTIFIER:id
        {:
          List<IToken> token = new ArrayList();
          token.add((IToken) id);
          RESULT = new AstNode(EShallowEntityType.VARIABLE, token);
        :}
...
;
```

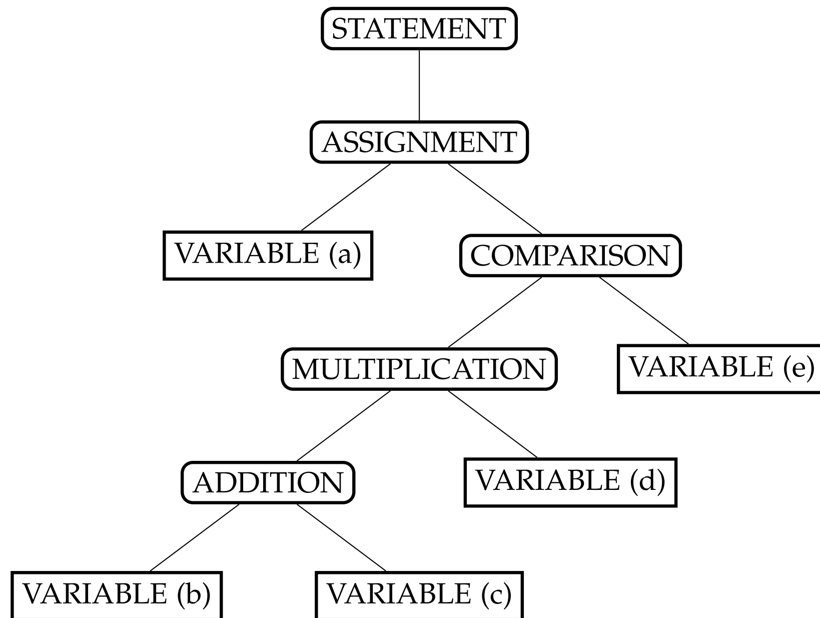


Abbildung 4.2: AST von `a := (b + c) * d >= e;`

Listing 4.10: Mehrdimensionaler Arrayzugriff

```
VAR
arr: ARRAY [0..10, 0..20, 0..1] OF INT;
END_VAR
...
arr[2, 5, 0] := 256;
```

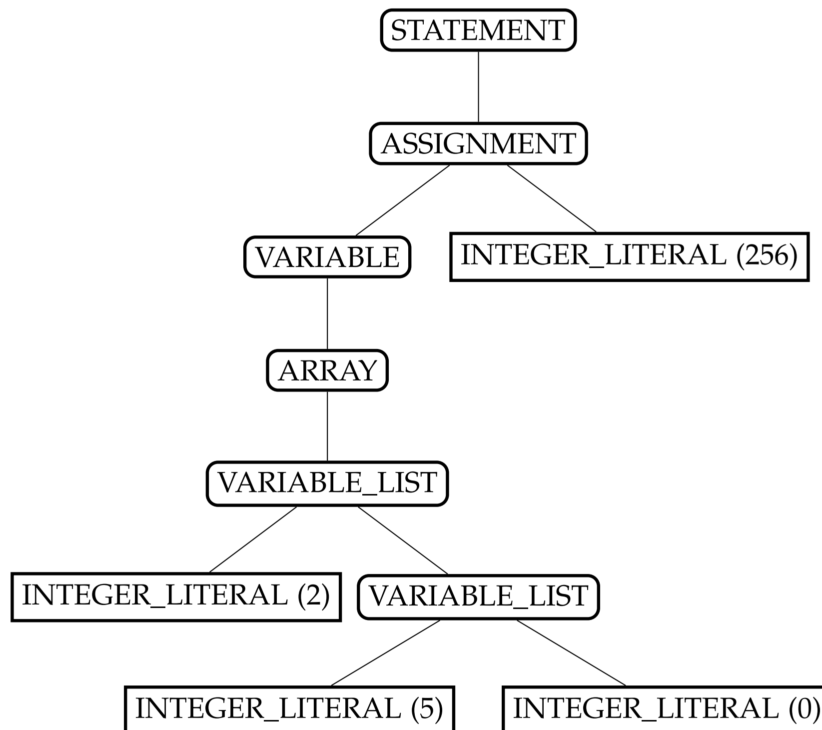


Abbildung 4.3: AST von `arr[2, 5, 0] := 256;`

Für die Anweisung `arr[2, 5, 0] := 256;` ensteht der AST aus Abbildung 4.3. Dabei wird die gesamte linke Seite der Zuweisung als Variable betrachtet, die erst anschließend als Array identifiziert wird. Das ist für die spätere Analyse wichtig. Die Indexliste ist eine Variablenliste, die aus dem Integer-Literal *2* und einer weiteren Variablenliste mit den Integern *5* und *0* als Kindern besteht.

Aufruf eines Funktionsblock Ein für *Structured Text* spezifisches Konstrukt ist der Aufruf eines Funktionsblocks. Das Setzen der Eingangswerte findet hierbei direkt im Aufruf statt. Bei der Anweisung `fb_pr(IN:=bIn1, T:=T#300ms);` wird der Funktionsblock *fb_pr* mit den Eingängen *IN* und *T* aufgerufen, denen die Boolean Variable *bIn1* und das Zeitliteral *T#300ms* zugewiesen werden. Für diesen Ausdruck ensteht der in Abbildung 4.4 dargestellte AST.

4.3.5 Der Scanner

Die zu analysierende Eingabe bekommt der Parser von einem Scanner. Im Normalfall liest der Scanner Text aus einer Datei ein und strukturiert die Eingabe als Token, mit

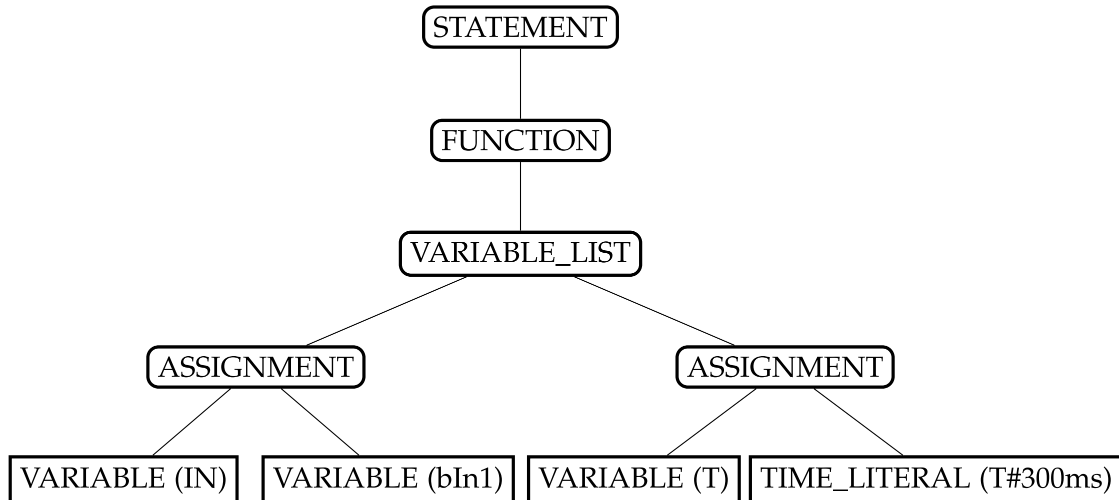


Abbildung 4.4: AST von `fb_pr(IN:=bIn1, T:=T#300ms);`

denen der Parser arbeiten kann. In dieser Arbeit liegen die einzelnen Anweisungen bereits als Tokenliste vor. Es wird also ein Scanner benötigt, der lediglich die vorhandenen Token in für den Parser lesbare Token umwandelt und ihm diese sowie ihre Position in der Anweisung übergibt. Hierfür wird ein selbstimplementierter Scanner verwendet werden, der das Scanner Interface des *CUP*-Parsers implementiert.

4.3.6 IF, CASE, FOR, REPEAT und WHILE

Um den Parser möglichst einfach zu halten, beinhaltet er keine Regeln für IF und CASE Blöcke sowie für FOR, REPEAT und WHILE Schleifen. Die genannten Konstrukte werden gefiltert, bevor einzelne Anweisungen an den Parser übergeben werden. Als erstes wird die Bedingung extrahiert, wenn eine vorhanden ist. Nach dem Anhängen eines Semikolons kann die Bedingung als einzelne Anweisung betrachtet und geparst werden. Anschließend werden dem Parser einzeln die Anweisungen in den Körpern der Konstrukte übergeben. Bei einer IF-Abfrage werden die umrandeten Anweisungen einzeln geparst:

```
IF t > 0 THEN
  sv_time := sv_time + t;
  sv_velocity := sv_length / sv_time;
END_IF
```

4.3.7 Generieren des Parsers

Um aus der *ExpressionParser.cup* Datei mit der Grammatik den Parser zu generieren, muss die Datei *java-cup-11.jar* von der Dokumentationsseite des *CUP*-Parsers[20] heruntergeladen werden. In dieser Arbeit wird mit Version 11 gearbeitet. Die *.jar* Datei muss in den Ordner kopiert werden, in dem sich die Datei *ExpressionParser.cup* befindet. Anschließend kann der Parser mit folgendem Befehl erzeugt werden:

```
java -jar java-cup-11.jar -expect 2 „ExpressionParser.cup“
```

Es werden zwei Dateien generiert. Die Datei *sym.java* enthält Token für alle Terminale, die später von dem Scanner verwendet werden. In der Datei *parser.java* ist der eigentliche Parser implementiert. Mit dem Flag *-expect* kann die Anzahl der erwarteten Konflikte angegeben werden. In diesem Fall können zwei Shift-Reduce Konflikte nicht vermieden werden. Sie werden automatisch zugunsten eines Shifts gelöst.

4.4 Die Typanalyse

Im ersten Schritt werden die Systemvariablen nach dem breschriebenen Schema aus den *.sv* Dateien extrahiert. Anschließend wird eine *.pu* Datei nach der anderen der Analyse unterzogen. Innerhalb einer Datei wird jeder *ALGORITHM* einzeln betrachtet und untersucht. Zu Beginn eines *ALGORITHM*s wird eine neue leere Liste für die lokalen Variablen erstellt, denen im Lauf der Analyse eine Einheit zugeordnet wird. Anschließend werden für jede Anweisung die folgenden zwei Schritte ausgeführt:

1. Das Statement wird geparst und es wird daraus ein AST generiert.
2. Der AST wird in einem Bottom-Up Ansatz traversiert, also von den Blättern aus bis zur Wurzel. Im Zuge dessen werden den Knoten Einheiten zugewiesen. Dabei wird aus den Kindknoten die Einheit des Elternknotens abgleitet. Im gleichen Schritt findet auch die Überprüfung der Operation auf ihre Gültigkeit statt. Wird eine Regelverletzung festgestellt, führt sie direkt zur Generierung eines Fehlers. In diesem Fall erhält der Elternknoten das Label *UNDEFINED*.

Wird bspw. die Anweisung $[Force] := [Pressure] + [Pressure]$ mit den gegebenen physikalischen Größen überprüft, wird als erstes der AST generiert, der in Abbildung 4.5 zu sehen ist. Hier ist auch in vier Schritten beschrieben, in welcher Reihenfolge die Einheiten gefolgert werden. So werden erst die Einheiten der Variablen gesetzt, bevor Ebene für Ebene die Operationen der Elternknoten überprüft werden. Für die Addition wird erst die Konsistenz der Operanden überprüft. Da beide Variablen einen Druck

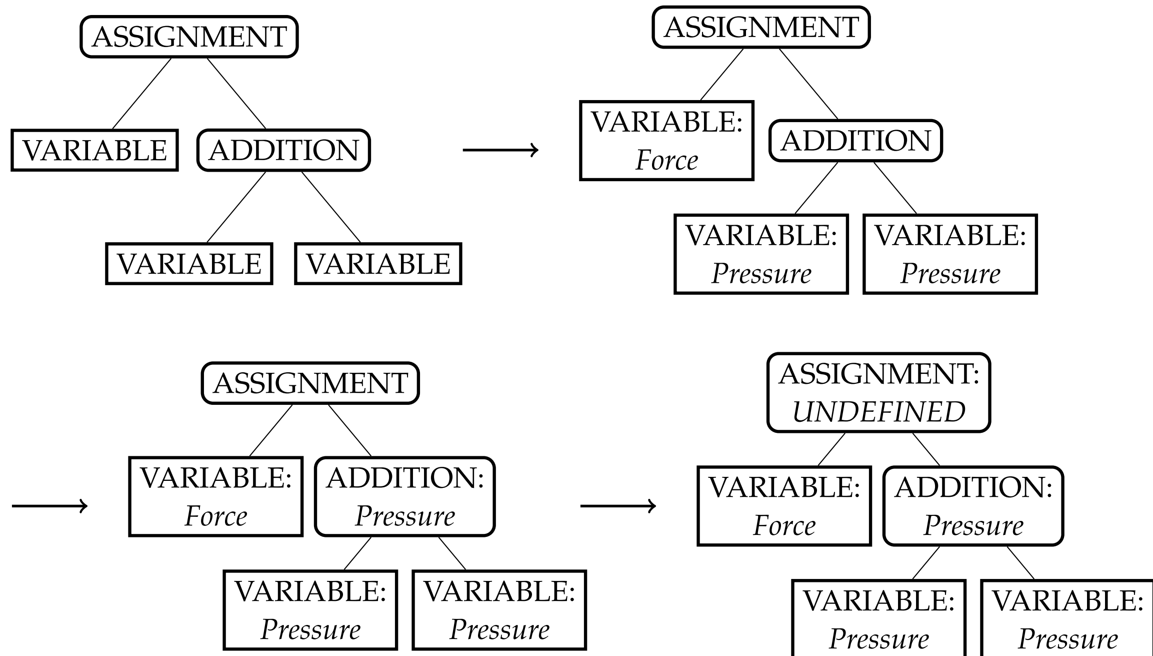


Abbildung 4.5: Folgerung der Einheiten in einem AST

repräsentieren, ist diese sichergestellt und dem Additionsknoten selbst wird die gleiche Einheit zugewiesen. In der Ebene darüber wird einer Variable mit der Einheit *Force* ein Druckwert zugewiesen. Die Operation führt zu einer Regelverletzung, was zur Folge hat, dass direkt ein Fehler gemeldet wird und der Zuweisungsknoten als *UNDEFINED* gesetzt wird.

In *Teamscale* wird ein identifizierter Regelverstoß als Finding bezeichnet. Die Typanalyse generiert vier verschiedene Arten von Findings mit den entsprechenden Beschreibungen.

Inconsistent use of unit in an operation: ... Mithilfe dieses Findings werden die eigentlichen Ergebnisse der Typanalyse angezeigt, also Regelverletzungen bei Operationen mit physikalischen Größen.

Warning: There is no unit defined for the system variable ... Mithilfe des Findings wird der Nutzer gewarnt, wenn einer einheitenlosen Systemvariable eine Einheit zugewiesen wird. Es handelt sich hierbei nicht um einen Fehler, es kann aber dazu führen, dass Folgefehler an anderer Stelle nicht erkannt werden. Wird der gleichen

Systemvariable in einer anderen POU eine Einheit zugewiesen, die von der bisherigen abweicht, kann diese Operation nicht als Fehler identifiziert werden. Außerdem ist die allgemeine Idee hinter Systemvariablen verletzt, wenn die Einheit nicht in der *.sv* Datei definiert wird. Mit hoher Wahrscheinlichkeit hat der Entwickler vergessen, die Einheit zu setzen. Darauf soll ihn das Finding hinweisen.

This statement could not be parsed correctly Falls ein Statement nicht korrekt geparst werden kann, wird dieses Finding angezeigt. Das kann passieren, wenn Syntaxfehler im *Structured Text* Code sind oder seltene Konstrukte verwendet werden, die dem Parser nicht bekannt sind. Das Finding ist nur für die Testphase gedacht, um die Korrektheit und die Vollständigkeit des Parsers zu evaluieren.

The subtype ... of this statement is not known Dieses Finding tritt bspw. auf, wenn in den *Structured Text* Code *IEC 61131-3 SFC (Sequential Function Chart)* Code eingebettet ist. Einzelne Algorithmen können auch in einer anderen *IEC 61131-3* Sprache geschrieben sein und als Baustein aus dem ST Code heraus aufgerufen oder instantiiert werden. Der umgekehrte Fall ist auch möglich, also dass *Structured Text* Bausteine aus anderen Sprachen heraus aufgerufen werden. *SFC* ist eigentlich eine grafische Programmiersprache, sie kann jedoch auch als textuelle Struktur in den Projektdateien dargestellt werden. Dabei werden jedoch Schlüsselwörter benötigt, die nicht in *Structured Text* vorkommen und deshalb nicht vom Parser erkannt werden. Auch dieses Finding dient nur Forschungszwecken und wird deaktiviert, wenn die Analyse auf Kundenprojekten durchgeführt wird.

4.5 Ausführen der Typanalyse in *Teamscale*

Nachdem nun alle Teilkomponenten vorbereitet sind, kann die eigentliche Typanalyse stattfinden. Um diese in *Teamscale* durchzuführen, muss als erstes das Analyseprofil angepasst werden. Dafür kann im Reiter *Project -> Analysis Profile* das *IEC-61131-3 ST* Profil editiert werden. Damit die Typanalyse aktiviert wird, muss der Custom Check *Inconsistent Units of Systemvariables* in der Kategorie *Comprehensibility* ausgewählt sein (s. Abbildung 4.6).

Anschließend kann das zu analysierende Projekt angelegt werden. Dafür wird ein Projektname vergeben und das Analyseprofil *IEC 61131-3 ST (default)* eingestellt (s. Abbildung 4.7). Nachdem der Projektordner mit dem Projekt verknüpft wurde, kann es erstellt werden. Danach beginnt *Teamscale* direkt mit der Analyse.

Abbildung 4.6: Custom Check aktivieren

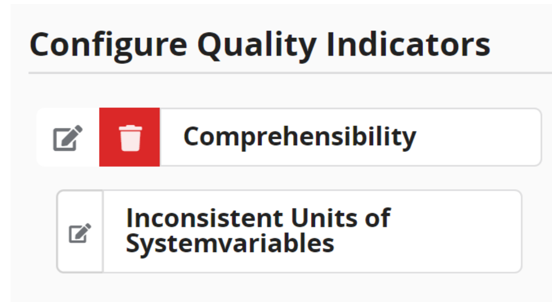
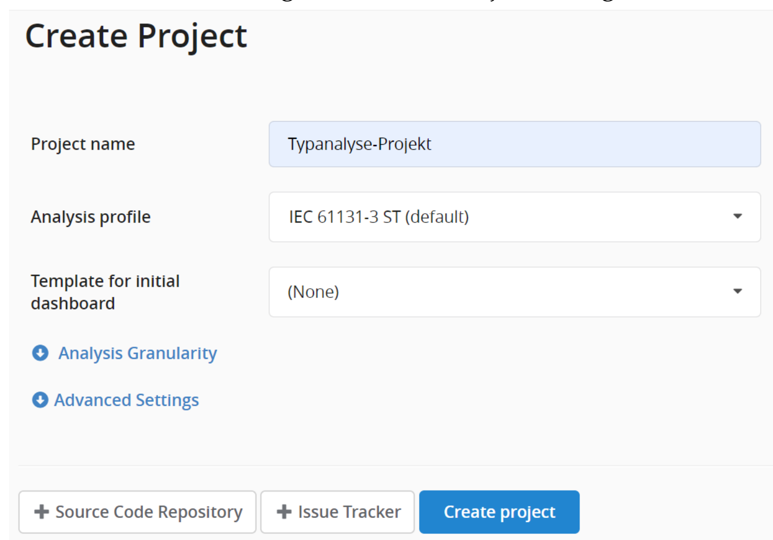


Abbildung 4.7: Neues Projekt anlegen



Nachdem die Analyse abgeschlossen ist, sind die Ergebnisse im *Findings*-Tab der *Teamscale* Weboberfläche zu sehen. In der Übersicht werden alle Findings aufgelistet. Einzelne Findings können durch einen Klick darauf genauer inspiziert werden. In der Detailansicht ist eine genaue Beschreibung zu finden, in der auch die betroffene Stelle im Code zu sehen ist.

Anhand eines Beispielprojektes werden die vier verschiedenen Finding-Typen demonstriert. Das Projekt besteht aus einer *.sv* Datei mit den Systemvariablendeklarationen und einer *.pu* Datei mit zwei POUs. Listing 4.11 zeigt den Inhalt der Datei *Typanalyse.pu*. Zu Beginn werden alle Systemvariablen aufgelistet, die innerhalb der Datei verwendet werden. Es folgt der *ALGORITHM* **PipeFlowRate**, der in mehreren Schritten die Berechnung eines Durchflusses (Volumenstroms) durch eine Rohrleitung implementiert. In die Berechnung wurden drei Fehler eingebaut. Es folgt der *PROCESS_ALGORITHM* **pInit**, der eine POU in der *IEC 61131-3* Sprache *Sequential Function Chart* repräsentiert. In Listing 4.12 ist der Inhalt der dazugehörigen *.sv* Datei zu sehen. Hier ist für jede Systemvariable außer **sv_volume** eine Einheit festgelegt. Wird nun die Typanalyse auf den Projektdateien ausgeführt, werden sechs Findings generiert (s. Abbildung 4.8).

Listing 4.11: Typanalyse.pu

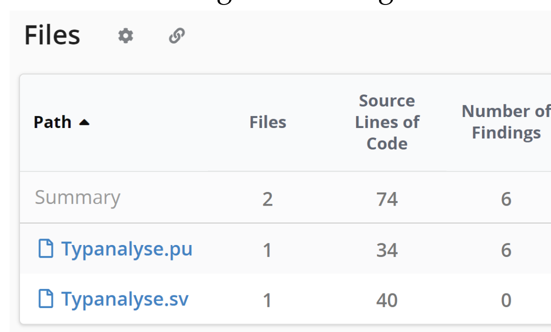
```
1  SYSTEM_VAR
2         sv_positionRight : INT;
3         sv_postionLeft : INT;
4         sv_postionAngleLeft : INT;
5         sv_posDiff : INT;
6         sv_deltaTime : INT;
7         sv_crossSectionalPipeArea : REAL;
8         sv_PipeRadius : REAL;
9         sv_flowRate : REAL;
10        sv_flowVelocity : REAL;
11        sv_volume : REAL;
12
13 END_VAR
14
15 ALGORITHM PipeFlowRate
16
17        IF sv_positionRight > sv_postionAngleLeft THEN
18               // calculate flow velocity
19               sv_posDiff := sv_positionRight - sv_postionLeft;
```

```
20              sv_flowVelocity := sv_posDiff / sv_deltaTime;
21
22              // calculate flow rate
23              sv_crossSectionalPipeArea := sv_PipeRadius ** 2 * PI;
24              sv_flowRate := sv_crossSectionalPipeArea + sv_flowVelocity;
25
26              // calculate volume within the last minute
27              sv_volume := sv_flowRate * 60;
28
29       ELSIF sv_positionRight = sv_postionLeft THEN
30              sv_flowVelocity := 0
31              flowRateWarning();
32       ELSE
33              flowRateError();
34       END_IF
35
36 END_ALGORITHM
37
38 PROCESS_ALGORITHM pInit ON TaskP
39
40       (* eingebetteter SFC Code *)
41       INITIAL_STEP Step1:
42       Action1 (N);
43       ;END_STEP
44
45       ACTION Action1:
46       WaitUntilFinished();
47       ;END_ACTION
48
49 END_ALGORITHM
```

Listing 4.12: Typanalyse.sv

```
1  %IMPORT_OVER_LISTFILE_SOURCE
2          Angle Area, FlowRate, Length, Time, Velocity, Volume
3  END_IMPORT
4
5  %SYSTEMVAR_DECL
6          sv_positionRight : INT %UNIT Length;
7          sv_positionLeft : INT %UNIT Length;
8          sv_postionAngleLeft : INT %UNIT Angle;
9          sv_posDiff : INT %UNIT Length;
10         sv_deltaTime : INT %UNIT Time;
11         sv_crossSectionalPipeArea : INT %UNIT Area;
12         sv_PipeRadius : INT %UNIT Length;
13         sv_flowRate : INT %UNIT FlowRate;
14         sv_flowVelocity : INT %UNIT Velocity;
15         sv_volume : REAL;
16 %END
```

Abbildung 4.8: Finding Metrik



Files

Path ▲	Files	Source Lines of Code	Number of Findings
Summary	2	74	6
Typanalyse.pu	1	34	6
Typanalyse.sv	1	40	0

Der erste Fehler mit der Beschreibung *Inconsistent use of unit in an operation: COMPARISON of sv_positionRight (unit: Length) and sv_postionAngleLeft (unit: Angle)* wird in Zeile 17 der *.pu* Datei angezeigt. Der Entwickler wollte in dieser Zeile eigentlich die rechte mit der linken Position vergleichen, um die Plausibilität zu prüfen. Statt der Variable **sv_positionLeft** hat er jedoch versehentlich die Variable **sv_positionAngleLeft** verwendet, die einen Winkel anstatt eines Längenmaßes widerspiegelt. An dieser Stelle wird ein Regelverstoß gemeldet und der Entwickler so auf seinen Fehler aufmerksam gemacht.

Inconsistent use of unit in an operation: COMPARISON of sv_positionRight (unit: Length) and sv_postionAngleLeft (unit: Angle)

Comprehensibility › Inconsistent Units of Systemvariables › measurement unit check

Inconsistent use of measurement units

In Typanalyse.pu:17

```
 7        sv_crossSectionalPipeArea : REAL;
 8        sv_PipeRadius : REAL;
 9        sv_flowRate : REAL;
10        sv_flowVelocity : REAL;
11        sv_volume : REAL;
12
13    END_VAR
14
15    ALGORITHM PipeFlowRate
16
17    IF sv_positionRight > sv_postionAngleLeft THEN
18        // calculate flow velocity
19        sv_posDiff := sv_positionRight - sv_postionLeft;
```

In Zeile 24 wird ein weiteres Finding generiert. Hier wird fälschlicherweise eine Fläche mit einem Geschwindigkeitswert addiert, was zu der Nachricht *Inconsistent use of unit in an operation: ADDITION of sv_crossSectionalPipeArea (unit: Area) and sv_flowVelocity (unit: Velocity)* führt. Damit die Berechnung stimmt, müsste statt der Addition eine Multiplikation durchgeführt werden, was der Entwickler an dieser Stelle vermutlich tun wollte.

Inconsistent use of unit in an operation: ADDITION of sv_crossSectionalPipeArea (unit: Area) and sv_flowVelocity (unit: Velocity)

Comprehensibility › Inconsistent Units of Systemvariables › measurement unit check

Inconsistent use of measurement units

In Typanalyse.pu:24

```
14
15   ALGORITHM PipeFlowRate
16
17   IF sv_positionRight > sv_postionAngleLeft THEN
18       // calculate flow velocity
19       sv_posDiff := sv_positionRight - sv_postionLeft;
20       sv_flowVelocity := sv_posDiff / sv_deltaTime;
21
22       // calculate flow rate
23       sv_crossSectionalPipeArea := sv_PipeRadius ** 2 * PI;
24 |     sv_flowRate := sv_crossSectionalPipeArea + sv_flowVelocity;
25
26       // calculate volume within the last minute
27       sv_volume := sv_flowRate * 60;
28
```

Es folgt ein Finding in Zeile 27 mit der Beschreibung *Warning: There is no unit defined for the system variable sv_volume*. Die Warnung wird erzeugt, weil der Systemvariable **sv_volume** eine Einheit zugewiesen wird, obwohl für sie keine definiert ist. Der Entwickler wird darauf hingewiesen, dass die entsprechende Einheit in der *.sv* Datei vergeben werden kann, um eventuelle spätere Fehler zu vermeiden.

Warning: There is no unit defined for the system variable sv_volume

Comprehensibility › Inconsistent Units of Systemvariables › measurement unit check

Inconsistent use of measurement units

In Typanalyse.pu:27

```
17   IF sv_positionRight > sv_postionAngleLeft THEN
18       // calculate flow velocity
19       sv_posDiff := sv_positionRight - sv_postionLeft;
20       sv_flowVelocity := sv_posDiff / sv_deltaTime;
21
22       // calculate flow rate
23       sv_crossSectionalPipeArea := sv_PipeRadius ** 2 * PI;
24       sv_flowRate := sv_crossSectionalPipeArea + sv_flowVelocity;
25
26       // calculate volume within the last minute
27 |     sv_volume := sv_flowRate * 60;
28
```

Beim Setzen der Fließgeschwindigkeit auf den Wert 0 fehlt am Ende der Zeile 30 das Semikolon. Die Typanalyse kann für diese Anweisung keine Aussage zu möglichen Regelverletzungen treffen. Die Zeile wird mit der Nachricht *This statement could not be parsed correctly* markiert.

This statement could not be parsed correctly

Comprehensibility › Inconsistent Units of Systemvariables › measurement unit check

Inconsistent use of measurement units

In Typanalyse.pu:30

```
20       sv_flowVelocity := sv_posDiff / sv_deltaTime;
21
22       // calculate flow rate
23       sv_crossSectionalPipeArea := sv_PipeRadius ** 2 * PI;
24       sv_flowRate := sv_crossSectionalPipeArea + sv_flowVelocity;
25
26       // calculate volume within the last minute
27       sv_volume := sv_flowRate * 60;
28
29   ELSIF
30 |     sv_flowVelocity := 0
31       flowRateWarning();
32   ELSE
```

Der *PROCESS_ALGORITHM* am Ende der Datei ist nicht in *Structured Text* geschrieben und kann deshalb nicht von der Typanalyse berücksichtigt werden. Das Finding mit der Beschreibung *The subtype ACTION of statement is not known* in Zeile 45-47 zeigt diesen Umstand an.

The subtype ACTION of statement is not known

Comprehensibility › Inconsistent Units of Systemvariables › measurement unit check

Inconsistent use of measurement units

In Typanalyse.pu:45-47

```
35
36    END_ALGORITHM
37
38    PROCESS_ALGORITHM pInit ON TaskP
39
40    (* eingebetteter SFC Code *)
41    INITIAL_STEP Step1:
42        Action1 (N);
43    ;END_STEP
44
45    ACTION Action1:
46        WaitUntilFinished();
47    ;END_ACTION
48
49    END_ALGORITHM
50
```

5 Evaluierung

Um die Forschungsfragen ausführlich untersuchen und beantworten zu können, wird eine Evaluierung der Typanalyse durchgeführt. Dafür wird sie auf einem produktiven System getestet. Die Ergebnisse sollen eine Bewertung der Funktionlität sowie der Effizienz ermöglichen. Darüber hinaus wird im Gespräch mit einem langjährigen Entwickler im Bereich der SPS-Programmierung eruiert, ob der Einsatz der Typanalyse im Unternehmen denkbar ist. Im Folgenden sind die zu untersuchenden Fragestellungen noch einmal aufgelistet.

5.1 Forschungsfragen

1. Ist eine statische messgrößenbasierte Typanalyse in *Structured Text* auf Basis von Systemvariablen möglich?
2. Findet die Analyse potenzielle Fehler auf echtem Projektcode?
3. Kann eine solche Typanalyse effizient umgesetzt werden?
4. Sind die Ergebnisse der Analyse für Entwickler in der Praxis nützlich?

Aufgrund der Ergebnisse in Kapitel 4 kann die erste Forschungsfrage bereits mit einem *Ja* beantwortet werden. Eine statische messgrößenbasierte Typanalyse in *Structured Text* ist auf der Basis von Systemvariablen generell möglich und konnte realisiert werden. Um die Funktionalität und Performance zu evaluieren, wird ein Test konzipiert, der im Folgenden beschrieben ist.

5.2 Studienobjekt

Der Test wird auf der Codebasis eines Maschinenbauunternehmens durchgeführt. Es handelt sich dabei um eine Software für Spritzgussmaschinen, die sich im produktiven Einsatz befindet. Das Projekt beinhaltet Konfigurationen und Anpassungen für verschiedene Maschinentypen und umfasst damit ca. 30.000 Dateien. Für die Analyse relevant sind alle Dateien mit den Endungen *.sv* und *.pu*. Davon existieren etwa 9.900 mit insgesamt ca. 1,6 Millionen Zeilen Code. Das Projekt ist damit umfangreich

genug, um Aussagen über die Funktionalität und Performance der Masterarbeit zu ermöglichen.

5.3 Studienaufbau

Für den Test wird eine eigene *Teamscale*-Instanz auf einem Rechner im Firmennetz des Maschinenbauunternehmens gestartet, die auf das SVN-Repository mit der Maschinensoftware zurückgreift. In *Teamscale* wird im Analyseprofil die Typanalyse aktiviert und alle anderen Checks deaktiviert, sodass nur die messgrößenbasierte Analyse ausgeführt wird. Nachdem die Ergebnisse vorliegen, werden sie stichprobenartig eingesehen und bewertet.
Insgesamt werden zwei Testdurchläufe durchgeführt, zwischen denen sowohl Anpassungen am Parser als auch an der Logik der Typanalyse selbst vorgenommen werden. Darüber hinaus ist die Einführung des *Warning*-Findings zu erwähnen, das auf Wunsch des Unternehmens für den zweiten Durchlauf implementiert wird. Wird also einer einheitenlosen Systemvariable eine Einheit zugewiesen, führt das zu einem Finding. Es soll den Entwickler darauf hinweisen, dass bei der Definition der Systemvariable eine Einheit festgelegt werden sollte.

5.4 Ergebnisse

Bei der ersten Analyse wurden **356** Findings für die inkonsistente Verwendung von Einheiten generiert. Sie wurden stichprobenartig bewertet und in vielen Fällen waren es tatsächliche Fehler. Jedoch wurden auch Findings erzeugt, die nicht oder nur indirekt als Fehler bewertet werden können. Aufgrund der Ergebnisse der zweiten Analyse kann die Menge der False Positives auf etwas über die Hälfte der Findings geschätzt werden. Einige Beispiele für diese False Positives sind im Folgenden aufgelistet.

IF/ELSE: In der Analyse wird die Reihenfolge der Anweisungen berücksichtigt. Wenn einer Variable eine Einheit zugewiesen ist, wird in den folgenden Zeilen mit dieser Einheit weitergerechnet. In einem IF/ELSE Statement kann das zu einem falschen Finding führen, wenn einer undefinierten Variable im IF-Zweig eine andere Einheit zugewiesen wird als in ELSE-Zweig. In der Analyse werden die Zuweisungen hintereinander bearbeitet, obwohl beim Ausführen des Codes nur einer der beiden Zweige durchlaufen wird. In Listing 5.1 ist ein solches Beispiel zu sehen. Der lokalen Ausgangsvariable **result** wird abhängig von der Eingangsvariable **b** entweder die physikalische Größe Druck im IF-Zweig oder Geschwindigkeit im ELSE-Zweig zugewiesen. Die Typanalyse erzeugt für die Anweisung **result := sv_velocity;** ein Finding, weil der gleichen Variable

davor anscheinend ein Druckwert zugewiesen wurde. Bei der Ausführung wird aber offensichtlich nur eine der Operationen ausgeführt und niemals beide. Das Finding ist dementsprechend in diesem Fall ein False Positive.

Listing 5.1: False Positive im IF/ELSE-Zweig

```
VAR_INPUT
b : BOOL;
END_VAR
VAR_OUTPUT
result : INT;
END_VAR

IF b THEN
result := sv_pressure;
ELSE
result := sv_velocity;
END_IF
```

Das False Positive kann vermieden werden, indem die Analyse in solchen Fällen nur innerhalb eines Blocks stattfindet und am Ende keine Ergebniseinheit übernommen wird. Mit der alleinigen Hilfe einer statischen Codeanalyse kann keine Aussage darüber getroffen werden, ob der IF- oder der ELSE-Zweig durchlaufen wird und mit welcher Einheit somit fortgefahren werden soll. In diesem Zusammenhang stellt sich aber die Frage, ob das Verhalten nicht wenigstens zu einer Warnung führen soll. Ist die Einheit der Variable nicht eindeutig, kann es im folgenden Verlauf zu Fehlern kommen, die durch die Typanalyse nicht erkannt werden können.

Prozentwerte: In der ersten Analyse wurde Prozent als eigene physikalische Größe betrachtet. Dabei wurde nicht bedacht, dass sie als Zahlenwert ohne Einheit verwendet werden kann, da bspw. bei einer Punktrechnung die bestehende Einheit nicht verändert wird. Prozentwerte können stattdessen wie undefinierte Variablen behandelt werden.

Im folgenden Schritt wurde die Typanalyse so angepasst, dass die beschriebenen False Positives berücksichtigt und vermieden werden. Außerdem kam es an ein paar Stellen zu Parserfehlern, da einige *Structured Text* Konstrukte nicht bekannt waren. Auch hier wurden Anpassungen vorgenommen, bevor die Analyse erneut ausgeführt wurde.

Im zweiten Durchlauf wurden **114** inkonsistente Verwendungen von Einheiten identifiziert. Die Reduktion der Anzahl lässt sich mit der Fehlerbehebung für die Sonderfälle

erklären, die im ersten Durchgang zu False Positives führten. Die Rate für tatsächliche Fehler war im zweiten Durchgang sehr hoch, es waren nur noch vereinzelt False Positives zu finden. Zusätzlich produzierte die Typanalyse **320** Warnungen bei der Verwendung von einheitenlosen Systemvariablen. Dieser Wert wäre vermutlich deutlich niedriger, wenn Boolsche Variablen bei der Analyse ignoriert werden würden. Diese haben im Normalfall keine Einheit und folglich ist hier keine Warnung notwendig. Die Anpassung hierfür wurde nach der zweiten Analyse vorgenommen.

Die Dauer des gesamten zweiten Testdurchlaufs bis zum Vorliegen der Ergebnisse lässt sich auf 2:10 Stunden festlegen. Das Parsen der Systemvariablen und die Typanalyse selbst nehmen davon ca. 48 Minuten in Anspruch.

5.5 Diskussion

Basierend auf diesen Ergebnissen lässt sich die zweite Forschungsfrage mit einem *Ja* beantworten. Eine messgrößenbasierte Typanalyse in *Structured Text* ist auf der Basis von Systemvariablen ist nicht nur möglich, sie findet auch fehlerhafte Typzuweisungen auf echtem Projektcode und kann den Entwickler somit vor potenziellen Fehlern warnen. Nach dem ersten Testdurchlauf konnten die meisten der generierten False Positives behoben werden, was im zweiten Durchlauf eine sehr geringe False Positive Rate zur Folge hatte. Daneben konnte ein weiteres Finding eingeführt werden, das den Entwickler bereits bei der Verwendung von einheitenlosen Systemvariablen warnt und Fehlern somit frühzeitig vorbeugen kann.

Neben der Funktionalität spielt auch die Performance eine wichtige Rolle für die Qualität der Typanalyse. Die initiale Analyse nimmt relativ viel Zeit in Anspruch. Sobald diese aber abgeschlossen ist, werden in Zukunft nur geänderte Dateien neu analysiert. Damit liegen die Ergebnisse nach Änderungen im Code deutlich schneller vor als zu Beginn.

Ist das zu analysierende Projekt als Git oder SVN Repository angelegt, wird in *Teamscale* zwischen drei Analysephasen unterschieden:

- **Initialanalyse**: Hier wird nur der Commit untersucht, der als Startpunkt ausgewählt wurde.
- **Historienanalyse**: Ist der initiale Commit analysiert, werden Schritt für Schritt weitere Commits berücksichtigt. Dabei werden pro Commit nur geänderte Dateien neu untersucht.

Tabelle 5.1: Dauer der Initial- und Historienanalyse

Aufgabe	Initialanalyse	Historienanalyse
Extraktion der Systemvariablen	3 min	1 s
Typanalyse	45 min	1h
Gesamte Analyse	2:10 h	1:05 h

- **Live-Analyse**: Die beiden vorherigen Analysen werden beim Erzeugen eines Projektes gestartet. Die Live-Analyse hingegen bearbeitet neue Commits, die nach der Historienanalyse hinzukommen.

Die initiale Analyse ist in diesem Fall am aussagekräftigsten, da die Dauer der Historien- und der Live-Analyse von der Anzahl und der Größe der Commits im Repository abhängt. Tabelle 5.1 gibt einen Überblick über die Dauer der Initial- und der Historienanalyse des zweiten Testdurchlaufs und zeigt damit eine Größenordnung für das Projekt mit den ca. 1,6 Millionen Zeilen Code auf. Dabei werden die Extraktion der Systemvariablen und die eigentliche Typanalyse mit Parsen und Generieren der Findings unterschieden. Zu der gesamten Dauer zählen neben den beiden genannten noch weitere Aufgaben, die bpsw. zum Einbinden des Repositories sowie zur Vor- und Nachbereitung der Analyse dienen. Sie gibt an, wie viel Zeit nach dem Starten der Analyse vergeht, bis die Ergebnisse einsehbar sind. Für die Historienanalyse ist die Anzahl und Größe der berücksichtigten Commits nicht bekannt. Gerade für die Extraktion der Systemvariablen ist aber zu erkennen, dass die Ergebnisse nach der Initialanalyse deutlich schneller vorliegen.

Forschungsfrage 3 kann nicht klar mit einem *Ja* beantwortet werden. Für den initialen Durchlauf benötigt die Analyse relativ viel Zeit. Es ist dabei zu berücksichtigen, dass im Normalbetrieb deutlich mehr Checks parallel auf dem Projekt ausgeführt werden. Sobald die erste Analyse abgeschlossen ist und bei weiteren Projektänderungen inkrementell gearbeitet wird, liegen die Ergebnisse wieder sehr zügig vor. Die Dauer der Typanalyse kann möglicherweise durch eine effizientere Implementierung verkürzt werden. Sie liegt jedoch in einer Größenordnung, die im Praxiseinsatz im Rahmen liegt. Dabei ist auch zu beachten, dass die Codebasis mit 1,6 Millionen Zeilen an Code sehr umfangreich ist und die Analyse tendenziell auf kleinere Projekte angewandt wird.

Des Weiteren stellt sich die Frage nach der Übertragbarkeit der Typanalyse. Es ist nicht bekannt, wie verbreitet die Verwendung von Systemvariablen im Bereich der SPS-Programmierung ist. Darüber hinaus können die Einheiten bzw. physikalischen Größen der Systemvariablen selbst definiert werden und sind nicht einheitlich. Die Vereinfachungsregeln müssten voraussichtlich bei dem Einsatz der Typanalyse in einem

anderen Unternehmen angepasst werden. Da die Analyse lediglich auf der Codebasis eines Unternehmens getestet wurde, kann eine direkte Übertragbarkeit nicht garantiert werden.

5.6 Fazit des Maschinenbauunternehmens

Die Ergebnisse der beiden Testdurchläufe wurden im Gespräch mit einem langjährigen Mitarbeiter des Maschinenbauunternehmens diskutiert. Er zeigt sich grundsätzlich sehr zufrieden mit der messgrößenbasierten Testanalyse und ihrer Umsetzung. Das Unternehmen setzt *Teamscale* bereits auf der beschriebenen Codebasis ein und würde die vorhandenen Checks gerne um die Typanalyse erweitern, um diese auf Dauer einzusetzen. Laut Mitarbeiter können die bestehenden und erkannten Inkonsistenzen der Einheiten damit behoben werden. Darüber hinaus können in Zukunft einige Fehlerquellen frühzeitig erkannt und die Codequalität allgemein verbessert werden. Damit lässt sich die vierte Forschungsfrage mit einem *Ja* beantworten. Die Ergebnisse aus der Analyse sind für Entwickler in der Praxis nützlich.

6 Zusammenfassung und weiterführende Arbeiten

6.1 Zusammenfassung

Die Typanalyse ist ein Mittel, um die Softwarequalität in der SPS-Programmierung mithilfe einer statischen Codeanalyse zu verbessern. Im Bereich des Maschinenbaus werden Variablen oft in einem physikalischen Kontext verwendet. Mit der Hilfe von Systemvariablen können Variablen mit den physikalischen Größen konnotiert werden. Auf dieser Basis wurde eine statische Analyse entwickelt, die sämtliche Operationen im Programmcode auf die konsistente Verwendung von Einheiten hin überprüft.

Dafür wurde im ersten Schritt ein Parser für die Systemvariablen entwickelt, um diese mitsamt den zugehörigen Einheiten aus den dafür vorgesehenen Dateien zu extrahieren. Im zweiten Schritt wurde auf der Basis von physikalischen Größen ein Kontext in der Form von Vereinfachungsregeln geschaffen, auf dessen Basis Regeln für die verschiedenen Operationen festgelegt werden konnten. Nachdem ein Parser für *Structured Text* Anweisungen entwickelt war, konnte auf den neu erstellten Syntaxbäumen die Typanalyse durchgeführt werden. Die Ergebnisse werden nun übersichtlich in der *Teamscale* Benutzeroberfläche dargestellt. Wird bspw. der Vergleich

$$a[Velocity] \geq b[Acceleration]$$

mit den gegebenen physikalischen Größen durchgeführt, wird der Entwickler mit folgender Nachricht darauf hingewiesen:

Inconsistent use of units in an operation: COMPARISON of a (unit: Velocity) and b (unit: Acceleration)

Um die Typanalyse auf ihre Funktionalität sowie die Effizienz hin zu überprüfen, wurde sie auf der Codebasis eines Maschinenbauunternehmens durchgeführt. Die Software für Spritzgussmaschinen umfasst ca. 30.000 Dateien, von denen etwa 9.900 für die Analyse relevant sind. Für die ca. 1,6 Millionen untersuchten Zeilen an Code wurden ca. zwei Stunden benötigt, bis die Ergebnisse vorlagen. Im ersten Testdurchlauf

wurden 356 Findings generiert, unter denen neben echten Fehlern auch False Positives zu finden waren. Nachdem ein paar Fehlerbehebungen und Verbesserungen an der Typanalyse vorgenommen waren, wurde ein zweiter Testdurchlauf auf dem gleichen System gestartet. Als Ergebnis lagen 114 Findings vor, die auf Regelverstöße in Bezug auf physikalische Größen hinweisen. Das Maschinenbauunternehmen sieht die Typanalyse und deren Ergebnisse als Erfolg und setzt sie gerne in Zukunft auf ihrem System ein.

Zusammenfassend lässt sich sagen, dass die messgrößenbasierte Typanalyse auf der Basis von Systemvariablen mit dem bestehenden Einheitensystem realisiert und erfolgreich auf echtem Projektcode getestet und evaluiert werden konnte. Sie stellt eine gute Möglichkeit dar, die Softwarequalität in der SPS-Programmierung kontinuierlich zu überprüfen und zu verbessern.

6.2 Weiterführende Arbeiten

Dennoch gibt es in diesem Zusammenhang einige offene Themen, die nicht vollständig umgesetzt werden konnten. In einigen Fällen stößt die statische Analyse an ihre Grenzen. An anderer Stelle ist der zeitliche Rahmen der Masterarbeit der begrenzende Faktor. In Kapitel 4.1.2 wurde bereits ein Lösungsansatz für das Problem von gegenseitigen Referenzierungen von Systemvariablen erwähnt. Auch die Grenzen der Analyse mit exakten Einheiten wurden aufgezeigt (vgl. Kapitel 4.2.5). Darüber hinaus sind drei weitere Aspekte erwähnenswert.

Rückgabeeinheiten von Funktionen Ein wichtiger Punkt ist die zukünftige Berücksichtigung von Funktionsaufrufen. Um eine vollständige Analyse durchführen zu können, müssen die Rückgabeeinheiten von Funktionen bekannt sein. Diese Einheiten können jedoch oftmals nicht mittels statischer Analyse bestimmt werden, da sie je nach Ausführung variieren können. Die Funktionen können in drei Kategorien eingeteilt werden:

1. Die Ergebniseinheit ist bei jedem Funktionsaufruf gleich.
2. Die Rückgabeeinheit hängt nur von den Eingangseinheiten ab.
3. Zusätzlich zu den Einheiten der Eingangsparameter hängt die Ergebniseinheit vom Zustand des Programmes ab.

Für Funktionen der ersten und zweiten Kategorie könnte eine Ergebniseinheit mithilfe der statischen Analyse bestimmt werden. Für Funktionen der dritten Kategorie kann

diese Einheit erst zur Laufzeit ermittelt werden. Darüber hinaus ist es nicht trivial, Funktionen automatisiert einer dieser Kategorien zuzuordnen und der manuelle Aufwand dafür wäre voraussichtlich zu groß. Möglicherweise kann die Ergebniseinheit für einige Sonderfälle bestimmt werden. Oftmals befindet sich der Name der Größe oder Einheit im Funktionsnamen selbst. Es ist keine festgelegte Regel, dennoch könnte dies in vielen Fällen nützliche Informationen liefern.

Konfigurierbarkeit von Regeln Dieser Punkt wird auf Anregung des Maschinenbauunternehmens aufgegriffen und diskutiert. Wird ein neues Projekt angelegt oder wird ein bestehendes größeren Änderungen unterzogen, können Anpassungen der Typanalyse erforderlich sein. Wird bspw. eine neue physikalische Größen eingeführt, muss diese im *Teamscale*-Backend angelegt werden. Auch kann eine individuelle Konfiguration für bestimmte Regeln gewünscht sein, da verschiedene Unternehmen möglicherweise unterschiedliche Programmierrichtlinien festlegen. Aktuell ist die Typanalyse nicht konfigurierbar und es ist ein Update von *Teamscale* nötig, um Anpassungen wirksam zu machen.

Sprachenerweiterung Systemvariablen können in jeder der fünf *IEC 61131-3* Sprachen verwendet werden. Im Projektcode des Maschinenbauunternehmens sind bspw. einige Funktionalitäten als *Sequential Function Chart* umgesetzt. Eine Erweiterung der Typanalyse auf alle Sprachen wäre denkbar. Der Aufwand würde darin bestehen, Parser für die verbliebenen Programmiersprachen zu erstellen. Die Logik der Typanalyse könnte übernommen werden. Diese Erweiterung geht über den Umfang der Arbeit hinaus, sie wäre aber mit Sicherheit ein guter Weg, um die Softwarequalität im Maschinenanlagenbau besser überwachen zu können.

Abbildungsverzeichnis

3.1 Struktur der Programm-Organisationseinheiten 10
3.2 *Structured Text* 11
3.3 Projektstruktur 12

4.1 Vereinfachungsprozess bei der Kombination von Einheiten 22
4.2 AST von `a := (b + c) * d >= e;` 32
4.3 AST von `arr[2, 5, 0] := 256;` 33
4.4 AST von `fb_pr(IN:=bIn1, T:=T#300ms);` 34
4.5 Folgerung der Einheiten in einem AST 36
4.6 Custom Check aktivieren 38
4.7 Neues Projekt anlegen 38
4.8 Finding Metrik 41

Tabellenverzeichnis

4.1 Auswahl an Basisdatentypen in IEC 61131-3 15
4.2 Operatorpräzedenz in *Structured Text* 28
4.3 Typen von *AstNode*s 30

5.1 Dauer der Initial- und Historienanalyse 50

Listings

4.1 Deklaration von Systemvariablen 14
4.2 Definition der Einheiten im Projekt 16
4.3 Ausschnitt des Enums mit den physikalischen Größen 17
4.4 Ausschnitt aus der *UnitDeducer* Klasse 20
4.5 Ausschnitt aus der Klasse *UnitDeducer* 25
4.6 Gültige Operationen mit Umrechnungen zwischen Einheiten 25
4.7 *CUP* Grammatik 27
4.8 Regeln für die Operatorpräzedenz 28
4.9 *CUP*-Grammatik mit Java Code 31
4.10 Mehrdimensionaler Arrayzugriff 32
4.11 Typanalyse.pu 39
4.12 Typanalyse.sv 41

5.1 False Positive im IF/ELSE-Zweig 48

Literatur

[13] *IEC 61131-3 International Standard*. International Electrotechnical Commission. Feb. 2013.

[20] *CUP User's Manual*. 4. Aug. 2020. URL: `http://www2.in.tum.de/projects/cup/docs.php`.

[HL09] S. Hangal und M. S. Lam. *Automatic Dimension Inference and Checking for Object-Oriented Programs*. 2009 IEEE 31st International Conference on Software Engineering. IEEE, Mai 2009, S. 155–165.

[HM95] I. J. Hayes und B. P. Mahony. „Using Units of Measurement in Formal Specifications". In: *Formal Aspects of Computing 7* (Mai 1995), S. 329–347.

[Hof13] D. W. Hoffmann. *Software-Qualität*. Springer Vieweg, 2013.

[JT09] K. H. John und M. Tiegelkamp. *SPS-Programmierung mit IEC 61131-3*. Springer-Verlag Berlin Heidelberg, 2009.

[Lic12] B. Lickly. „Static Model Analysis with Lattice-based Ontologies". Diss. https://escholarship.org/content/qt2hn7w4x1/qt2hn7w4x1.pdf: University of California, Berkeley, 2012.

[Prä+12] H. Prähofer, F. Angerer, R. Ramler, H. Lacheiner und F. Grillenberger. *Opportunities and Challenges of Static Code Analysis of IEC 61131-3 Programs*. In Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA). IEEE, Sep. 2012, S. 1–8.