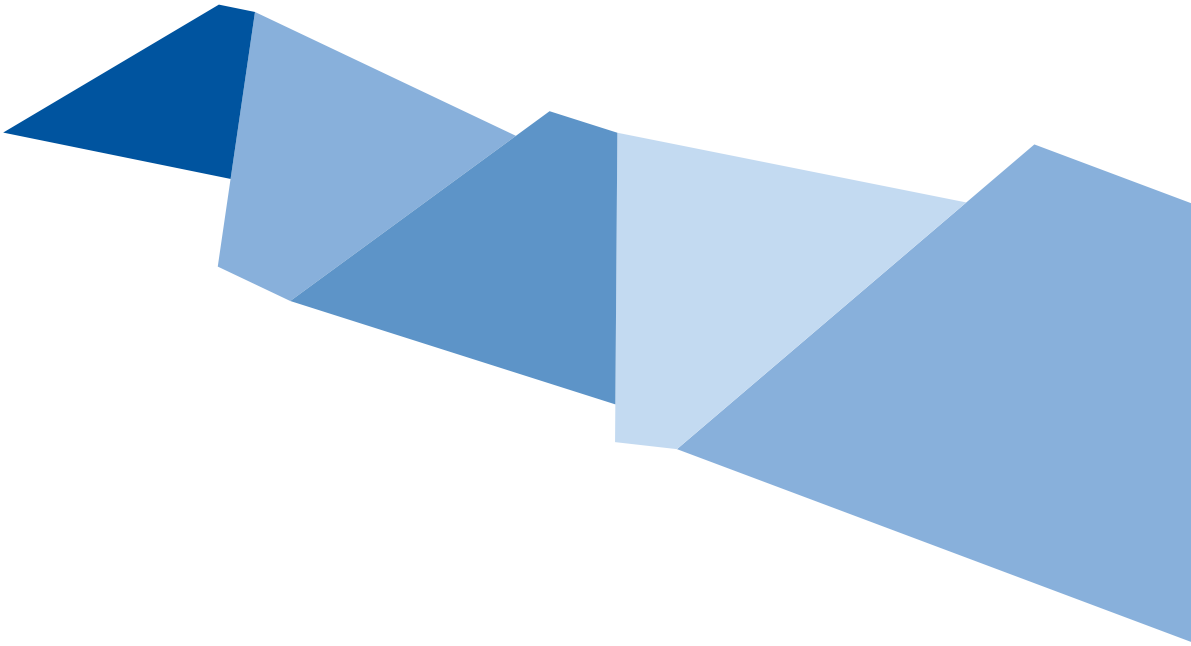


Unterstützung der SPS-Programmierung durch Statische Analyse während der Programmeingabe

Mathias Obster

Department of Computer Science

Technical Report

Aachener Informatik-Berichte (AIB) | ISSN 0935-3232 | AIB-2021-01
RWTH Aachen University | Department of Computer Science | Februar 2021

The publications of the Department of Computer Science of *RWTH Aachen University* are in general accessible through the World Wide Web.

`http://aib.informatik.rwth-aachen.de/`

Unterstützung der SPS-Programmierung durch Statische Analyse während der Programmeingabe

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften der
RWTH Aachen University zur Erlangung des akademischen Grades
eines Doktors der Ingenieurwissenschaften genehmigte Dissertation

vorgelegt von

Mathias Obster
(Master of Science)
aus Bonn

Berichter: Universitätsprofessor Dr.-Ing. Stefan Kowalewski
Universitätsprofessor Dr.-Ing. Georg Frey

Tag der mündlichen Prüfung: 10. November 2020

Mathias Obster
Lehrstuhl Informatik 11 – Embedded Software
obster@embedded.rwth-aachen.de

Gefördert durch die Deutsche Forschungsgemeinschaft (DFG) – GRK 1298 (AlgoSyn)

Aachener Informatik Bericht AIB-2021-01

Herausgeber: Fachgruppe Informatik
RWTH Aachen University
Ahornstr. 55
52074 Aachen
GERMANY

ISSN 0935-3232

Abstract

Using methods of static analysis, errors in program code can be detected fully automatically without executing it. This includes the technique of abstract interpretation and value set analysis in specific, which examines program behavior based on sets of possible variable assignments in order to find critical code areas.

Programs for programmable logic controllers (PLCs) can also benefit from being analyzed this way. Error detection and avoidance is of particular interest here, since PLCs control and monitor machines and systems in industrial safety-critical environments.

This dissertation is concerned with the question whether static analysis can contribute to error detection and avoidance during program input, especially during the activity of development of a PLC program. Therefore, the analysis framework ARCADE.PLC was extended in a way that it can annotate analysis results in an industry standard development environment. In addition to the constantly updated notes and warnings, this enhanced editor also allows the developer to display possible variable values which are calculated as an intermediate product in the value set analysis.

Besides the integration and visualization work, a newly introduced incremental approach can reduce the computational costs that otherwise result from the frequent execution of the analysis process. It is taking advantage of the fact that during short intervals of the development, small changes in the program often have only limited impact on the overall result. The described implementation has been checked using a number of test programs and scenarios for program changes. These were reflecting usual modifications as they can occur while writing source code.

Finally, a user study with participants from two industrial companies investigated whether developers can benefit from the results of static analysis while entering or modifying PLC source code.

Zusammenfassung

Durch Methoden der Statischen Analyse lassen sich automatisch Fehler in Programmcode finden, ohne diesen auszuführen. Darunter fällt besonders die Technik der abstrakten Interpretation bzw. der Wertemengenanalyse, welche Programmverhalten auf Basis von Mengen möglicher Variablenbelegungen untersucht, um kritische Codebereiche zu finden.

Auch in Programmen für Speicherprogrammierbare Steuerungen (SPSen) kann auf diese Weise nach Fehlern gesucht werden. Hier ist die Fehlervermeidung von besonderem Interesse, da SPSen im industriellen Umfeld zur Steuerung und Überwachung von Maschinen und Anlagen eingesetzt werden.

In dieser Dissertation wird untersucht, ob Statische Analyse bereits bei der Programmeingabe, also während der Entwicklung eines SPS-Programms, zur Fehlererkennung und -vermeidung beitragen kann. Dafür wurde das Analyseframework ARCADE.PLC erweitert, sodass es Analyseergebnisse in einer Entwicklungsumgebung darstellen kann, die auch in der Industrie zum Einsatz kommt. Neben den stets aktualisierten Warnungen können dem Programmierer durch diese Erweiterung zusätzlich mögliche Variablenwerte angezeigt werden, die als Zwischenprodukt in der Wertemengenanalyse berechnet werden.

Ein neu eingeführter inkrementeller Ansatz kann darüber hinaus den Berechnungsaufwand verringern, der sonst durch die häufige Ausführung der Analysen entsteht. Dabei wird ausgenutzt, dass sich während der Entwicklung in kurzen Zeitintervallen üblicherweise nur kleine Änderungen für das Gesamtprogramm ergeben. Die vorgestellte Implementierung wurde anhand mehrerer Testprogramme und Szenarien für Programmänderungen überprüft, wie sie beim Schreiben von Quelltexten auftreten können.

Schließlich wurde in einer Nutzerstudie mit Teilnehmern aus zwei Industrieunternehmen untersucht, ob Programmierer während der Eingabe und Bearbeitung eines SPS-Programms von Ergebnissen der Statischen Analyse profitieren können.

Inhaltsverzeichnis

Abbildungsverzeichnis

Glossar

ADA

Abstrakte Domänenanalyse (*engl.* Abstract Domain Analysis), eine Analyse der Semantik eines Programms auf Basis einer abstrakten Domäne z.B. Bitvektoren oder Wertebereichen.

AI

Abstrakte Interpretation (*engl.* Abstract Interpretation), die Simulation eines Programms auf abstrakten Domänen bzw. Informationsmengen. Beispiele sind die ADA und die LVA.

AS

Ablaufsprache (*engl.* Sequencial Function Chart, SFC), eine Programmiersprache der IEC 61131-3.

AST

Abstrakter Syntaxbaum (*engl.* abstract syntax tree), eine hierarchische Darstellung des syntaktischen Aufbaus eines Programms.

AWL

Anweisungsliste (*engl.* Instruction List), eine Programmiersprache der IEC 61131-3 ähnlich zu Assembler.

CFA

Kontrollflussautomat (*engl.* Control Flow Automaton), ein Automat, der den Kontrollfluss eines Programms abbildet. Auf seinen Kanten sind Instruktionen aufgetragen. Als annotierter CFG enthält er zusätzlich Analyseergebnisse in den Knoten.

CFG

Kontrollflussgraph (*engl.* Control Flow Graph), ein Graph, der den Kontrollfluss eines Programms abbildet. Seine Knoten entsprechen einer oder mehreren Instruktionen.

FAT

engl. Factory Acceptance Test. Abnahmetest im Werk oder in der Fabrik eines technischen Produkts beim Hersteller.

FBS

Funktionsbaustein-Sprache (*engl.* Function Block Diagram, FBD), eine grafische Programmiersprache der IEC 61131-3.

HMI

engl. Human Machine Interface. Generell Einrichtungen zum Überwachen und Bedienen von Prozessen in Maschinen. Im engeren Sinne üblicherweise grafische Bedienoberflächen für stationäre oder mobile Ein-/Ausgabegeräte.

IDE

Integrierte Entwicklungsumgebung (*engl.* Integrated Development Environment), Sammlung an Werkzeugen zur Entwicklung von Software.

IR

Zwischencode (*engl.* Intermediate Representation), eine interne Darstellung von Programmcode, die üblicherweise nur noch aus wenigen Instruktionen besteht, damit aber die komplette Semantik mehrerer Eingabesprachen abdeckt.

JIT

engl. Just-in-Time-Compiler. Im In Java Laufzeitumgebungen wird Programmcode zur Laufzeit in nativen Maschinencode übersetzt, um ihn ein nächstes Mal schneller ausführen zu können.

JSON

engl. Java Script Object Notation, eine Notation, um Objekte aus objektorientierter Sprache als Unicode-String darzustellen.

KOP

Kontaktplan (*engl.* Ladder Logic, LL), eine grafische Programmiersprache der IEC 61131-3.

LCS

engl. longest common subsequence (problem). Eine Problemstellung der Informatik, in der eine möglichst lange gemeinsame Teilfolge zwischen zwei Listen gefunden werden soll, wobei Elemente der Listen ausgelassen werden können, ihre Reihenfolge aber erhalten bleiben muss.

LVA

engl. Live Variable Analyse, eine Analyse, die zu einer Variablenzuweisung bestimmt, ob diese an einem späteren Zeitpunkt noch gelesen wird, bevor sie wieder überschrieben wird.

MoM

engl. Manufacturing Operations Management. Systeme zur Abwicklung und Optimierung von Ende-zu-Ende Produktionsprozessen.

POE

Programmorganisationseinheit, ein nach IEC 61131-3 definierter Baustein eines SPS-Programms bestehend aus Variablen oder Schnittstellenbeschreibung und Programmlogik.

RDA

engl. Reaching Definition Analysis, eine Analyse zur Berechnung der Sichtbarkeit einer Variablenzuweisung an einem bestimmten Programmpunkt.

SDK

engl. Software Development Kit, eine Sammlung von Werkzeugen und Bibliotheken zur Entwicklung oder Erweiterung von Software.

ST

Strukturierter Text (*engl.* Structured Text), eine Programmiersprache der IEC 61131-3 ähnlich zu Pascal.

VSA

Wertemengenanalyse (*engl.* Value-Set Analysis), eine Analyse, bei der die Ausführung des Programms mit Wertemengen simuliert wird, um eine Überapproximation aller zur Laufzeit möglichen Werte zu erhalten.

Kapitel 1

Einleitung

In der industriellen Automatisierung bestehen hohe Anforderungen an die Betriebssicherheit und die Verfügbarkeit eingesetzter Systeme. Speicherprogrammierbare Steuerungen (SPSen) werden hier genutzt, um z.B. Fertigungsstraßen, Montageroboter, Kraftwerke und chemische Prozessanlagen zu steuern und zu überwachen. Steuerungsprogramme, die auf SPSen ausgeführt werden, sind damit eine kritische Komponente, da sie die korrekte Verarbeitung aller Eingangs- und Ausgangssignale übernehmen. Fehler in der Software können folglich zur Gefahr für Menschen und Umwelt werden sowie über Produktionsausfälle auch zu finanziellen Schäden führen. Maßnahmen zur Risikovermeidung in einem Automatisierungsprojekt müssen daher auch Vorkehrungen zur Fehlervermeidung oder Fehlertoleranz der Software beinhalten.

Neben dem Bewusstsein für die Sicherheitsrelevanz eingesetzter Software stieg in den letzten Jahren auch die Komplexität der Steuerungssoftware [54] – unter anderem durch den Einsatz optimierender Steuerungsalgorithmen und die Vernetzung der Anlagen im Kontext von Industrie 4.0 [53]. Die Interessengemeinschaft Automatisierungstechnik der Prozessindustrie NAMUR[1] schätzt Steuerungssoftware daher im Vergleich zu anderen Bestandteilen der Prozessautomatisierung als fehleranfällig und kritisch hinsichtlich der Auswirkung solcher Fehler ein [36].

Für die Entwicklung von Steuerungssoftware werden mittlerweile oft modellbasierte Verfahren und Bibliotheken verwendet, um den Entwicklungsaufwand zu reduzieren. Durch die jeweils individuellen Anforderungen an eine Anlagensteuerung muss jedoch weiterhin Quelltext auf konventionelle Weise entwickelt werden. Der Testaufwand wächst damit kontinuierlich und es werden auch hier üblicherweise modellbasierte Ansätze und Simulationen genutzt [25, 38, 41]. Über das reine Testen der Software hinaus ist teilweise die Nutzung formaler Methoden für die Sicherstellung des korrekten Betriebs vorgeschrieben oder zum Beispiel im Rahmen sogenannter Safety Integrity Level (SIL) mindestens empfohlen. Die SIL-Klassifikation nach der IEC 61508 bzw. der IEC 61511 [23, 24] beschreibt vier Stufen, die abhängig von den Anforderungen der Anwendung Sicherheitsfunktionen und -maßnahmen definieren, um Menschen, Güter und Umwelt zu schützen. Formale Methoden basieren auf einem mathematischen Kalkül und können durch Werkzeuge wie der Modellbildung generelle Eigenschaften eines Programms verifizieren oder Fehler in

[1] Interessengemeinschaft Automatisierungstechnik der Prozessindustrie e.V.: `https://www.namur.net`

einer Implementierung finden. Doch obwohl formale Methoden im klassischen Software Engineering bereits eingesetzt werden, um wachsende Komplexität zu beherrschen, den Testaufwand zu reduzieren und die Softwarequalität zu verbessern, erfahren diese Methoden im Umfeld der Automatisierungstechnik bisher keine breite Anwendung [39].

1.1 Motivation

Statische Analyse ist eine formale Methode, Quellcode automatisch auf Fehler und Fehlerquellen zu überprüfen. Sie arbeitet dabei ohne Spezifikation eines erwarteten Verhaltens oder eines Anlagenmodells, sondern weist auf Schwachstellen in einer Implementierung hin, bevor das Steuerungsprogramm auf einer Zielplattform ausgeführt wird.

Es existieren bereits mehrere Werkzeuge aus wissenschaftlichen Projekten oder Untersuchungen, die mittels Statischer Analyse Implementierungsfehler in IEC 61131-3-Programmen finden können [7, 35, 40]. Diese Werkzeuge analysieren einen Teil oder auch ganze Programme, meist als separate Werkzeuge oder als spezialisierte Lösung im Rahmen einer Machbarkeitsstudie. Sofern Analyseverfahren in Entwicklungsumgebungen bisher integriert sind, beschränken sich diese meist nur auf syntaktische Prüfungen und nutzen nicht das Potenzial verfügbarer Methoden der Statischen Analyse. Auf Entwicklerseite besteht dagegen das Interesse, solche Verfahren in Softwaresystemen zur Steuerungsentwicklung zukünftig zu nutzen, wie Schmidt und Lüder et al. unter anderem in einer Umfrage 2014 bei deutschsprachigen Entwicklern herausfanden [29, 45].

Bisherige Untersuchungen beziehen sich auf die Methoden und Werkzeuge zur Statischen Analyse, mit denen Fehler in Implementierungen gesucht wurden. Anschließend konnten Korrekturen durch den Programmierer vorgenommen werden. Stehen die Informationen jedoch früher, also schon während der Entwicklung zur Verfügung, können Fehler früher identifiziert und direkt behoben werden, wie Saff et al. [43] beschrieben. Sie ließen in ihren Untersuchungen Testfälle für Programme im Hintergrund der Entwicklungstätigkeit prüfen und gaben direktes Feedback an den Programmierer. So konnte der Wechsel zwischen Programmier- und Testphase optimiert und die Entwicklungszeit verkürzt werden. Diese direkte Korrektur im Editor ohne einen Moduswechsel zu einer separaten Überarbeitungsumgebung wird von Entwicklern üblicherweise bevorzugt [33] und für bessere Codequalität auch empfohlen [19]. Einen Schritt weiter in diese Richtung geht der Live-Coding-Ansatz von Krämer et al. [27]. Dieser Ansatz verfolgt die Annotation von Quellcode während der Entwicklungsphase mit Ausführungsergebnissen wodurch ein positiver Einfluss auf die Zeit zum Korrigieren von Fehlern erzielt werden konnte.

Die vorliegende Arbeit nimmt daher den Programmierer in den Fokus und untersucht, ob ihm Ergebnisse und Warnungen als Ergebnis der Analyse von Steuerungscode bereits während der Programmierung einen Vorteil bringen. Abgeleitet von den Möglichkeiten der Statischen Analyse kann dieser Vorteil in kurzer Einlesezeit in existierenden Code liegen, in schnellerer Wahrnehmung von typischen Programmfehlern während der Programmierung oder insgesamt schnellerem Lösen von Entwicklungsaufgaben. Daraus abgeleitet wird die folgende Hypothese untersucht:

Hypothese 1 (H1): *Die Bereitstellung von Informationen gewonnen durch Statische Analyse des bereits geschriebenen Programms bietet dem Entwickler eine Unterstützung bei der Entwicklung des SPS-Programms.*

Für diese Hypothese wird davon ausgegangen, dass ein Programm bereits zu einem Teil geschrieben wurde und dieser Teil mithilfe von Statischer Analyse untersucht wird. Diese Momentaufnahme des Programms wird in dieser Arbeit als *Iteration* bezeichnet. Die Ergebnisse der Analyse weisen dann auf Auffälligkeiten in der Iteration hin. Sie können zur Verbesserung oder Korrektur bis zur nächsten Iteration genutzt werden und bieten damit die Möglichkeit, Fehler schon deutlich früher im Entwicklungsprozess erkennen und beheben zu können. Es wird davon ausgegangen, dass die Bereitstellung der so gewonnenen Informationen für den SPS-Programmierer hilfreich sind. Für die Evaluation wird die aufgestellte Hypothese weiter konkretisiert, um anhand implementierter Unterstützungsfunktionen den Nutzen für einen SPS-Programmierer auch messen zu können.

1.2 Aufbau und bibliografische Hinweise

In dieser Arbeit werden in Kapitel 2 zunächst Hintergründe und verwandte Arbeiten vorgestellt. Dabei wird kurz die Anwendungsdomäne der Automatisierungstechnik und der Speicherprogrammierbaren Steuerungen skizziert, bevor Eingabeunterstützungssysteme, speziell das Live Coding und die Statische Analyse, vorgestellt werden.

In Kapitel 3 werden dann die Anforderungen an den Prototyp aufgestellt, der im Rahmen der Arbeit entwickelt und in Kapitel 4 beschrieben wird. Nachfolgend befasst sich Kapitel 5 mit einer Nutzerstudie, die mithilfe des Prototypen durchgeführt wurde. Die Arbeit schließt mit einer Zusammenfassung und einem Ausblick in Kapitel 6.

Das in Kapitel 4 beschriebene Verfahren zur inkrementellen Analyse als Vorbereitung für die Unterstützung des Programmierers während der Eingabe wurde vom Autor dieser Arbeit bereits vorab ausgearbeitet und in [37] veröffentlicht.

Im Rahmen dieser Dissertation wurden thematisch verwandte Abschlussarbeiten betreut, deren Resultate teilweise in die folgende Arbeit eingeflossen sind. In den entsprechenden Kapiteln wird eine solche Verwendung explizit erwähnt und durch Literaturangaben mit dem Nachnamen des Studenten und dem Jahr der Abgabe gekennzeichnet.

Kapitel 2

Stand der Technik

Der Anwendungsbezug der vorliegenden Arbeit ist die Programmierung von speicherprogrammierbaren Steuerungen (SPSen). Diese werden dafür in diesem Grundlagenkapitel kurz eingeführt und die Entwicklung von Steuerungssoftware im Kontext der Automatisierungstechnik betrachtet. Anschließend wird der Fokus auf existierende SPS-Entwicklungsumgebungen gelegt und aktuelle Techniken vorgestellt, die einen Entwickler bei der Programmierung unterstützen können. Zuletzt wird kurz die Technik der Statischen Analyse und das Framework ARCADE.PLC vorgestellt.

2.1 Speicherprogrammierbare Steuerungen

Speicherprogrammiebare Steuerungen sind universell einsetzbare Steuerungscomputer, mit denen im industriellen Kontext Prozesse gesteuert und Abläufe koordiniert werden. Ihre Einsatzgebiete reichen von einfachen Heizungssteuerungen über Vergnügungs-Fahrgeschäfte bis hin zu komplexen Fertigungsstraßen und großindustriellen chemischen Prozessanlagen. Entsprechend der Anforderungen variiert die meist modular erweiterbare Hardware einer SPS von eigenständigen, kleinen Baugruppen für den platzsparenden Einbau in einem Schaltschrank bis hin zu leistungsfähigen Industrie-PCs, auf denen Software-SPSen ausgeführt werden und die über Bus- und Netzwerksysteme mit entfernten I/O-Modulen kommunizieren. Eine zentrale Anforderung an SPSen ist immer die Zuverlässigkeit der Steuerung, weshalb die Hardware dieser Systeme meist redundant ausgelegt ist.

2.1.1 Programmierkonzepte

Die Logik einer SPS inklusive benötigter Regelungen sind durch Software definierbar, die heute üblicherweise in einer der Sprachen geschrieben ist, die in der IEC 61131-3 [16] vorgegebenen werden. Die Norm definiert mit Anweisungsliste (AWL) und Strukturiertem Text (ST) zwei textuelle und mit Kontaktplan (KOP) und der Funktionsbaustein-Sprache (FBS) zwei grafische Programmiersprachen. Daneben definiert sie mit der Ablaufsprache (AS) noch eine Sprache zur Strukturierung paralleler und sequenzieller Abläufe ähnliche zu Petrinetzen.

Alle Sprachen haben das Konzept der Programm-Organisationseinheiten (POEs) gemeinsam, welches semantische Einheiten eines Programms zur Strukturierung zusammenfasst. Sie bestehen jeweils aus einem Deklarations-Abschnitt für Variablen und einem Rumpf für das Programm in der jeweiligen Sprache, das auf diesen Variablen operiert. Variablen können dabei als Schnittstelle für andere POEs dienen, oder auf globale Ressourcen zugreifen.

Vier Arten von POEs sind definiert: *Funktionen*, *Funktionsbausteine*, *Programme* und *Klassen*. Funktionen sind die einfachste Einheit. Sie können *Eingangs-* und *Ausgangsvariablen*, sowie lokale Variablen verwenden. Als einzige POE können Sie einen Rückgabewert definieren, wodurch sie Funktionen aus Sprachen wie C sehr ähnlich sind. Einen Zustand können Funktionen nicht speichern, da keine Variablen erlaubt sind, die über den Aufruf einer Funktion hinweg ihren Wert erhalten. Entsprechend ist es auch weder nötig noch möglich, Funktionen zu instanziieren. Funktionsblöcke können dagegen beliebig oft instanziiert werden und dabei jeweils eigene Zustandsvariablen nutzen. Sie können von anderen Funktionsblöcken oder von Programmen instanziiert und aufgerufen werden. Programme können über den Funktionsumfang der Funktionsblöcke hinaus direkt auf Hardware-Adressen zugreifen und können globale Variablen deklarieren. Schließlich führt die neueste Version der Norm [16] auch Klassen als Erweiterung der Funktionsblöcke ein, die mit Interfaces, Methoden und Vererbung an Konzepte objektorientierter Sprachen anknüpfen.

Welche Sprache im Programm-Abschnitt der POE zum Einsatz kommt, kann der Entwickler seinen Fähigkeiten und dem Anwendungsfall anpassen. Auch Kombinationen von POEs verschiedener Sprachen sind möglich. Die Sprache ST eignet sich unter den genannten Sprachen dabei gut für kompakte Darstellungen komplexerer Algorithmen. Es handelt sich um eine Hochsprache mit einer Pascal-ähnlichen Syntax.

Charakteristisch für SPSen ist eine zyklische Ausführung der definierten Programme [51]. Hierfür gliedert sich die Ausführung in drei Phasen:

- Eingänge lesen
- Programm ausführen
- Ausgänge schreiben

Zu Beginn werden die Eingänge der SPS gelesen und für die Dauer der Programmausführung in einem Speicherabbild vorgehalten. Ein konfiguriertes Programm oder ein Funktionsblock wird danach mit diesen Werten aufgerufen, die sich währen der Ausführung nicht mehr ändern. Dieses Programm berechnet dann Werte für die Ausgänge, die erst in der nächsten Phase nach Abschluss des Programms an den Ausgängen der SPS geschrieben bzw. angelegt werden. Wann dieser Zyklus mit diesen drei Phasen erneut ausgeführt wird, lässt sich konfigurieren. Im häufig verwendeten periodischen Modus wird ein Zyklus in einem definierten Intervall gestartet. Beim Interrupt-Modus ist die Ausführung dagegen an ein externes Ereignis gebunden und im kontinuierlichen Ausführungsmodus schließt sich an das Schreiben der Ausgänge direkt wieder der nächste Zyklus an.

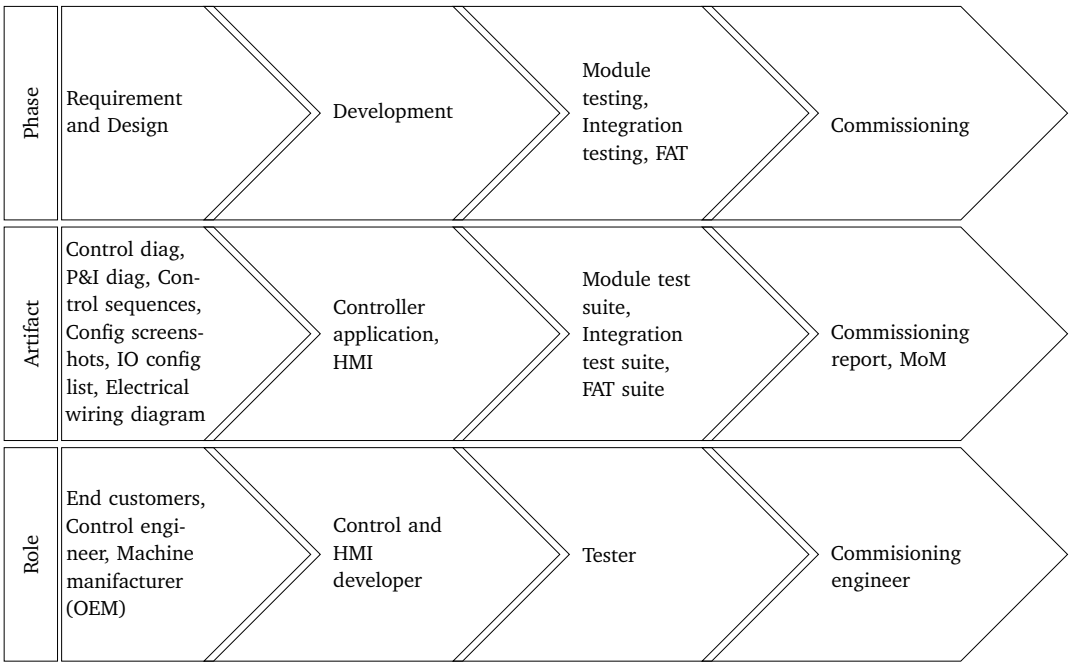


Abbildung 2.1: Vorgehensmodell zur Planung und Umsetzung einer Anlagensteuerung in der Automatisierungstechnik (Abbildung aus [18], © 2011 IEEE)

2.1.2 Vorgehensmodell in der Automatisierungstechnik

Um die Anforderungen an die Programmierung einer SPS nachvollziehen zu können, ist ein Blick auf den umgebenden Prozess in der Automatisierungstechnik sinnvoll. Die Entwicklung einer Automatisierungslösung besteht aus mehreren Phasen. Exemplarisch sind sie zusammen mit dabei anfallenden Artefakten und den ausführenden Personen bzw. Rollen in Abbildung 2.1 dargestellt. In der Anforderungs- und Design-Phase sind viele Schritte zusammengefasst, die den gesamten physikalischen und elektrotechnischen Aufbau ermöglichen. Darauf baut die Entwicklungsphase auf, in der die in dieser Arbeit betrachtete Programmierung für die vorgesehenen Steuerungen erfolgt. Anschließend folgen üblicherweise die Schritte zum Testen der Teil- und Gesamtsysteme sowie die Inbetriebnahme. In der Abbildung nicht enthalten ist der anschließende Übergang der Analge in die Phase von Betrieb, Wartung und Instandhaltung.

Das abgebildete Vorgehensmodell wird in der Praxis oft verwendet, die Einhaltung der zeitlichen Reihenfolge dieser Phasen ist jedoch nicht strikt [18]. So ist es durchaus gängige Praxis, auch in der Testphase oder nach der Inbetriebnahme Änderungen an früheren Schritten vorzunehmen, zu denen bevorzugt auch die Änderung und Wartung der Software zählen.

In der Forschungs- und Anwendungsdomäne der Automatisierungstechnik werden die in den beschriebenen Phasen anfallenden Tätigkeiten im deutschen Sprachgebrauch oft als *Engineering* (aus dem Englischen *to engineer*: *etw. entwickeln*) oder als *Application Engineering* (Deutsch: *Anwendungsentwicklung*) bezeichnet. In dieser Arbeit liegt der Fokus auf der zweiten Phase, der Entwicklungsphase, weshalb hier weiter die Begriffe Entwickler,

SPS-Programmierer und (SPS-)Programm verwendet werden. Die Tätigkeit des Entwickelns schließt dabei das Programmieren und Testen mit ein.

Die Anforderungen der Phasen in Abbildung 2.1 leiten sich aus den Anforderungen an das Gesamtsystem und damit aus der zu konstruierenden oder zu überarbeitenden Anlage ab. Da die zu kontrollierenden Prozesse der Anlage meist sicherheitskritisch sind, gehört es zu den grundlegenden Zielen der Automatisierungstechnik, die Zuverlässigkeit und Sicherheit des Betriebsablaufes zu gewährleisten. Dazu gehört insbesondere auch die Korrektheit des auf der Steuerung laufenden Programms, welches dafür einem intensiven Testprozess unterzogen werden muss [38, 41]. Die Zeit um Tests durchzuführen ist jedoch in der Regel knapp bemessen und beträgt laut einer Umfrage durch Rösch et al. [42] etwa 25 % der Zeit, die insgesamt für Softwareentwicklung im Projekt vorgesehen wird. In dieser Arbeit werden daher Möglichkeiten untersucht, Testaktivitäten bereits in den Entwicklungsprozess einzubinden.

2.1.3 Aktuelle Entwicklungsumgebungen für SPSen

Anbieter von SPSen wie Siemens oder ABB liefern ihre Geräte üblicherweise zusammen mit einer Software zur Programmierung aus oder bieten eine Entwicklungsumgebung (IDE) passend zu ihren Geräten zur Lizenzierung an. Diese Programme beinhalten üblicherweise einen ähnlichen Funktionsumfang wie eine IDE für Desktop- oder mobile Applikationen: Projektverwaltung, Code-Editoren für unterstützte Sprachen, Konfigurationsmöglichkeiten, Compiler und Debugger. Darüber hinaus enthalten sie oft weitere Programmkomponenten zur Netzwerk-Verwaltung oder Visualisierungs-Editoren für spezifische Funktionen der herstellereigenen Automatisierungs-Produktpalette.

Ein Wechsel der Entwicklungsumgebung ist jedoch im Vergleich zu normalen Desktop-IDEs nicht ohne Weiteres möglich, da herstellerspezifische Funktionen nicht unterstützt und der benötigte Binärcode für die Ziel-SPS oft proprietär ist. Diese Abhängigkeit vom Hersteller verhindert die Nutzung von Programmen von Drittherstellern mit einem möglicherweise größeren Funktionsumfang. Über Plugins sind IDEs wie das TIA-Portal von Siemens und Codesys von 3S zwar erweiterbar, allerdings sind auch diese Plugins wieder herstellerabhängig zu entwickeln, was zu einem absolut gesehen kleinen Markt führt. Anwender aus dem Bereich der Automatisierungstechnik sehen daher Werkzeugintegration als weiterhin offene Anforderung an diese IDEs [45].

Speziell hinsichtlich der Programmentwicklung lassen sich SPS-Editoren heute nicht mit den Fähigkeiten moderner IDEs messen. Die Komfortfunktionen gehen zwar über die eines primitiven Texteditors hinaus und bieten mindestens alle eine Syntaxhervorhebung, zeigen also Schlüsselwörter in anderen Schriftstilen oder -Farben. Allerdings bieten nicht alle Editoren darüber hinaus noch Hilfe für den Programmierer, um Programmfehler schon vor einer Testphase zu erkennen.

In Tabelle 2.1 ist eine im Rahmen dieser Arbeit entstandene Übersicht über die Funktionen einiger verbreiteter IDEs für SPSen abgebildet. Getestet wurde jeweils, ob Programmcode automatisch vervollständigt werden kann (*Autocomplete*), ob Syntaxfehler direkt im Editor angezeigt werden und sie erst beim Compilieren des Programms auffallen (*Syntax*), und ob Bezeichner von Variablen markiert werden, die im aktuellen Kontext unbekannt sind

Produkt	Autocomplete	Syntax	Def.	Mult.Assign	Div0
ABB Automationbuilder V1.1	✗	✓	✓	✗	✗
B&R Automation Studio 3	✓	✗	✗	✗	✗
Codesys 2.3	✗	✗	✗	✗	✗
Codesys 3.5	✓	✓	✓	✗	✗
LogiCAD 3	✓	✓	✓	✗	✗
Beckhoff TwinCat 3.1	✓	✗	✓	✗	✗

Tabelle 2.1: Feature-Vergleich der Programmcode-Editoren in verbreiteten Entwicklungsumgebungen

(*Def.*). Diese beiden Funktionen sind zumindest in aktuelleren Versionen der IDEs in vielen Fällen enthalten.

Weiter wurden zwei Fälle untersucht, bei denen die Semantik des Programms ausgewertet werden müsste. *Mult.Assign* steht für den Test, ob mehrfache Zuweisung an Ausgangsvariablen markiert werden. Diese doppelten Zuweisungen gehören zu den Code-Smells, es handelt sich also nur um Auffälligkeiten im Programm, die Fehler produzieren können und besser vermieden werden sollten. Zuletzt wurde eine Division durch Null (*Div0*) getestet. Diese Fehler können zu falschen Berechnungsergebnissen führen, sofern die SPS den Laufzeitfehler abfängt. Im schlimmsten Fall kann der Fehler dazu führen, dass sich die Steuerung neustartet oder in einen Fehlerzustand wechselt. Um zu entscheiden, ob an einer Stelle durch Null geteilt wird, muss jedoch die Semantik des Programms interpretiert oder das Programm anhand konkreter Werte getestet werden. Diese beiden letzten Funktionen werden von keinem der untersuchten Programme unterstützt. Für Codesys 3.5 ist zusätzlich zur vertriebenen Version noch ein Plugin für Statische Analyse enthalten, das diese Tests teilweise unterstützt. Eine Erkennung der Division durch Null ist jedoch auch hier nicht möglich, wenn der Divisor wie in diesem Test keine Konstante sondern eine Variable mit Wert 0 ist.

Die Untersuchung zeigt, dass die in dieser Arbeit adressierte Art von Unterstützungsfunktionen für SPS-IDEs bisher nicht umgesetzt sind. Einfache semantische Prüfverfahren sind in Entwicklungsumgebungen für Desktopanwendungen wie Visual Studio und Eclipse dagegen mittlerweile sehr verbreitet. Der Nutzen solcher Unterstützungssysteme wurde in der Literatur wissenschaftlich untersucht, worauf im Folgenden eingegangen werden soll.

2.2 Eingabeunterstützungssysteme

Benutzeroberflächen für Quelltexteditoren sind ein wichtiger Bestandteil der Entwicklungsumgebung. Ziel der Forschung in dem Bereich ist es, den üblichen Moduswechsel zwischen entwicklungs- und werkzeug-unterstützter Überarbeitungs-Phase zu verringern [31, 32, 33]. Werden Überarbeitungen direkt im Editor ohne Moduswechsel vorgenommen, kann dies die Codequalität verbessern [19], da die nachfolgende Bearbeitung des Quelltextes direkt darauf aufbauen kann.

Liveness-Level	Beschreibung
1	Nicht ausführbare Programmbeschreibung, Flussdiagramme
2	Bei Bedarf ausführbare Modelle, ausführbarer Quelltext
3	Automatische Neuberechnung oder Ausführung bei Änderungen
4	Programm wird kontinuierlich auf änderbarem Code ausgeführt

Tabelle 2.2: Liveness-Level 1 bis 4 nach Tanimoto [49, 50]. Jedes Level ist eine Erweiterung des nächstkleineren Levels.

Ein Moduswechsel lässt sich durch die Anzeige zusätzlicher Informationen über Sprachkonstrukte verhindern, indem zum Beispiel Variablentypen zu einem Bezeichner angezeigt werden. Es lassen sich auch weitergehende Analysen durchführen und Modelle generieren, mit deren Hilfe das Programmverhalten für den Entwickler visualisiert werden kann. Programmverhalten lässt sich in Form von Kontrollflussgraphen, Warnungen zu möglicherweise fehlerhaftem Verhalten oder konkreten Wertinformationen darstellen.

Van de Vanter entwickelte bereits 1992 [52] ein Unterstützungssystem, welches den eingegebenen Quellcode kontinuierlich in ein Modell überführte und ohne zusätzliche Benutzerinteraktion Hilfestellungen bei Navigation und Strukturierung des Quellcodes anbietet. Die Annotation von Übersetzungsfehlern, hervorgerufen durch syntaktische Fehler im Programmcode, waren auch für ihn damals nur ein Anfang dieser möglichen Hilfestellungen.

2.2.1 Liveness

Einen Schlüsselbegriff für interaktive Programmiersysteme prägte Tanimoto [49, 50]. Mit seinem Begriff *liveness* teilt er den Grad der Interaktivität ein, also wie der Computer auf die Eingaben des Benutzers reagiert. Das erste dieser Liveness-Level ist definiert als nicht ausführbare Darstellung eines Codes in Form von Text oder UML. Level 2 entspricht dem bekannten Schema, Quellcode zu entwickeln und anschließend auszuführen oder zu debuggen. Level 3 reagiert auf die Eingaben des Programmierers und führt Programme nach Änderungen des Benutzers automatisch aus. Das Programm kontinuierlich auszuführen und während der Laufzeit durch Änderungen des Programmierers unmittelbar zu beeinflussen wird als Level 4 bezeichnet. Eine Übersicht dieser vier Level ist in Tabelle 2.2 dargestellt. In der späteren Veröffentlichung [50] erweiterte Tanimoto die Skala noch um zwei weitere Stufen bis hin zur Vorhersage des erwarteten Verhaltens bei möglichen Fortsetzungen des Programms.

Nach dieser Definition erfüllen heute alle bekannten IDEs die Liveness-Level-2-Eigenschaft. Level 3 ist dagegen schwieriger zu erreichen, da der Programmcode dafür kontinuierlich aus den Benutzereingaben interpretiert oder ausgeführt werden muss. Tabellenkalkulationsprogramme erfüllen diese Eigenschaft, indem sie Formeln automatisch neu berechnen, wenn sich Werte in referenzierten Zellen ändern.

Einige IDEs übersetzen den aktuell geschriebenen Programmcode fortlaufend in die Zielsprache bzw. Binärcode und können auf diese Weise Syntaxfehler bei der Eingabe

annotieren. Dieses Prinzip ist unter dem Begriff *Continuous Compilation* bekannt und erlaubt eine schnellere Ausführung, da der Compiliervorgang nicht erst auf Anforderung gestartet werden muss. Auch manche SPS-IDEs wie Codesys 3.5 bieten diese Funktion. Ein Liveness-Level 3 ist damit jedoch noch nicht erfüllt, da hier der Programmcode semantisch nicht untersucht bzw. ausgeführt wird. Auch beschreibt ein Syntaxfehler nicht das Verhalten des Programms.

Einen Schritt weiter gehen Saff et al. [44] in einem IDE-Prototyp für Java-Quellcode. Während der Entwicklung wird der Programmcode mittels Continuous Compilation übersetzt und vordefinierte Testfälle werden auf dem Programmcode ausgeführt. Das Ergebnis der Tests wird dem Programmierer angezeigt. Eine angeschlossene Benutzerstudie ergab bei Saff et al. einen positiven Effekt auf die Dauer der Aufgabenbearbeitung. Diese automatische Verarbeitung und Ausführung des Programmcodes kann der Definition Tanimotos nach bereits als Liveness Level 3 gewertet werden, auch wenn die Informationen über Variablenwerte oder Meldungen über Verhalten des Programmcodes nicht detailliert an den Programmierer zurückgemeldet werden. Entscheidend ist hier, dass die Ausführung keine explizite Handlung des Programmierers erfordert.

2.2.2 Live Coding

Den Begriff *Live Coding* verwendete Jan Peter Krämer in seiner Dissertation [26] für Code-Editoren mit einem Liveness-Level von 3. Mit dem Metis-Projekt [27] wurde dafür ein Prototyp entwickelt, der in einer IDE Unterstützungsfunktionen für die Entwicklung von JavaScript-Programmen bereitstellt. Metis bietet Laufzeitinformationen direkt im Editor an, die aus dem gerade entwickelten Programm generiert werden. Der Editor sendet dafür das aktuell entwickelte Programm bei jeder Änderung an ein Back-End, welches den Programmcode ausführt und die Ergebnisse zurück sendet. Angezeigt werden neben einzelnen konkreten Werten in jeder Zeile auch Wahrheitswerte in Bedingungen.

Metis wurde verwendet, um in einer Benutzerstudie zu untersuchen, ob Nutzer von dem Live-Coding-Editor profitierten und ob sie ein anderes Entwicklungsverhalten zeigten. Unter Berücksichtigung unterschiedlicher Erfahrungen der Programmierer konnte in der Benutzerstudie [26] nachgewiesen werden, dass die Bearbeitung der Aufgaben mit den Laufzeitinformationen schneller möglich ist als in der Kontrollgruppe. Weiter wurde sichtbar [27], dass die Probanden der Testumgebung weniger Zeit in einer expliziten Debug-Umgebung verbringen sondern stattdessen die im Editor angezeigten Laufzeitinformationen nutzen konnten, um ihren Programmcode zu testen.

2.3 Statische Analyse durch ARCADE.PLC

Mit der Statischen Analyse wird nun eine formale Methode vorgestellt, mit der es möglich ist, Programmcode vollautomatisch zu untersuchen. Im Allgemeinen werden Verfahren als Statische Analyse bezeichnet, die auf dem Quellcode eines Programms arbeiten und nicht eine Ausführung auf tatsächlicher oder simulierter Hardware voraussetzen. Im Gegensatz zu anderen Verfahren wie dem Model-Checking sind keine zusätzlichen Eingaben in Form

von Spezifikationen neben dem Programmcode nötig, um Aussagen darüber zu treffen. So können Fehler wie unerreichbare Programmteile, Divisionen durch Null oder unerlaubte Array-Zugriffe automatisch erkannt werden. Möglich wird dies durch eine kontroll- und datenflusssensitive Analyse auf der Semantik des Programms.

Die Methode der abstrakten Interpretation wird bei der Statischen Analyse genutzt, um ein Programm nicht nur auf einzelnen Werten konkret ausführen zu müssen sondern die Instruktionen eines Programms über alle möglichen Werte zu betrachten. Hierfür werden Wertemengen verwendet, die von den konkreten, möglichen Werten abstrahieren – sie werden daher abstrakte Wertemengen oder Werte einer abstrakten Domäne genannt. Eine anschauliche Domäne ist die Intervalldomäne, bei der eine untere und eine obere Grenze angegeben werden, um alle Werte dazwischen als mögliche konkrete Werte zu beschreiben. Mit diesen Wertemengen werden dann alle Instruktionen des Programms interpretiert.

Neben der hier genannten Definition der Statischen Analyse werden oft auch Prüfverfahren so bezeichnet, die lediglich syntaktische Eigenschaften wie Namenskonventionen, Code-Muster oder Aufruf-Muster überprüfen. Eine solche Erweiterung wurde auch im Rahmen der Abschlussarbeit [Rath, 2017] erarbeitet. Im Folgenden bleiben wir jedoch bei der ursprünglichen Definition.

ARCADE.PLC

ARCADE.PLC ist ein Framework zur formalen Analyse und Verifikation von SPS-Code. Es wurde am Lehrstuhl Informatik 11 der RWTH Aachen entwickelt [3, 4] und verfügt neben Statischer Analyse auch über einen Model-Checker, der Eigenschaften eines SPS-Programms beweisen kann. Die Definition des zu untersuchenden Verhaltens kann dabei in Form von CTL-Formeln oder durch Automaten erfolgen [5]. Im Gegensatz zu anderen Modelcheckern ist keine weitere Angabe eines Modells nötig, auf dem die Eigenschaften untersucht werden, da ARCADE.PLC oder kurz ARCADE dieses direkt aus dem SPS-Programm erzeugt.

Als Eingabesprachen werden derzeit Structured Text, Instruction List, Function Block Diagram und Sequencial Function Chart nach dem Standard IEC 61131-3 von 2003 [15] unterstützt, also ohne die Erweiterung durch Klassen in der neueren Version [16] der Norm. Die Eingabedateien lassen sich als einfache Textdateien, PLCopenXML-Datei oder im Format des ABB-Control-Builders zum Öffnen bereitstellen.

ARCADE bietet zwei verschiedene Interfaces, um SPS-Programme als Eingabe entgegenzunehmen und zu analysieren. Ein Eclipse-Benutzerinterface basierend auf der Rich-Client-Platform bietet die Möglichkeit, Textdateien mit dem zu untersuchenden Programmcode zu importieren, anzusehen und die Ergebnisse der Analyse darauf zu betrachten. Auch weitere Darstellungen des Programms, wie die als Kontrollflussautomat, sind damit möglich. Ein zweites Interface ist die Kommunikation über einen Websocket, über die ST-Programme an ARCADE geschickt werden können. Warnungen und Hinweise als Ergebnis werden dann im JSON-Format zurück geschickt. Beide Interfaces arbeiten mit derselben Statischen Analyse und liefern dieselben Ergebnisse.

Wertemengen- und Live-Variable-Analyse

Eines der Alleinstellungsmerkmale von ARCADE.PLC im Bereich der Statischen Analyse für SPS-Code ist die *Wertemengenanalyse* (VSA). Mit ihr werden basierend auf der Semantik des Programms Variablenwerte berechnet, die bei der Ausführung auftreten können. Die Analyse operiert dabei stets mithilfe einer oder mehreren Domänen. So wird die Ausführung des Programms nicht mit konkreten Werten simuliert, was aufgrund des großen Rechen- und Speicheraufwandes nicht möglich ist. Stattdessen werden Mengen von Variablenwerten verwendet und die Operationen des Programms auf diese Mengen angewendet bzw. interpretiert. ARCADE unterstützt neben der Intervalldomäne noch Mengen von konkreten Werten, Mengen von Intervallen und Bitvektoren.

Die beschriebene Vorgehensweise wird *abstrakte Interpretation* (AI) genannt. Das Ergebnis dieser Analyse liefert eine Überapproximation des tatsächlichen Verhaltens bzw. der tatsächlich möglichen Variablenwerte. Die Wertemengen werden in SPS-typischer Weise von den Analysen nicht nur über eine Programmausführung hinweg berechnet sondern in Abhängigkeit ihrer Variablen-Typen über die kontinuierliche Ausführung im zyklischen Betriebsablauf der SPS.

Die berechneten Wertemengen werden in einem *Kontrollflussautomaten* (CFA) gespeichert. Diese Automaten entsprechen den *Kontrollflussgraphen* (CFG), die üblicherweise für Programmanalysen verwendet werden. Ihr Unterschied ist die Semantik der Knoten und Kanten. Ein CFA speichert Instruktionen, also Befehle des Programms auf den Kanten, die im Automat als Transitionen bezeichnet werden. Knoten, also die Zustände des Automaten, formalisieren jeweils den Zustand des Programms vor bzw. nach der Ausführung einer Instruktion.

Eine weitere Analysetechnik, die von ARCADE.PLC unterstützt wird, ist die *Live Variable Analyse* (LVA). Diese Analyse prüft für jede Zuweisung einer Variable, ob diese Zuweisung an einem späteren Punkt im Programm noch einmal gelesen wird. Wurde der Wert der Variable mit einem anderen Wert überschrieben, so gilt die frühere Zuweisung nicht mehr als *live*. Diese Analyse arbeitet ebenfalls datenflussorientiert wie die VSA und liefert für diese Analyse wertvolle Informationen, durch die die VSA das Programm speichersparender analysieren kann.

Die Ergebnisse der LVA und der VSA werden zusammen mit den Ergebnissen anderer Analysen im CFA gespeichert, der dann als annotierter Kontrollflussautomat bezeichnet wird. Auf Basis dieses annotierten CFAs werden dann Prüfungen durchgeführt, um Warnungen und Fehlermeldungen daraus abzuleiten. Ein Beispiel für die VSA und die Annotation im CFA ist in Abbildung 4.1 und 4.3 dargestellt.

Abbildung 2.2 zeigt den Analyseablauf im Zusammenhang mit allen Analyseschritten. Zunächst wird das SPS-Programm mithilfe eines Parsers in einen AST geparst. Dieser wird in eine Zwischendarstellung, die *Intermediate Representation* (IR), überführt, mit der die nachfolgenden Analysen unabhängig von der verwendeten Sprache des SPS-Programms sind. Aus der IR wird dann ein CFA generiert, auf dem schließlich die Analysen arbeiten. Ergebnis ist ein annotierter CFA, auf dem die Prüfungen für Warnungen und Hinweise ausgeführt werden.

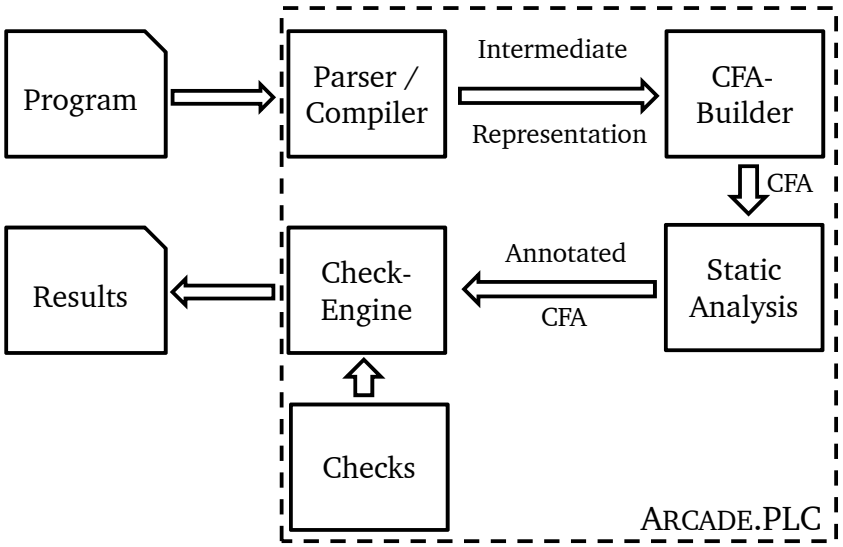


Abbildung 2.2: Architektur der Statischen Analyse in ARCADE.PLC (Abbildung angelehnt an [6])

Das Ergebnis dieser Analyse liefert eine Überapproximation des tatsächlichen Verhaltens bzw. der tatsächlich möglichen Variablenwerte. Ein Grund für die Überapproximation ist die Verwendung der Wertemengen – so können Intervall-Domänen üblicherweise keine konkreten Werte auslassen, da sie jeweils nur Werte von einem Minimalwert bis zu einem Maximalwert beschreiben. Weiterhin ist insbesondere bei der Analyse von Programmen mit Schleifen auf Analyseseite meinst ein Kompromiss zu finden zwischen der genauen Abarbeitung aller Schleifendurchläufe für ein genaueres Ergebnis und der aufzubringenden Rechenzeit und dem Speicherverbrauch für die Analyse.

Die Überapproximation ist der Grund, weshalb ARCADE.PLC möglicherweise auch False-Positive-Warnungen generiert, die bei der tatsächlichen Ausführung der SPS nicht zu einem Problem führen. Sie erzeugen dann einen Nachbearbeitungs-Aufwand, indem der Programmierer oder Tester zunächst entscheiden muss, ob gemeldete Warnungen berechtigt sind oder nicht. Davon abgesehen lassen sich mit dieser Analysetechnik jedoch Fehler finden, die mit üblichen Testmethoden nicht oder nur schwer aufzuspüren sind [48]. Weiterhin kann die Überapproximation durch Techniken wie die Verwendung mehrerer Domänen abgemildert werden.

Eine Weiterentwicklung in Zukunft kann hier die Einführung relationaler Domänen sein, die Abhängigkeiten zwischen Variablen modellieren können. Auf diese Weise kann das Programmverhalten noch genauer simuliert werden.

Reaching Definition Analyse und Slicing

Neben der LVA und der VSA unterstützt ARCADE auch die *Reaching Definition Analyse* (RDA). Auch diese Analyse arbeitet auf dem CFA und berechnet den umgekehrten Fall der LVA. Für jede gelesene Variable wird hier berechnet, welche Definitionen dieser Variable über den Kontrollflussautomaten erreicht werden können. Eine Definition der Variable

entspricht dabei einem schreibenden Zugriff auf diese Variable. Diese Abhängigkeit wird *Datenabhängigkeit* genannt.

Nun ist es im CFA möglich, dass die Definition einer Variable nur unter einer bestimmten Bedingung erfolgt, wenn eine Zuweisung beispielsweise im Block einer `IF`-Anweisung vorgenommen wird. In diesem Fall ist die Zuweisung von Variablen in der Bedingung der `IF`-Anweisung *kontrollabhängig*. Diese Abhängigkeiten werden in einem *Dominator Tree*, kurz DomTree gespeichert.

Beide Abhängigkeiten, also Daten- und Kontrollabhängigkeiten zusammen werden in ARCADE.PLC zu einem gemeinsamen Graphen verarbeitet, der die Abhängigkeiten kombiniert. So lässt sich feststellen, welche Variablen auf eine bestimmte Stelle im Programmfluss Einfluss haben. Genutzt werden diese Analysen für eine Technik namens *Slicing* (aus dem *engl.* to slice: zerschneiden). Slicing wird üblicherweise zur Fehlersuche eingesetzt [20]. Unter Angabe einer Programmzeile lassen sich mit Slicing genau die Zeilen des Programms identifizieren, die das Ergebnis der Instruktion dieser Zeile möglicherweise beeinflussen (genannt *backward Slicing*) oder die Zeilen, die von dieser Zeile beeinflusst werden können (genannt *forward Slicing*). Ohne weitere formale Definition soll dies an einem Beispiel erläutert werden (aus [37], © 2017 IEEE):

```
1 A := TRUE;
2 B := NOT(A);
3 C := TRUE;
4 IF B THEN
5   OUT := 1;
6 END_IF;
```

In diesem Beispiel besteht ein einfacher Backward-Slice auf Zeile 3 aus lediglich wieder Zeile 3. Keine andere Variable und damit keine Zeile ist für die Auswertung nötig. Ein Backward-Slice auf Zeile 4 enthält Zeilen 1, 2 und 4 da die Definition von B in Zeile 2 und damit die von A in Zeile 1 für die Auswertung in Zeile 4 notwendig sind. Das genaue Gegenteil passiert bei einem Forward-Slice auf Zeile 1, in dem die Zeilen 1, 2, 4 und 5 enthalten sind. Diese Zeilen müssten erneut ausgewertet werden, wenn sich Zeile 1 ändert.

Kapitel 3

Anforderungen an ein Unterstützungssystem

Ziel dieser Arbeit ist ein Konzept für die Unterstützung eines SPS-Programmierers bereits während der Programmerstellung mithilfe von Statischer Code-Analyse. Um die Anforderungen an ein solches Unterstützungssystem aufzustellen, wird zunächst das Einsatzszenario vorgestellt, und es wird auf Besonderheiten im Bezug auf die Entwicklungsumgebungen von SPSen eingegangen. In Vorbereitung auf eine Implementierung werden dann Anforderungen an die visuelle Repräsentation und an ein Back-End zur Berechnung der zusätzlichen Informationen gegeben.

3.1 Generelle Anforderungen

In Abschnitt 2.1.2 wurde das Vorgehensmodell über den Ablauf von Automatisierungsprojekten beschrieben. Die Methoden der Statischen Code-Analyse sollen für diese Arbeit nicht erst in der Testphase sondern bereits in der Implementierungsphase eingesetzt werden. Hierzu wurde Hypothese 1 aufgestellt, nach der der frühere Einsatz dieser Methoden einem Entwickler einen Vorteil bringt. Für den früheren Einsatz von Statischer Analyse ist die Integration der daraus gewonnenen Informationen in die Entwicklungsumgebung bzw. den Codeeditor für das SPS-Programm vorgesehen.

Entwicklungsumgebungen für SPSen wurden bereits in Abschnitt 2.1.3 einführend vorgestellt. Sie umfassen Möglichkeiten zur Programm-Entwicklung, Compiler für die Übersetzung in Maschinencode und Debugger-Anbindungen, erfüllen damit die Kriterien einer integrierten Entwicklungsumgebung (IDE). Speziell die IDEs für SPSen enthalten darüber hinaus Werkzeuge für spezifische Funktionen der Geräte. Die damit entwickelten Programme lassen sich üblicherweise nur mit der IDE des Herstellers verwenden. Schnittstellen zur herstellerunabhängigen Entwicklung wie PLCOpenXML sind oder werden zwar standardisiert [17], allerdings stellen inkompatible Datenformate in der Praxis weiterhin eine Einschränkung der Portabilität von Programmen dar. Für ein Unterstützungssystem bedeutet diese Besonderheit, dass es in eine IDE eines Herstellers integriert werden muss, um an die SPS-Programme zu gelangen und die einfache Anwendung der Statischen Analyse zu ermöglichen.

Zwei Arten von Informationen soll das zu entwickelte Unterstützungssystem dafür bieten: Warnungen über möglicherweise ungewolltes Verhalten im SPS-Programm und Informationen über Variablenwerte, die eine Variable annehmen kann. Beide Informationen lassen sich mithilfe von Statischer Analyse unter Einsatz von abstrakter Interpretation generieren. Die so erzeugten Variablenwerte können dem SPS-Entwickler im Quellcode jeweils einen Hinweis darauf geben, welchen Wert eine Variable an einer bestimmten Stelle im Code annehmen kann. Um dem Programmierer ein möglichst direktes Feedback zu geben, sollen diese Informationen zusammen mit dem SPS-Programm angezeigt werden. Eine solche Entwicklungsumgebung ist ähnlich zu Tanimotos Klassifikation von Level 3 Liveness ([50] und Abschnitt 2.2.1), wobei die durch Statische Analyse gewonnenen Erkenntnisse nicht durch die konkrete Ausführung des Programmcodes gewonnen werden, wie es Tanimoto ursprünglich definierte, sondern durch Interpretation des Programms über alle möglichen Ausführungen.

Es werden damit die folgenden Anforderungen an den Prototyp eines Unterstützungssystems gestellt:

Anforderung 1 (A1): *Das Unterstützungssystem soll Warnungen aus der Statischen Analyse des aktuellen Programms anzeigen. Dazu gehören Warnungen vor (vgl. Abschnitt 2.3):*

- *Unerreichbarem Code*
- *Array-Zugriff außerhalb der spezifizierten Grenzen*
- *Möglicher Division durch 0*
- *Variablenüberlauf*
- *Konstanten Variablen, die nicht als solche deklariert sind*
- *Fehlende* `case`*-labels*
- *(Teil-)Bedingungen, die konstant zu* `TRUE` *oder* `FALSE` *auswerten*
- *Mehrfach zugewiesene Ausgangsvariablen*

Anforderung 2 (A2): *Das Unterstützungssystem soll zu Bezeichnern in einem SPS-Programm jeweils die möglichen Werte an beliebiger Stelle des Programms darstellen können.*

Die beiden grundlegenden Anforderungen 1 und 2 beschreiben den Funktionsumfang der Unterstützungsfunktionen auf Basis der Statischen Analyse. Dieser Funktionsumfang soll dann genutzt werden, um Hypothese 1 zu untersuchen. Unterstützend folgen nun noch weitere, überwiegend nichtfunktionale Anforderungen, die den beabsichtigten Funktionsumfang näher beschreiben. Nachfolgend in Abschnitt 3.2 und 3.3 werden noch die sich daraus ergebenden Anforderungen an ein Back-End und an die visuelle Repräsentation aufgestellt.

Anforderung 3 (A3): *Das Unterstützungssystem soll jeweils eine derzeit bearbeitee POE annotieren können, dabei jedoch weitere, vom Nutzer geschriebene POEs in die Analyse einbeziehen können. Stehen POEs noch nicht zur Verfügung sollen Worst-Case-Annahmen getroffen werden.*

Anforderung 4 (A4): *Das Unterstützungssystem soll die Informationen automatisch neuberechnen, wenn sich das SPS-Programm ändert.*

Anforderung 5 (A5): *Für das Unterstützungssystem soll keine weitere Konfiguration durch den SPS-Programmierer notwendig sein.*

Diese letztgenannte Anforderung ist üblich für Werkzeuge, die Statische Analyse verwenden und stellt einen Vorteil gegenüber anderen formalen Methoden zur Analyse von Programmen dar. Bei der Integration direkt in eine Entwicklungsumgebung sollen keine Konfigurationen durch den Nutzer erforderlich sein, um die Einstiegsbarrieren zur Nutzung bewusst niedrig zu halten. Sofern für die Analyse weitere Metadaten über das Programm nötig sind, sollen diese automatisch aus dem verwendeten Editor oder der verwendeten IDE bezogen werden.

Die IEC 61131-3 [16] sieht fünf Programmiersprachen für SPS-Programme vor, von denen für die vorliegende Arbeit zunächst nur die Sprache Structured Text als Vertreter der textuellen Programmiersprachen betrachtet wird. Die Adaption auf grafische Sprachen wird im Ausblick in Abschnitt 6 erwähnt.

Anforderung 6 (A6): *Das Unterstützungssystem soll die SPS-Programmiersprache Structured Text unterstützen.*

3.2 Back-End

Als Back-End (aus dem Englischen *back-end*: *Unterbau*) werden für diese Arbeit alle Aufgaben bzw. der Prozess bezeichnet, der für die Darstellung im Editor die Daten bereitstellt. Im Back-End werden die Statische Analyse auf dem SPS-Programm durchgeführt, Meldungen generiert und Informationen über Variablenwerte berechnet und für die Darstellung aufbereitet.

Diese Berechnungen müssen im Hintergrund bei jeder Änderung des Programms durchgeführt werden. In modernen Entwicklungsumgebungen wird der Programmcode kontinuierlich oder in festen Zeitintervallen kompiliert, um den Schritt des späteren Ausführens bei Bedarf schneller durchführen zu können. Auch Fehlermeldungen des Compilers können so schon früh im Quellcode annotiert werden. Dieses Prinzip soll auch für die Analyse im Back-End gelten. Sie soll kontinuierlich ablaufen und geeignete Maßnahmen treffen, auch wenn das Programm noch nicht vollständig geschrieben ist. Eine Herausforderung ist dabei, dass ein Programm während der Entwicklung nicht notwendigerweise syntaktisch korrekt ist. So können Identifier und Schlüsselwörter noch nicht komplett ausgeschrieben sein, Ausdrücke nicht vollständig, Bedingungs- oder Schleifenblöcke nicht abgeschlossen und Zeilenende nicht terminiert sein. Ein solches Programm, deren Identifier alle bekannt

und dessen Programmcode syntaktisch korrekt ist, nennen wir *wohlformuliert*. Entsprechend muss das Back-End mit *nicht wohlformulierten* Programmen umgehen können. Diese Herausforderungen führen zu folgenden Anforderungen an das Back-End:

Anforderung 7 (A7): *Das Back-End muss Informationen mithilfe von Statischer Analyse bereitstellen. Dazu gehören Hinweise und Warnungen zu Anforderung 1 sowie die Informationen über Variablenwerte nach Anforderung 2.*

Anforderung 8 (A8): *Die Informationen im Back-End müssen geeignet schnell berechnet werden, sodass in der grafischen Darstellung keine auffälligen Verzögerungen der Informationsdarstellung auftreten.*

Anforderung 9 (A9): *Das Back-End muss geeignet damit umgehen können, dass das zu analysierende Programm noch nicht wohlformuliert ist. Dazu gehören insbesondere die Fälle:*

- *Das Programm weist syntaktische Fehler auf*
- *Bezeichner von Variablen oder POEs sind nicht bekannt*

3.3 Visuelle Repräsentation

Die visuelle Repräsentation der Ergebnisse ist ein wichtiger Aspekt bei der Akzeptanz eines Unterstützungssystems. Im Hinblick auf die zu untersuchende Hypothese 1 ist es daher wichtig, eine Darstellungsform zu finden, die der SPS-Programmierer gewohnt ist oder nach kurzer Zeit in seine Arbeitsweise übernehmen kann.

Quellcode-Texteditoren sind in diesem Zusammenhang zu einem wichtigen Mittel von IDEs geworden, dem Programmierer bei der Strukturierung Hilfestellung und Feedback zum entwickelten Programm zu geben [52]. Dabei kommen üblicherweise grafische Annotationen nahe dem Quellcode zum Einsatz, um kontextabhängige Meldungen über bestimmte Teile des Programmcodes darzustellen [28]. Diese grafischen Annotationen sind heute in kommerziellen und open-source IDEs, im speziellen auch in IDEs für SPSen, verbreitet.

Grafische Annotationen werden generell verwendet, um Quellcode anhand syntaktischer oder semantischer Merkmale zu formatieren, also zum Beispiel um Schlüsselwörter hervorzuheben oder Blöcke als zusammenhängen zu kennzeichnen. Weiter werden Annotationen wie Unterstreichungen üblicherweise genutzt, um auf Fehler hinzuweisen. IDEs aus dem Umfeld der SPSen nutzen diese Methode üblicherweise um Fehler des Compilers oder einer vorgeschalteten Sprach-Konsistenzprüfung darzustellen.

Die Meldungen der Statischen Analyse, die laut Anforderung 1 angezeigt werden sollen, geben anhand des bisher geschriebenen Quellcodes Hinweise aus, wenn das SPS-Programm möglicherweise Fehler enthält oder es zu nicht beabsichtigtem Verhalten kommen kann. Nicht beabsichtigtes Verhalten kann zum Beispiel beim erneuten Zuweisen an eine Ausgangsvariable auftreten. Mögliche Fehler sind Variablenüberläufe bei Operationen mit Variablen oder Zugriffe außerhalb von Array-Grenzen. Beide dieser Arten von Meldungen sollten den Entwickler auf diesen Umstand aufmerksam machen und ihm erlauben, geeignete Maßnahmen zur Verbesserung oder Fehlerbeseitigung zu ergreifen.

Zwei Punkte sind hierbei besonders wichtig. Erstens sollten die Meldungen der Statischen Analyse mit möglichst kurzer Verzögerung bereitstehen, um den Entwickler noch in seinem Arbeitsfluss beim Bearbeiten einer betroffenen Stelle unterstützen zu können. Dieser Punkt wird bereits durch Anforderung 8 gefordert. Zweitens sollten Meldungen und Variableninformationen eine visuelle Darstellung erhalten, die ihrer Dringlichkeit und Relevanz entspricht. In Systemen mit proaktivem Informationsangebot ist dies oft nicht der Fall, was zu geringerer Nutzerakzeptanz führt [47]. Von Gluck et al. [21] wurde in einem ähnlichen Szenario dazu jedoch experimentell nachgewiesen, dass Hinweissysteme dieser Art besser angenommen werden, wenn wichtigere Meldungen auffälliger dargestellt werden als weniger wichtige.

Für die visuelle Repräsentation wird daraus gefolgert, dass es mindestens zwei Stufen unterschiedlicher Auffälligkeit für die Darstellung geben soll, sodass Hinweise und Warnungen anders dargestellt werden als Informationen über Variablenwerte.

Anforderung 10 (A10): *Die Darstellung der Informationen aus Anforderung 1 und 2 soll in einer üblichen Entwicklungsumgebung für SPS-Code integriert sein. Der Entwickler soll die IDE wie gewohnt nutzen können, auch wenn er die Informationen nicht nutzt. Warnungen des Compilers sollen weiterhin angezeigt werden.*

Anforderung 11 (A11): *Warnungen und Hinweise aus der Statischen Analyse sollen im Quellcode an den entsprechenden Stellen sichtbar sein, ohne den Arbeitsfluss des Entwicklers zu unterbrechen. Es soll möglich sein, genauere Informationen über die annotierte Codestelle zu erhalten.*

Anforderung 12 (A12): *Informationen über Variablenwerte sollen im Quellcode auf weniger auffällige Weise als Warnungen und Hinweise dargestellt werden.*

Mit den hier aufgestellten Anforderungen wird im nachfolgenden Abschnitt nun die Implementierung des Prototyps für dieses Unterstützungssystem beschrieben.

Kapitel 4

Konzept und prototypische Implementierung

Für das Unterstützungssystem für SPS-Programmierer wurden in Kapitel 3 Anforderungen aufgestellt und diskutiert. Sie sind darauf ausgerichtet, formale Methoden in Form der Statische Analyse als Unterstützung schon bei der Eingabe von SPS-Programmen anzubieten und nicht erst in einem späteren Schritt in einer Testphase. Ausgerichtet sind die Maßnahmen alle auf eine Unterstützung des Programmierers. So wird eine Umgebung geschaffen, mit der Hypothese 1 untersucht werden kann – die Frage, ob Informationen der Statischen Analyse bei der Entwicklung eines SPS-Programms unterstützen können.

Drei zentrale Bestandteile wurden für die Implementierung der Anforderungen und im Rahmen der vorliegenden Arbeit geplant und umgesetzt:

1. Die Erweiterung des ARCADE.PLC-Frameworks zur Nutzung als Back-End für das Unterstützungssystem und als Editor-Prototyp
2. Entwicklung eines Codesys-Plugins zur Integration der Analysedaten in eine Entwicklungsumgebung für SPS-Programme
3. Die Erweiterung des Back-Ends um eine inkrementelle Analyse zur Performance-Steigerung

ARCADE.PLC wird als Back-End und damit zur Durchführung der Statischen Analyse verwendet. Es liegt als Implementierung in Java 1.8 vor und konnte um einige Funktionen erweitert werden, um notwendige Informationen für das Unterstützungssystem bereitzustellen. Auf die vorhandenen Ressourcen in ARCADE.PLC geht dabei der erste Abschnitt 4.1 ein, bevor die Erweiterungen dann im folgenden Abschnitt im Detail vorgestellt werden.

Als Evaluationsplattform wurde Codesys des Herstellers 3S-Smart Software Solutions[1] ausgewählt. Codesys ist ein komplettes Programmiersystem und umfasst eine IDE für die Entwicklung von Programmen für SPSen mehrerer Hersteller. Die IDE selbst kann dabei kostenfrei genutzt werden, lediglich die Ausführungsplattformen auf SPSen oder eingebetteten Geräten müssen mit Lizenzen versehen werden. Hersteller von Steuerungssystemen passen Codesys üblicherweise mithilfe eines SDKs an ihre Geräte an. Dieses SDK stand für

[1] Hersteller-Webseite: `https://de.codesys.com/`

die hier vorgestellte Implementierung ebenfalls zur Verfügung. Abschnitt 4.3 geht auf die Strukturierung und Umsetzung des entwickelten Plugins ein.

Abschnitt 4.4 widmet sich schließlich einer Erweiterung des Back-Ends mit dem Ziel der Performance-Steigerung insbesondere bei großen Programmen.

4.1 ARCADE.PLC als Grundlage

ARCADE.PLC liegt als grundlegendes Analyseframework für die Statische Analyse im Quelltext vor. Eine Einführung darin wurde bereits in Abschnitt 2.3 gegeben. Für das zu implementierende Unterstützungssystem bietet es eine geeignete Basis und wird daher im Folgenden als Back-End verwendet. Warnungen und Hinweise über potenziell fehlerhaftes Programmverhalten werden im Zuge der Statischen Analyse bereits generiert. Damit ist die Anforderung 1 vorbereitet, die die Darstellung dieser Warnungen fordert. Auch die Forderung nach der Unterstützung von ST in Anforderung 6 ist mit diesem Framework bereits erfüllt.

Die Ausgabe möglicher Variablenwerte als Approximation des Laufzeitverhaltens ist dagegen noch nicht vorgesehen. Für diese Anforderung 2 muss ARCADE.PLC daher erweitert werden. Die Informationen befinden sich jedoch im annotierten CFA in Form von Wertemengen der VSA. Sie können gesammelt und den entsprechenden Codeabschnitten wieder zugeordnet werden, wie in Abschnitt 4.2.1 beschrieben wird.

Anforderung 5 fordert die konfigurations-freie Nutzung der Statischen Analyse. Diese Anforderung ist durch ARCADE wieder bereits erfüllt – für die Analyse wird lediglich das Programm mit einer ausgezeichneten Haupt-POE und Angabe eines verwendeten Sprachdialekts benötigt. Die Haupt-POE entspricht der derzeit bearbeiteten Organisationseinheit, die annotiert werden soll. Es ist darüber hinaus möglich, weitere POEs anzugeben, die vom Programmierer geschriebene Funktionen beinhalten. Der Dialekt wirkt sich auf den Parser von ARCADE aus und kann so kleine Unterschiede der Programmsemantik in herstellereigenen Implementierungen der Sprache Structured Text berücksichtigen. Diese Angabe kann von einem Plugin mitgeliefert werden, wodurch Anforderung 5 erfüllt ist. Da ARCADE.PLC somit auch die Analyse mehrerer POEs unterstützt, ist auch Anforderung 3 erfüllt. Darüber hinaus ist dort gefordert, dass die Analyse auch möglich sein soll, wenn POEs nicht zur Verfügung stehen, unbekannt oder noch nicht implementiert sind. Stehen POEs nicht zur Verfügung, da sie im Zuge der Implementierung noch nicht geschrieben sind oder sich in proprietären, vorkompilierten Bibliotheken befinden, auf deren Quellcode nicht zugegriffen werden kann, so trifft ARCADE.PLC Worst-Case-Annahmen. Das bedeutet, dass für Variablen, auf die durch diese POE geschrieben wird, stets der maximal mögliche Wertebereich der Variable angenommen wird. Somit ist sichergestellt, dass der tatsächliche Variablenwert bzw. das tatsächliche Programmverhalten durch diese Überapproximation abgedeckt ist.

4.2 Back-End und Eclipse-Editor

Wie im letzten Abschnitt beschrieben, erfüllt das Framework ARCADE.PLC als Back-End bereits die Anforderungen 1, 3, 5 und 6 aus Kapitel 3. Die Erweiterungen, die im Rahmen der vorliegenden Arbeit integriert wurden, verfolgen alle das Ziel, die Nutzung von ARCADE.PLC als Werkzeug während der Programmierung von SPS-Programmen zu erleichtern.

Als Prototyping-Plattform für die grafische Oberfläche des Editors wird der integrierte Editor in der Eclipse-RCP-Oberfläche verwendet. Dieser Editor bietet auf Structured Text bezogen bisher nur die Möglichkeit, Programme als Textdatei zu importieren und Warnungen darin anzeigen zu lassen. Ein Modifizieren und erneutes Analysieren war bisher kein Anwendungsfall und ist über die grafische Oberfläche so auch nicht möglich. Der Editor zeigt das Programm schreibgeschützt an und ein erneutes Parsen des Programms ist nur unter bestimmten Umständen möglich. In den folgenden Abschnitten wird auf die Modifikationen des Editors eingegangen, um damit die Funktionalität des Back-Ends zu testen und erste Entwürfe einer visuellen Repräsentation im Editor umzusetzen. Weiter wird erläutert, wie mit nicht parsbaren Programmen umgegangen werden kann und wie die Webschnittstelle erweitert wurde, um die gewonnenen Wertemengen auch in einer IDE anzeigen zu können. Dem voraus geht die Gewinnung der Variablenwerte aus dem annotierten CFA, um diese Werte bzw. Mengen im Programm anzeigen zu können.

4.2.1 Extraktion von Variablenwerten

Die Statische Analyse von ARCADE.PLC berechnet mithilfe der abstrakten Interpretation bereits für jeden Zustand im CFA mögliche Variablenwerte. Wertemengen stehen so den Analysen zur Verfügung, die auf dem CFA arbeiten. Die direkte Zuordnung dieser Wertemengen aus dem CFA auf das Eingabeprogramm ist aufgrund des Übersetzungsprozesses in ARCADE.PLC jedoch nicht ohne Weiteres möglich, da er aus einer Zeile des Eingabeprogramms eine aber auch mehrere Instruktionen und damit Kanten im Graph erstellt. Daher wurde ein Verfahren implementiert, mit dem die Wertemengen aus dem CFA zurück auf den SPS-Code zur zeilenweisen Auswertung und Anzeige abgebildet werden können.

Prä- und Post-Mengen

Für das weitere Vorgehen betrachten wir zunächst die folgende Zuweisung:

```
1 X := X+1;
```

Bei dieser Zuweisung werden zwei Wertemengen für die Variable `X` gespeichert. Zunächst die Werte von `X` vor dem Ausführen der Zuweisung, also der Wert, den `X` auf der rechten Seite der Zuweisung hat. Außerdem die Werte von `X` um eins inkrementiert, wenn die Zeile ausgeführt wurde und `X` entsprechend um eins erhöht wurde. Beide Werte sollen im Quellcode annotiert werden können und müssen daher für diese Zeile zur Verfügung stehen. Wir nennen die Wertemenge, die Variable `X` vor Ausführung der Zeile annehmen kann, die *Prä-Menge* für `X`. Die Menge, die sie nach der Zeile annehmen kann, nennen wir entsprechend *Post-Menge* für `X`. Zu jeder Zeile werden für üblicherweise mehrere Variablen

Wertemengen berechnet. Die Prä- und Post-Mengen einer Zeile umfassen im Folgenden dann die Prä- bzw. Post-Mengen aller dort bekannten Variablen.

Werden Variablen an einer Stelle lediglich gelesen, so sind die Prä- und Post-Mengen der Variable an einer Zeile gleich, da nach dem Ausführen einer Zeile keine Änderung am Variablenwert erfolgt. Dies ist beispielsweise bei Zuweisungen der Fall, bei der eine Variable nur auf der rechten Seite der Zuweisung vorkommt.

Structured Text ermöglicht es, mehrere Instruktionen durch ein Semikolon getrennt in eine Zeile zu schreiben. Jeder dieser Instruktionen erzeugt im CFA eine neue Variablenbelegung, allerdings ist die Schreibweise in ST-Code unüblich und kann zu schwer lesbarem Code führen. Im nachfolgenden Algorithmus wird daher ein pragmatischer Ansatz genutzt und lediglich der Variablenzustand vor und nach dem Ausführen einer textuellen Zeile Programmcode berücksichtigt. Nur diese Informationen werden später für die Darstellung im Editor benötigt.

Assoziation zwischen ST-Codezeile und CFA

Für die Zuordnung der Wertemengen zu einer Zeile ist es nötig, den Prozess zur Ableitung des CFAs zu betrachten. Im Analyseprozess von ARCADE.PLC (siehe Abbildung 2.2) wird aus dem Eingabeprogramm zunächst ein AST generiert. In ASTs bleibt die Information gespeichert, an welcher Stelle des Eingangsprogramms die Token, also die grundlegendsten Programmbestandteile, gelesen wurden. Hier ist eine Zuordnung auf den Quelltext also möglich. Bei der weiteren Transformation des Programms in Instruktionen der IR bleibt die Zuordnung ähnlich erhalten – Hier speichert ARCADE einen Verweis auf Einträge des ASTs, in dem wiederum Zeileninformationen hinterlegt sind. Im CFA werden die Instruktionen dann auf die Transitionen des Automaten aufgetragen. Letztlich kann so von den Transitionen im Automaten auf die Zeile des Ursprungsprogramms geschlossen werden. Eine direkte Zuordnung von einer Zeile auf eine Transition ist jedoch nicht ohne weiteres möglich.

Abbildung 4.1 zeigt ein einfaches ST-Programm, 4.2 die Darstellung als IR und Abbildung 4.3 den daraus erstellten und annotierten CFA. Zur einfacheren Darstellung wurde der AST hier ausgelassen und die Zeilen-Zuordnung direkt an der IR dargestellt. Im CFA stehen die berechneten Variablenmengen jeweils annotiert an den Zuständen des Automaten. Wie zuvor motiviert, sind bei Zuweisungen zwei Wertemengen wichtig: Die Prä- und die Post-Mengen für jede Zeile ST-Code mit Instruktionen. In einem CFA sind diese Mengen jeweils am Vorgängerzustand und am Nachfolgezustand einer Instruktion annotiert. Der Zustand mit der Bezeichnung `3-2` enthält so die Informationen, dass `X` an dieser Stelle immer den Wert 1 hat und `Y` die Werte 0 und 2 haben kann. Der Wert von `X` kommt durch die einzige Transition des Automaten zustande, durch die der Variable konstant der Wert 1 zugewiesen wird. Der Wert von `Y` ist das Resultat von zwei Durchläufen durch den Automaten. Zunächst weist ein Funktionsblock bei der Initialisierung Variablen den Wert 0 zu, was durch den Standard IEC 61131-3 festgelegt ist, sofern kein anderer Initialwert vorgeben ist. Nachdem die folgenden Zeilen des Programms ausgeführt werden, behält `Y` den Wert 2 auch bis zu einem möglichen erneuten Aufruf des Blocks, da die Variable als Typ `VAR` deklariert wurde und kein anderer Initialwert vorgegeben wurde. Daher hat Y in Zustand `3-2` die möglichen Werte 0 und 2.

```
1  FUNCTION_BLOCK Simple
2  VAR
3  X : INT;
4  Y : INT;
5  END_VAR
6  VAR_OUTPUT
7  Z : INT;
8  END_VAR
9  X := 1;
10 Y := 2;
11 Z := X + Y;
12 END_FUNCTION_BLOCK
```

Abbildung 4.1: Ein einfacher Funktionsblock geschrieben in Structured Text. Es berechnet aus zwei vorgegebenen Werten die Summe.

```
1  ASSIGN Simple.X ← 1                  //Code–Zeile 9
2  ASSIGN Simple.Y ← 2                  //Code–Zeile 10
3  ASSIGN Simple.Z ← Simple.X + Simple.Y //Code–Zeile 11
4  RETURN
```

Abbildung 4.2: Darstellung des Programms aus Abbildung 4.1 in IR. Als Kommentar der Verweis auf die Code-Zeilen des ST-Quelltextes.

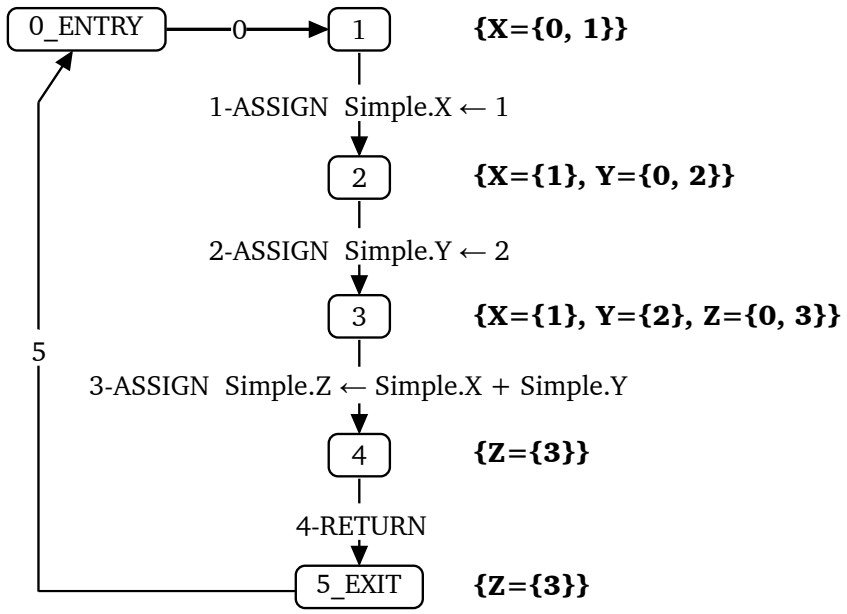


Abbildung 4.3: Der annotierte CFA zu 4.2. Zustände sind jeweils mit der Ausführungspriorität bzw. der Ausführungsreihenfolge beschriftet. Auf den Transitionen sind die Instruktionen aufgetragen. Daneben sind die berechneten, relevanten Wertemengen der VSA dargestellt.

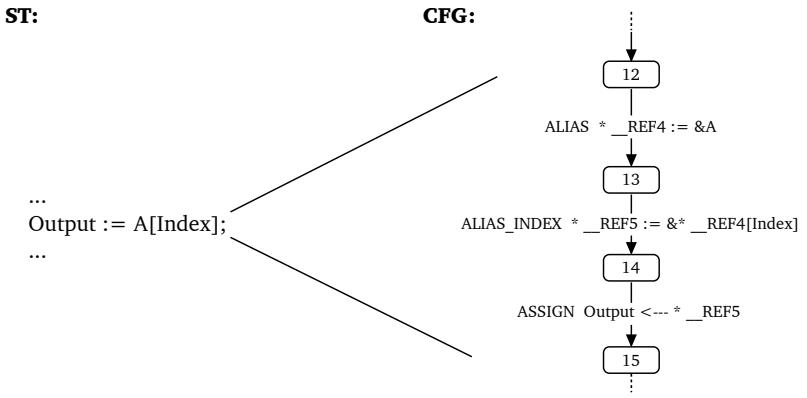


Abbildung 4.4: Repräsentation eines Array-Zugriffs in ST im CFA in ARCADE

Die Zuordnung von Variablenmengen auf Instruktionen in Structured Text ist im dargestellten Beispiel in Abbildungen 4.1, 4.2 und 4.3 einfach, da die Zeilen 9-11 im ST-Code sich in die gleiche Anzahl an Anweisungen in der Intermediate Representation übersetzen lassen und damit jeweils genau einer Kante im Graph entsprechen. Dies trifft jedoch nicht auf alle Instruktionen zu, da der Sprachumfang von ST größer ist als jener der IR und einige ST-Instruktionen daher in mehrere IR-Instruktionen übersetzt werden. Ein Beispiel dafür ist in Abbildung 4.4 gegeben. Hier wird eine Zuweisung eines Array-Elements zu einer normalen Variable `Output` in ST und im CFG dargestellt. ARCADE erzeugt hierfür zunächst einen internen Alias für das Array `A`. Die Instruktion auf der nächsten Transition ist ein Alias auf das konkrete Array-Element an diesem Index. Erst danach wird das `Assign` durchgeführt, also die tatsächliche Zuweisung.

Durch diese Aufsplittung von ST-Instruktionen ist es nötig, pro Zeile mehrere Kanten im Graph zu betrachten und die Wertemengen mehrerer Zustände zusammenzuführen. Dies tritt auch bei Zugriffen auf Struct-Datentypen und die Nutzung von `IF`-Anweisungen zu.

Relevante Wertemengen versus vollständige Analyse

In Abbildung 4.3 fällt auf, dass nicht an jedem Zustand des Graphen alle Wertemengen annotiert sind. Hier kommt die sogenannte *Sparse Analysis* der VSA zum Einsatz, die von Sebastian Biallas [3] entwickelt wurde und an jedem Zustand des CFA nur die relevanten Wertemengen speichert. Relevant heißt in diesem Zusammenhang, dass an einem Zustand eine der folgenden Bedingungen für die Variable zutrifft:

- Die Variable wird in der darauffolgenden Instruktion verwendet.
- Die Variable wurde als konstant identifiziert.
- Die Variable ist an diesem Zustand *live*.

Eine Variable wird entsprechend der Live Variable Analyse (LVA) *live* genannt, wenn die Zuweisung einer Variable an einer nachfolgenden Stelle im Programm sichtbar ist. Diese Eigenschaft ist für eine Variablenzuweisung beispielsweise nicht mehr erfüllt, wenn sie

nach einer Zuweisung ein weiteres Mal geschrieben wird, ohne zuvor gelesen zu werden. Es ist dann aus Speicher- und Performance-Gründen nicht sinnvoll, diesen Variablenwert weiterhin im CFA zu speichern. Entsprechend sind die Wertemengen für diese Variable an einem solchen Punkt ggf. nicht mehr verfügbar.

Für eine Annotation der Variablen in einem sich nicht ändernden Programm ist dies keine Einschränkung. Sollen jedoch mehr Informationen bereitgestellt werden, sodass sich auch solche Variablen annotieren lassen, die an einem Punkt im Programm aktuell noch nicht vorkommen, muss das Kriterium der VSA zum Speichern der Wertemengen an einem Zustand angepasst werden, um für jede Zeile die dort gültigen Variablenwerte jeder Variable berechnen zu können. *Relevant* bedeutet in diesem Kontext dann relevant für das aktuelle Programm gegenüber einer vollständigen VSA, die alle möglichen Werte berechnet.

Die Implementierung in ARCADE.PLC musste hierfür nur geringfügig angepasst werden. Es wird an jedem Zustand dann nicht auf die oben genannten Kriterien hin geprüft, ob Wertemengen für eine Variable zu speichern sind. Stattdessen werden die Mengen jeweils für alle Variablen im Gültigkeitsbereich der POE berechnet und gespeichert. Diese Modifikation wurde konfigurierbar gemacht, um je nach Bedarfsfall alle oder nur die relevanten Wertemengen zu berechnen. Abbildung 4.5 zeigt den Graph zum Programm in Abbildung 4.1 und 4.2 mit vollständig annotierten Wertemengen.

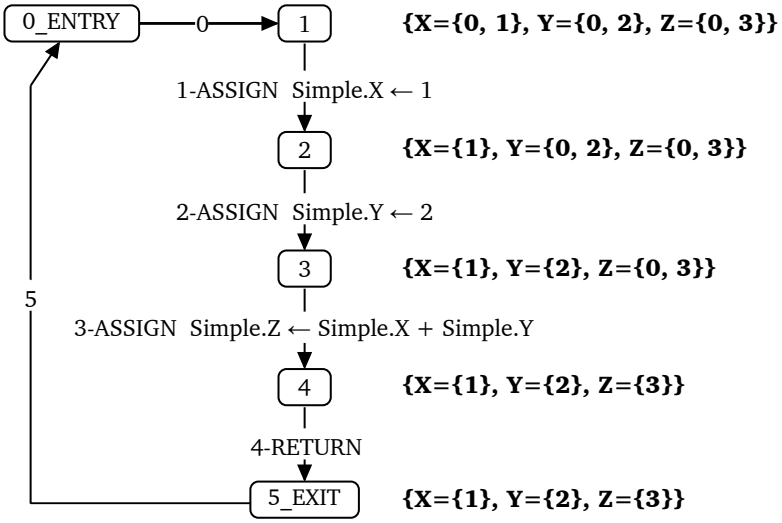


Abbildung 4.5: Der CFA zu 4.2 mit vollständigen Wertemengen. Im Gegensatz zu Abbildung 4.3 werden alle Variablen an allen Knoten gespeichert, auch wenn sie im folgenden Programm nicht gelesen oder dort überschrieben werden.

Durch die Speicherung zusätzlicher Werte im CFA wird die Analyse langsamer und verbraucht mehr Speicher. Werden kleinere Programme mit weniger als 50 Zeilen analysiert, so sind Auswirkungen nicht messbar. Ein Beispielprogramm mit 600 Codezeilen und 1000 Variablen benötigte jedoch etwa 15 % länger als bei einer Analyse, die nur auf relevanten Wertemengen arbeitet. Weiterhin war die Analyse nur lauffähig, wenn zuvor der verfügbare Speicher der ausführenden Java Virtual Machine (JVM) auf mindestens 1024 MB verdoppelt wurde.

Der Algorithmus

Nach der Darstellung der Zusammenhänge zwischen ST-Programmcode, IR und CFA sowie der Modifikation der Wertebereichsanalyse folgt nun das im Rahmen dieser Arbeit entwickelte Verfahren bzw. der Algorithmus für die Extraktion der Wertemengen. Ausgangspunkt dafür ist ein annotierter CFA. Gesucht sind zu jeder Zeile des Structured-Text-Programms die Prä- und Post-Mengen.

Der Algorithmus verwendet eine implementierte Datenstruktur namens DMap, die aus zwei geschachtelten Map-Datenstrukturen bestehen. Das Indexfeld der ersten eckigen Klammer ordnet die Einträge nach Programmzeile ein. Das nachfolgende Feld ordnet die Einträge nach Variable, deren Wertemengen dort abgelegt werden. Ein Eintrag in der DMap ist ein Tupel, um die Prä- als auch die Post-Mengen dort speichern zu können. Daneben wird noch die Funktion *findeZeile(e)* verwendet, die zu einer übergebenen Instruktion *e* mithilfe der Zeileninformationen in der IR bzw. dem AST die Zeile identifiziert, für die diese Instruktion generiert wurde.

In Algorithmus 1 ist das Verfahren beschrieben. Für jede Transition wird die dort stehende IR-Instruktion geprüft. Handelt es sich um interne Anweisungen wie `RETURN`, oder eine leere Transition zurück zum Startzustand, also um eine Kante ohne ST-Instruktion bzw. Entsprechung im ST-Code, so wird sie nicht weiter betrachtet. Anderenfalls wird der Vorgängerzustand und der folgende Zustand betrachtet. Da der CFA ein gerichteter Graph ist, werden diese als Start- und Endzustand bezeichnet. Es wird dann über die Variablen im Startzustand iteriert. Dies sind entweder alle für die Instruktion relevanten Variablen oder alle Variablen im Geltungsbereich der Instruktion.

Um nun die Informationen mehrerer Zustände zur Kombination aus Zeile und Variable zu speichern, wird in dieser DMap nun der Startzustand und der Endzustand gemeinsam in einem Eintrag gespeichert. Befindet sich dort bereits ein Eintrag, so wird unter den beiden jeweils der Zustand mit der niedrigsten Priorität als Startzustand übernommen und der mit der höchsten Priorität als Zielzustand. Die Prioritäten der Zustände entsprechen im CFA der Ausführungsreihenfolge. Eine Instruktion weiter vorne im Programm hat also eine niedrigere Priorität als eine nachfolgende Instruktion. Besteht eine Zeile ST-Code aus mehreren Transitionen im Automat, ist so sichergestellt, dass für eine Zeile immer die Wertemengen vor der ersten bzw. nach der letzten Instruktion übernommen werden. Somit wird der Zustand der Variablen vor bzw. nach Ausführen der Zeile korrekt erfasst.

Nachdem dieser Algorithmus die Werte in DMap gesammelt hat, kann dann auf die Wertemengen über die beiden Indizes zugegriffen werden. Die so berechneten Werte können in einer grafischen Darstellung dann verwendet werden, um damit den Quelltext zu annotieren. Abbildung 4.6 zeigt das Ergebnis des Algorithmus angewandt auf den ST-Programmcode aus Abbildung 4.1. In den später vorgestellten Benutzeroberflächen werden die Mengen nicht als Kommentare sondern anderweitig als Annotation zu der Zeile dargestellt.

```
Daten: Annotierter CFA
Ergebnis: DMap zum Zugriff auf Wertemengen über Zeilen und Variablen
Für alle Transitionen e des CFA
    Wenn e eine Instruktion enthält dann
        zeile = findeZeile(e)
        startzustand = e.start
        zielzustand = e.ziel
        Für alle gültigen Variablen var in startzustand
            // Behält jeweils den Zustand mit der kleinsten Priorität (= Vorgänger) und den mit der
               höchsten (= Nachfolger) zu einer Kombination aus Zeile und Variable
            bekannt = DMap[zeile][var]
            Wenn bekannt == leer dann
                bekannt.Prä = startzustand.WerteVon(var)
                bekannt.Post = endzustand.WerteVon(var)
            Ende
            Sonst
                Wenn priorität(startzustand) < priorität(bekannt.Prä) dann
                    bekannt.Prä = startzustand.WerteVon(var)
                Ende
                Wenn priorität(endzustand) > priorität(bekannt.Post) dann
                    bekannt.Post = endzustand.WerteVon(var)
                Ende
            Ende
            DMap[zeile][variable] = bekannt
        Ende
    Ende
Ende
// Zugriff auf Wertemengen dann über:
DMap[zeile][variable].Prä
DMap[zeile][variable].Post
```

Algorithmus 1: Entwickeltes Verfahren zur Zuordnung im CFA gespeicherter Wertemengen zu konkreten Zeilen des Eingabeprogramms

```
 9  X := 1;       //Prä: {X={0,1}, Y={0,2}, Z={0,3}} Post: {X={1}, Y={0,2}, Z={0,3}}
10  Y := 2;       //Prä: {X={1}, Y={0,2}, Z={0,3}}   Post: {X={1}, Y={2}, Z={0,3}}
11  Z := X + Y;   //Prä: {X={1}, Y={2}, Z={0,3}}     Post: {X={1}, Y={2}, Z={3}}
```

Abbildung 4.6: ST-Programmcode-Ausschnitt aus Abbildung 4.1 mit vollständigen Prä- und Post-Wertemengen extrahiert aus dem CFA durch Algorithmus 1

Laufzeit

Für diese Anwendung ist es sinnvoll, die Laufzeit des entworfenen Algorithmus zu betrachten. Unter Verwendung der grafischen Eclipse-Oberfläche wurden dazu Beispielprogramme analysiert und die Laufzeiten der Analyseschritte untersucht. Bei Programmen mit unter 50 Codezeilen bleibt die Laufzeit der kompletten Statischen Analyse jeweils deutlich unter 100 ms. Der Algorithmus zur Extraktion der Wertemengen hat daran einen Anteil von wenigen Millisekunden. Bei kleineren Programmen fällt also der zusätzliche Berechnungsaufwand nicht ins Gewicht.

In einem größeren Programm mit 600 Zeilen Code und 1000 Variablen benötigt die Statische Analyse etwa 7200 ms, wenn nur die relevanten Wertemengen zuvor errechnet wurden. Der Anteil des neuen Algorithmus liegt bei etwa 550 ms, in diesem Fall also bei etwa 8 %. Wurden zuvor die vollständigen Wertemengen durch die VSA berechnet, steigt die Gesamtdauer der Analyse auf 8300 ms, wovon 1200 ms auf den Algorithmus entfallen, also etwa 15 %. Bei größeren Programmen ist die Gesamtdauer der Analyse ein Thema, das zusammen mit einer detaillierteren Betrachtung aller involvierten Schritte in Abschnitt 4.4.1 untersucht wird.

Werden für das Back-End abschließend die Anforderungen betrachtet, so ist Anforderung A7 mit dem entwickelten Algorithmus erfüllt. Gefordert waren die Bereitstellung von Warnungen und Hinweisen sowie der Variablenwerte an jedem Punkt im Programm. Anforderung 9 fordert die Analyse auch syntaktisch inkorrekter Programme und die Analyse trotz nicht verfügbarer POEs. Während letzteres ein bereits in ARCADE.PLC implementiertes Verhalten ist, kann das Back-End nicht mit syntaktisch inkorrekten Programmen umgehen. In diesem Fall wird lediglich die Meldung erzeugt, an welcher Stelle der integrierte Parser einen Fehler entdeckt hat. Diese Anforderung wird daher als Bestandteil der Integration in den Editor umgesetzt, worauf Abschnitt 4.2.3 noch eingeht.

Anforderung 8 fordert eine geeignet schnelle Berechnung der Ergebnisse. Diese Anforderung muss im Kontext des Anwendungsfalls betrachtet werden. Wie zuvor erwähnt, ist die Analyse bei großen Programmen rechen- bzw. zeitaufwändig. Bei einzelnen POEs mit einer praxisnahen Größe dauert die Analyse dagegen nur wenige hundert Millisekunden. Überprüft wurde dies mithilfe der Aufgaben aus der Evaluation in Abschnitt 5.2.2. Eine dort verwendete POE besitzt 44 Variablen und einen 126 Zeilen langen Programmcode. Auf diesem Programm benötigt die hier vorgestellte Analyse im Mittel 200 ms für die komplette Berechnung. Sie ist damit schneller als die oft im Hintergrund laufende continuous compilation in Maschinencode, mit deren Hilfe IDEs üblicherweise syntaktische Fehler annotieren können. Für den betrachteten und evaluierten Anwendungsfall kann Anforderung 8 damit als erfüllt betrachtet werden. Abschnitt 4.4 wird sich noch mit Optimierungen beschäfti-

gen, um den Berechnungsaufwand der Statischen Analyse mithilfe eines inkrementellen Verfahrens zu verkleinern. Dies gilt insbesondere im Hinblick auf größere Programme.

4.2.2 Editor-Modifikationen

Die grafische Oberfläche von ARCADE.PLC ist als Eclipse-Rich-Client-Plattform-Anwendung[2] implementiert. Wie in der Entwicklungsumgebung Eclipse selbst sind damit IDE-ähnliche Programme mit einem Pluginsystem realisierbar. ARCADE implementiert damit eine Infrastruktur zum Laden und Anzeigen von SPS-Programmen, die es erlaubt, Statische Analyse und Modelchecking darauf auszuführen. Eine Entwicklungsmöglichkeit innerhalb von ARCADE war bisher kein Anwendungsfall und daher nicht vorgesehen. Es fehlt die Möglichkeit, ein Programm zu editieren und zu speichern. Weiterhin ist die Analyse-Infrastruktur bisher nicht darauf ausgelegt, einmal berechnete Informationen neu zu generieren.

Diese zunächst trivial klingenden Einschränkungen sind jedoch tiefer in den Grundlagen von ARCADE.PLC verankert als zunächst vermutet. Die Codebasis für dieses Projekt wurde bereits 2005 aufgebaut, damals als Modelchecker für Mikrocontroller. Erweiterungen und neue Plugins führten zur Funktionalität, die ARCADE.PLC heute bereitstellt. Abhängigkeiten innerhalb der bestehenden Codebasis führten dazu, dass sich Modifikationen und Erweiterungen wie diese jedoch nur mit größerem Aufwand integrieren lassen.

Neben der reinen Darstellung und Veränderung des Programms, muss dieses neu geparst werden können, um so den Analyseprozess von ARCADE auch erneut durchlaufen zu können. Mehrfach vorhandene Verweise auf gelesene oder generierte Daten und ein nichtlinear implementierter Analyse-Prozess mussten überarbeitet werden. Der nichtlineare Analyse-Prozess bedeutet in diesem Fall, dass der AST zu einem SPS-Programm bereits erstellt wurde, wenn es in die IDE importiert wird. So ließen sich Syntaxfehler bereits in einer Übersicht verfügbarer Programme noch vor dem Öffnen in einem Editor darstellen.

Die mehrfach vorhandenen Verweise erlaubten es bisher auch nicht, eine weitere Version eines Programms zu analysieren. Dies ist auch für die Nutzung von ARCADE als Back-End ungünstig, da es dann vollständig neu initialisiert werden müsste.

Die folgenden Punkte wurden daher im Rahmen dieser Arbeit in ARCADE.PLC umgesetzt:

- Bereitstellen eines Editors mit der Möglichkeit, ein Programm mithilfe von Statischer Analyse zu überprüfen
- Ändern und speichern des Quelltextes eines in Structured Text geschriebenen Programms
- Löschen und Neuberechnen der Analyseergebnisse auf Basis des neuen Programms
- Darstellung von Wertemengen an Variablen

Der existierende Editor in ARCADE konnte als Vorbild genutzt werden. Es wurde daran angelehnt das Eclipse-Editor-Element *STEditor* entwickelt, welches parallel zum existierenden Editor genutzt werden kann und darüber hinaus den hier genannten Punkten entspricht.

[2] Projekt-Webseite: `https://eclipse.org`

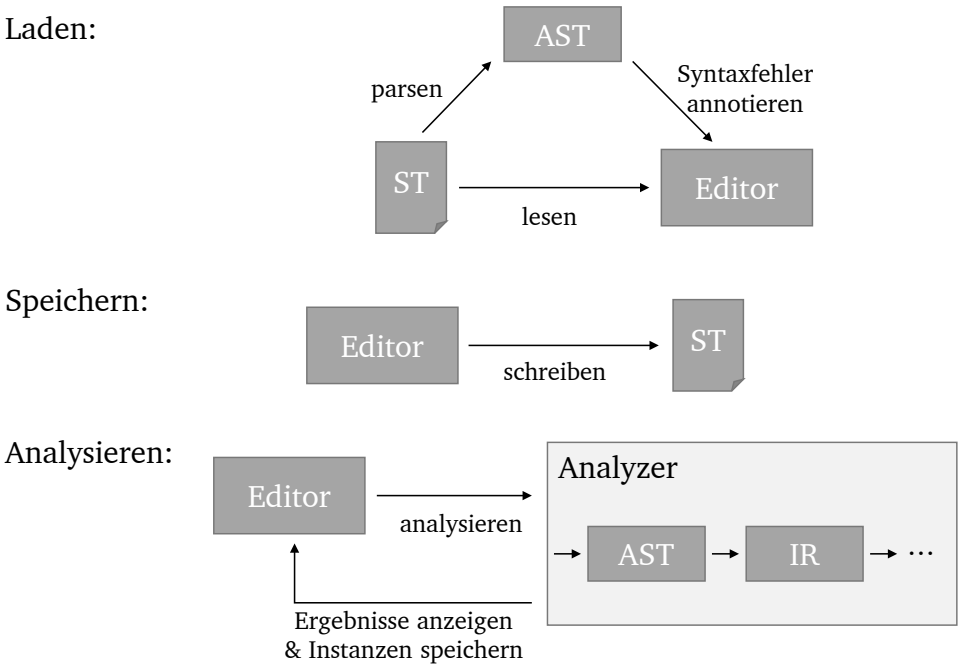


Abbildung 4.7: Informelle Darstellung der implementierten Funktionen des STEditors. Die Analyse wird direkt auf dem Quelltext des Editors durchgeführt.

Die Umsetzung nutzt dabei die Eclipse-Standard-Interfaces, um sich nahtlos in die Umgebung einzufügen und an Bedienelemente des einbettenden Programmfensters anzuknüpfen. Dazu gehört auch der Mechanismus zum Speichern geänderter Inhalte. Hierzu feuert der STEditor lediglich ein Event, um Änderungen im von ihm verwalteten Dokument anzuzeigen. Es ist dann möglich, diese Änderungen über übliche Tastenkombinationen oder die Symbolleiste zu speichern. Im Hintergrund arbeitet dafür ein sogenannter *Content-Provider*, dessen Aufgabe es ist, den Inhalt des Editors aus einer Datei bereitzustellen und in diesem Fall in eine Datei zu speichern. Diese Trennung in Editor und Content-Provider ist eine der wesentlichen Veränderungen, die dem STEditor ein reibungsloses Ändern und Speichern ermöglichen und im existierenden Editor zu Konflikten geführt hat.

Abbildung 4.7 zeigt die implementierten Funktionen des STEditors. Beim Öffnen eines ST-Programms wird dieses im neuen Programm in einen AST geparst, um Syntaxfehler direkt anzeigen zu können und Steuerelemente zu initialisieren. Außerdem wird der darin enthaltene Text in den Editor geladen. Die Definition zur Syntaxhervorhebung konnte aus dem bisherigen Editor übernommen werden und wird automatisch angewendet. Der AST wird an dieser Stelle nicht weiter gespeichert und zu einem späteren Zeitpunkt ggf. neu erstellt. Der Speichervorgang ist über den beschriebenen Content-Provider implementiert und speichert den editierten Quellcode im aktuellen Zustand.

Der Schritt der Analyse wurde im Vergleich zur bisherigen Arbeitsweise in ARCADE.PLC umstrukturiert. Alle Analysen und Zwischenprodukte werden von einem zentralen Objekt verwaltet, dem *Analyzer*. Zuerst wird das aktuell geschriebene Programm im Editor als Eingabe verwendet. Diese wird wie üblich in den AST geparst, bevor daraus die IR als Basis

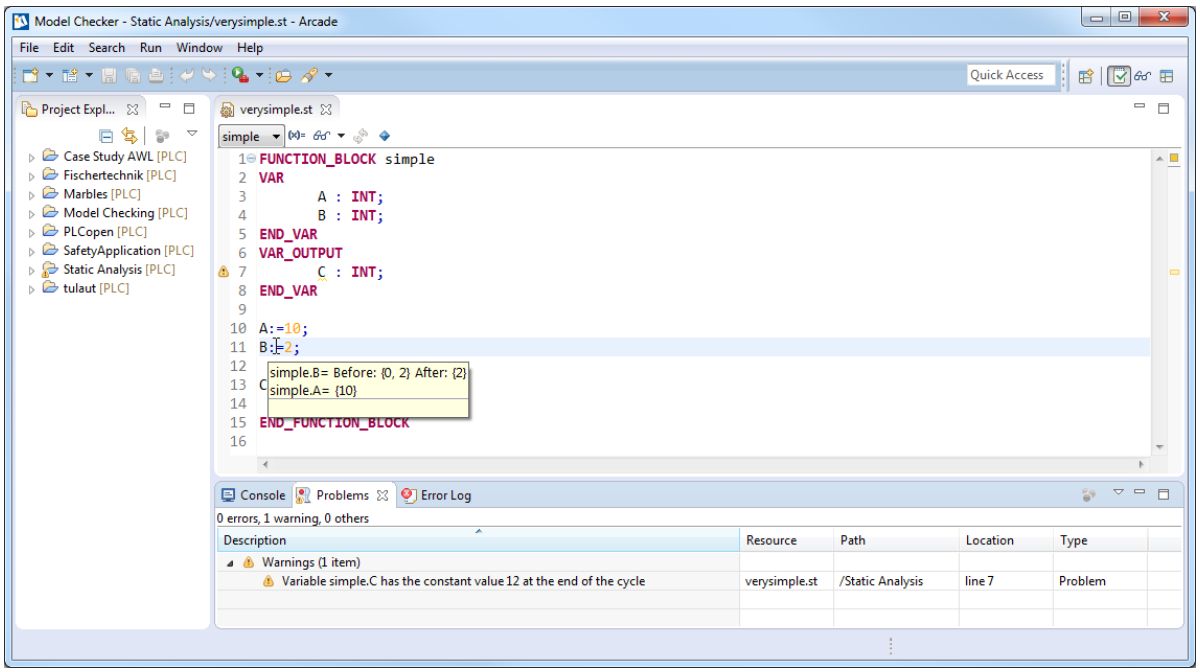


Abbildung 4.8: Der implementierte STEditor mit Darstellung einer Wertemengen-Annotation durch einen Mouseover in Zeile 11. Testweise werden alle verfügbaren Wertemengen angezeigt.

des CFAs generiert wird. Warnungen und Variablenwerte werden als Ergebnis zurück an den Editor übergeben. Außerdem wird die Instanz der Analyzer-Klasse dort gespeichert, um später darauf zurückgreifen zu können. Wird die Analyse nun wiederholt, kann das gesamte Analyzer-Objekt verworfen und mit dem aktuellen Quelltext erneut instantiiert werden. Mehrfache Verweise auf den Analyzer sind durch diese klar geregelte Objekt-Zugehörigkeit nicht mehr nötig und nicht mehr implementiert.

Der Editor ist damit bereit, Warnungen auf Basis eines aktuell geladenen und modifizierbaren Programms anzuzeigen. Diese werden in der für IDEs üblichen Weise dargestellt - mit einem Icon vor der Zeile, das auf den Fehler hinweist und einer geschlängelten Linie für die genaue Position des Fehlers im Quelltext.

Als letzte Funktion wurde die Annotation von Variablenwerten implementiert. Dies stellt im Gegensatz zu den Warnungen und Hinweisen für den Anwender eine optionale Information dar, die er bei Bedarf abrufen kann, durch sie jedoch nicht von der eigentlichen Arbeit abgelenkt werden soll. Auf die Markierung zur Darstellung der Wertemengen wurde daher verzichtet. Sie lassen sich bei Variablennamen durch das Überfahren mit der Maus anzeigen. Es erscheint ein kleines Fenster, das den Variablenwert der Variable an dieser Stelle ausgibt. Implementiert wurde diese Darstellung wie die Warnungen mithilfe von Eclipse-Boardmitteln für Annotationen im Quelltext.

Abbildung 4.8 zeigt den implementierten Editor. Zu Entwicklungs-Zwecken wurden in diesem Fall alle verfügbaren Informationen angezeigt, die in der Zeile verfügbar waren und nicht nur die Wertemengen, die konkret eine Variable betreffen. Im Beispielprogramm hat

Variable B in Zeile 11 vor der Zuweisung den Wert 0 oder 2, abhängig davon, ob die POE das erste oder zweite Mal aufgerufen wird. Nach dem Ausführen der Zuweisung hat sie den Wert 2.

Der Editor steht damit als Prototyp für die grafische Darstellung bereit. Weiterhin können mit diesem Editor stichprobenartig die Ergebnisse der Statischen Analyse überprüft werden. Auch Performancemessungen in folgenden Kapiteln wurden teilweise mit diesem Prototyp durchgeführt.

4.2.3 Umgang mit unvollständigen Programmen

Das Einsatzszenario des zu entwickelnden Unterstützungssystem sieht vor, dass Annotationen im Quelltext eines SPS-Programms bereits während dessen Entstehung und Fortentwicklung dem SPS-Programmierer helfen sollen. Daraus lassen sich die beiden Anforderungen A4 und A9 folgern, die eine kontinuierliche Analyse des Programms fordern. Weiterhin wird dadurch die Analyse auch auf nicht wohlgeformten Programmen gefordert, also bei solchen, die teilweise unfertige Ausdrücke oder Strukturen im Quelltext enthalten, da das Programm noch bearbeitet wird. Eine ähnliche Funktionalität ist bei vielen modernen Entwicklungsumgebungen sichtbar, die den Quellcode nicht erst beim Abspeichern einer Datei sondern mit der sogenannten *Continuous Compilation* fortlaufend während der Eingabe kompilieren und dabei gefundene Fehler im Quelltext annotieren. Aus diesem Anwendungsfall ist jedoch auch bekannt, dass eine noch nicht fertig geschriebene Zeile selbst oder umliegende Zeilen dadurch üblicherweise als fehlerhaft markiert werden.

```
5  IF B AND A THEN
6      D :=
7  ELSE
```

Abbildung 4.9: Syntaxfehler in Zeile 7 durch unvollständige Zeile 6

```
5  IF B AND A THEN
6      D := FALSE;
7  ELSE
```

Abbildung 4.10: Syntaxfehler aus Abbildung 4.9 durch vollständige Anweisung behoben

Abbildung 4.9 und 4.10 zeigen dafür ein Beispiel. Hier wurde Zeile 6 zunächst nur teilweise eingegeben. Es ist möglich, diese Eingabe fortzusetzen und damit zu einer syntaktisch korrekten Instruktion zu machen. Der automatische Compilierprozess versucht hier jedoch erfolglos, diese Zeile einzulesen und zu verarbeiten. Hinter dem Zuweisungsoperator wird etwas Anderes als das `ELSE`-Schlüsselwort erwartet, weshalb dieses hier als Fehler erkannt wird. Unerheblich ist dafür, dass der unvollständige Ausdruck an dieser Stelle noch zu einem Vollständigen ergänzt werden kann, wie Abbildung 4.10 zeigt. Zeile 6 entspricht nicht der erwarteten Form und führt dazu, dass der gesamte Quelltext so nicht eingelesen werden kann.

Die Erkennung und der Umgang von bzw. mit Syntaxfehlern ist unabhängig von der späteren Verwendung des ASTs, also ob dieser für Continuous Compilation oder in ARCADE.PLC zur Analyse des Programms genutzt werden soll. Ob und wie syntaktische Fehler im Programm behandelt werden können, wird von der Implementierung des Parsers bestimmt. Dieser liest ein Programm in seiner textuellen Form ein und überführen es in eine maschinenlesbare Form, mit deren Hilfe die Semantik des Programms ausgewertet werden kann. Parser sind in eine lexikalische und eine syntaktischen Analyse aufgeteilt [22]. Die lexikalische Analyse liest aus einem Programmtext, der als String oder Textdatei vorliegt, zusammenhängende Wörter, auch *Token* genannt. Ein Beispiel dafür sind Schlüsselwörter wie `IF`, Variablenbezeichner `var1` oder Abschlusszeichen wie `;`. Die so eingelesenen Token werden in der Syntaktischen Analyse auf ihre Anordnung hin überprüft, um daraus zusammenhängende Einheiten wie Instruktionen, Blöcke oder ganze Programme zu erhalten. Ein Beispiel für eine Instruktion wäre eine Zuweisung wie `var1 := 12;`. Welche Einheiten gebildet werden und wie diese zusammenhängen, definiert eine *Grammatik*, die dem Parser zugrunde liegt. Als Ergebnis liefert der Parser üblicherweise einen Baum, der die Struktur des Programms in Blöcken, Instruktionen und einzelnen Token widerspiegelt. Dieser Baum wird als Abstrakter Syntaxbaum bezeichnet (AST, *engl.*: abstract syntax tree). Der Parser ist dabei üblicherweise darauf angewiesen, dass ein Eingabeprogramm genau der Grammatik entspricht, um diese Umwandlung durchführen zu können. Entspricht ein Programm der zugrundeliegenden Grammatik, wird es auch als wohlgeformt bezeichnet.

Während der Eingabe eines Programms ist es häufig der Fall, dass der Quelltext nicht vollständig syntaktisch korrekt ist. Insbesondere an der aktuell bearbeiteten Stelle treten Syntaxfehler als Folge der zeichenweisen Eingabe auf. Der Grund für die Fehler ist für einen Parser jedoch nicht relevant. Für ihn ist es nicht unterscheidbar, ob ein Syntaxfehler aufgrund einer noch nicht vollständigen Eingabe entstanden ist, oder ob Eingabezeichen bzw. Token auf andere Weise zu viel oder zu wenig in der Eingabe stehen.

Ein Ansatz kann es daher sein, den Quellcode nur in dem Moment zu analysieren, in dem syntaktische Korrektheit vorliegt. Ein einfaches Kriterium zur Erkennung gibt es dafür jedoch nicht. Zwar sind in Structured Text die meisten Instruktionen durch ein Semikolon terminiert, einige der Kontrollstrukturen jedoch nicht. Auch bei Kontrollstrukturen wie `IF-THEN-ELSE`, hilft diese Erkennung nicht gegen fehlende Schlüsselwörter.

Timer kommen üblicherweise zum Einsatz, um bei Inaktivität des Programmierers das Programm zu parsen. Aber auch dieses Kriterium ist nicht genau genug, um ein wohlgeformtes Programm garantieren zu können. Sicher ist lediglich die regelmäßige Prüfung mithilfe eines Parsers, ob er erfolgreich einen AST generieren kann oder nicht.

Für die hier vorgestellten Prototypen werden Inaktivitäts-Timer verwendet, um den Quelltext nach deren Ablauf zu analysieren. Das Unterstützungssystem muss dann jedoch vorsehen, dass das Eingabeprogramm noch nicht syntaktisch korrekt ist. Dafür wurden im Rahmen dieser Arbeit zwei Vorgehensweisen untersucht:

1. Parser-Optimierungen
2. Puffern der Analyseergebnisse und Ergebnisdarstellungen auf Editor-Ebene

Der grundlegende Unterschied zwischen den beiden Ansätzen besteht darin, dass bei den Parser-Optimierungen versucht wird, auch solche Programme einzulesen, die nicht der IEC 61131-3 entsprechen und das Programm dann auf dieser Basis zu analysieren. Das Puffern der Analyseergebnisse speichert dagegen die zuletzt berechneten Ergebnisse. Schlägt eine Neuberechnung aufgrund von Syntaxfehlern fehl, werden die zuletzt erfolgreich berechneten Ergebnisse genutzt.

Parser-Optimierung

Bei den Parser-Optimierungen gibt es zunächst die Möglichkeit, auf den Parser-Prozess komplett zu verzichten, in dem der Texteditor durch einen projektionalen Editor ersetzt wird. Projektional bedeutet hier, dass statt eines Textdokuments direkt ein Syntaxbaum vom Programmierer geschrieben wird, der optisch dargestellt dem Quelltext der Sprache entspricht. Solche Editoren können zum Beispiel mithilfe des Meta-Programming-Systems (MPS) von Jetbrains[3] [9] erstellt werden. Im Rahmen der Abschlussarbeit [Wolf, 2015] wurde ein Structured-Text-Editor in MPS implementiert. Die Eingabe gleicht einem dynamischen Formular, in dem Felder mit Inhalt gefüllt werden können. Vorteil dieses Ansatzes ist, dass zu jedem Zeitpunkt ein AST existiert und noch nicht eingegebene Felder als solche deklariert sind. Diese leeren Felder entsprechen den Teilen eines Ausdrucks, die zur syntaktischen Korrektheit noch fehlen. Nachteilig ist jedoch die Bedienbarkeit des Editors, da Ausdrücke, Instruktionen und Kontrollstrukturen stets wohlgeformt eingegeben werden müssen. Eingaben sind damit nicht mehr an beliebiger Stelle des Programms möglich. Aus diesem Grund wurde der Ansatz nicht weiter verfolgt.

In konventionellen Parsern gibt es zwei Strategien für die Fehlerbehandlung [14]:

- Fehlererkennung bzw. den Panikmodus
- Fehlerkorrektur durch Ersetzen, Einfügen und Löschen von Token

Nutzt ein Parser einen Modus zur Fehler*erkennung* mit Panikmodus, so wird bei der syntaktischen Analyse ein Teil der eingelesenen Eingabe verworfen, bis ein Synchronisationstoken, auch *Beacon* genannt, gefunden wird. Eingelesen wird das Programm von Programmstart aus und ein typisches Beacon sind Schlüsselwörter oder Abschlusszeichen wie das Semikolon. In Abbildung 4.9 würde Zeile 6 komplett verworfen und eine Synchronisation mit der Eingabe würde ab dem Schlüsselwort `ELSE` erfolgen können.

Die Fehler*korrektur* kommt ebenfalls bei der syntaktischen Analyse zum Einsatz. Beim Erkennen eines Fehlers in der Eingabe werden mittels einer Heuristik Eingabetoken generiert, ersetzt oder gelöscht, um aus der Eingabe mit möglichst wenigen Modifikationen eine syntaktisch korrekte Eingabe zu erhalten. Dieses Verfahren ist aufwändig und kann dazu führen, dass sich aus der gefundenen, lokalen Korrektur im weiteren Programm Folgefehler ergeben. Diese Fehler werden auch *Spurious-Errors* genannt. Ein Parser-Generator mit üblicherweise wenigen Spurious-Errors ist AntLR[4]. Dieses vielseitige und in der Praxis oft verwendete Parser-Framework erlaubt auch die Erweiterung des generierten Parsers mit

[3] Webseite: `http://www.jetbrains.com/mps`
[4] Projektwebseite: `http://www.antlr.org/`

Java-Code. Für die vorliegende Arbeit wurde daher ein Structured-Text-Parser entwickelt, der auch unvollständige Programme verarbeiten kann. Nachteil dieses Parsers ist jedoch die Laufzeit im Fehlerfall. Ein Programm mit 100 Zeilen Code und einem Syntaxfehler im Quelltext benötigte bis zu 600 ms im AntLR-Parser statt 6 ms im Normalfall. Darüber hinaus kann es durch die Fehlerkorrektur zu bedeutenden Unterschieden im AST und damit in der eingelesenen Semantik des Programms, für die in der Statischen Analyse signifikant andere Ergebnisse berechnet werden würden, als das vom Programmierer beabsichtigte Programm.

Für das Unterstützungssystem wurde daher keiner dieser Ansätze verfolgt. Stattdessen wurde die zweite vorgeschlagene Vorgehensweise genutzt: Puffern der Analyseergebnisse und Ergebnisdarstellungen auf Editorebene.

Puffern der Analyseergebnisse

In ARCADE.PLC ist ein Parser für Structured Text integriert, der mit dem SableCC-Parser-Generator[5] erstellt wurde. Dieser Parser weist gute Laufzeiteigenschaften auf, bringt aber keine Fehlerbehandlungsmethoden mit. Bei einem syntaktisch nicht korrekten Programm wird direkt der vermutete Fehler gemeldet und ein AST wird nicht generiert.

Es wird zunächst das Szenario betrachtet, dass ein syntaktisch korrektes SPS-Programm am Ende um eine weitere Zeile ergänzt wird, bei der nicht sicher ist, ob sie wohlformuliert ist. Nach dieser letzten Zeile folgen keine weiteren Zeilen ausgenommen des Schlüsselwortes zum Abschließen der POE (`END_PROGRAM`, `END_FUNCTION`, ...).

Im Eclipse-basierten Editor wurden nun letztlich zwei Fälle vorgesehen: 1. Das Programm ist insgesamt syntaktisch korrekt und kann analysiert werden. 2. Das Programm ist nicht syntaktisch korrekt und es kann nicht durch ARCADE.PLC analysiert werden. Der erste Fall muss nicht weiter betrachtet werden, da er der normalen, beschriebenen Funktionalität entspricht. Es sei lediglich erwähnt, dass die Daten, mit deren Hilfe die Annotationen erstellt wurden, im Editor vorgehalten werden, bis die nächste Analyse sie überschreibt. Im zweiten Fall gibt es keine Analyseergebnisse und die Annotationen im Quelltext werden entsprechend weder gelöscht noch neu gebaut. Es bleiben die zuletzt generierten Warnungen und Hinweise sowie die annotierten Wertemengen sichtbar.

Die Betrachtung des Quelltextes erfolgt dann zeilenweise. Eine nicht geänderte und oder gelöschte Zeile wird vom Eclipse-Editor bereits korrekt behandelt. Annotationen werden beibehalten bzw. zusammen mit dem gelöschten Quelltext verschoben oder gelöscht. Für geänderte und neue Zeilen wird auf die Variablen und zugehörigen Variablenmengen der Vorgängerzeile zurückgegriffen. Deren Berechnung wurde in Abschnitt 4.2.1 beschrieben. Bezeichner der bekannten Veriablen werden in der Folgezeile mithilfe eines regulären Ausdrucks gesucht und es werden mit den Wertemengen zusammen Annotationen für die Zeile generiert. Enthielt die Vorgängerzeile eine Zuweisung, so wird entsprechend der Wert nach der Zuweisung verwendet. Die Werte der anderen Variablen können sich nicht geändert haben und sie werden daher annotiert wie in den Zeilen zuvor.

[5] Projektwebseite: `https://sablecc.org`

Mit diesem Vorgehen ist es nicht notwendig, neue und geänderte Zeilen mithilfe eines Parsers einzulesen oder das komplette Programm erneut zu analysieren. Zunächst nicht auswertbar ist dabei der Einfluss, den die Zeile auf Variablen oder den Rest des Programms haben kann. So bleiben Aufrufe von POEs und Funktionen ebenso ohne Effekt wie Zuweisungen, deren linke Seite der Zuweisung so nicht berechnet werden kann. Da Variablen abhängig von ihrem Typ ihren Wert auch im nächsten Aufruf der POE behalten können, hat auch eine Zuweisung am Ende der POE möglicherweise einen Einfluss auf beliebige andere Variablen im nächsten Aufruf. Bis die Zeile am Ende des Programms jedoch zur syntaktischen Korrektheit vervollständigt wurde und somit eine erneute Analyse möglich ist, kann ohnehin keine klare Aussage über den Effekt dieser Zeile getroffen werden. Dieses Verhalten wird daher als unproblematisch für das Unterstützungssystem angesehen.

Wird nun das Szenario nicht länger auf die letzte Zeile der aktuellen POE beschränkt sondern in Betracht gezogen, dass eine Zeile an beliebiger Stelle im Programm geändert werden kann, so kann festgestellt werden, dass lediglich die Erkennung der geänderten oder neuen Zeile hinzu kommt. Da auch zuvor von einem Einfluss der letzten Zeile auf den Rest des Programms ausgegangen werden musste und es vertretbar ist, die angezeigten Wertemengen anderer Zeilen erst wieder im Zustand der syntaktischen Korrektheit zu aktualisieren, ändert sich in diesem erweiterten Szenario für die Annotation neuer und geänderter Zeilen nichts weiter. Die Detektion geänderter Zeilen kann über den Eclipse-Editor erfolgen. Eclipse bietet für den Benutzer im Editor bereits eine farbliche Markierung für geänderte oder neue Zeilen an. Auf diese Informationen kann auch das Unterstützungssystem zugreifen, um damit die Annotation noch nicht neu-analysierter Zeilen zu steuern.

Da in den beschriebenen Szenarien davon ausgegangen werden kann, dass betreffende Zeilen in naher Zukunft noch geändert bzw. vervollständigt werden, kann auf diese Weise der Aufwand eines kompletten Analysevorgangs für eine vorläufige Version der Zeile eingespart werden. Wertemengen lassen sich so trotzdem bereits annotieren. Lediglich Hinweise und Warnungen von ARCADE.PLC können auf diese Weise für unvollständige Zeilen nicht erzeugt werden.

Die Anforderung 9, also der adäquate Umgang mit syntaktisch inkorrekten Eingaben, wird damit durch den Editor statt vom Back-End übernommen.

4.2.4 Erweiterung der JSON-Schnittstelle

Statische Analyse in ARCADE.PLC kann sowohl mithilfe der grafischen Oberfläche als auch über eine Webschnittstelle genutzt werden. Letztere ist ursprünglich integriert worden, um Structured-Text-Programme über eine Online-Demonstration[6] entgegenzunehmen und dort in einem JavaScript-basierten Editor annotieren zu können. Sie liefert die gleichen Warnungen und Hinweise zu einem SPS-Programm, die auch in der Eclipse-GUI angezeigt werden. Ausgeführt als Server-Anwendung wird ARCADE.PLC auch als *Arcade-Server* bezeichnet. Für die Anbindung an andere Anwendungen eignet sich eine Webschnittstelle bzw. einen HTTP-Socket sehr gut, da Bibliotheken für die Interaktion mit Sockets in allen

[6] Webseite: `https://arcade.embedded.rwth-aachen.de/go`

üblichen Programmiersprachen vorhanden sind und so eine universelle Verwendbarkeit gegeben ist. Die Bezeichnung *Web*schnittstelle bezieht sich dabei nicht darauf, dass diese Anwendung über das Internet angeboten oder genutzt werden muss. Server und Client können hier auch auf dem selben Computer operieren.

Die Schnittstelle verwendet ein einfaches, auf HTTP-basierendes Protokoll, um Daten auszutauschen. Auf Port 5000 werden über einen `POST`-Request die Parameter *Dialekt*, *Programm* sowie *Ziel-IP* und *Client-Programm* entgegengenommen. Die letzten beiden Parameter dienen der Abwicklung des Verbindungsaufbaus. Die ersten beiden werden direkt an die Statische Analyse durch ARCADE.PLC weiter gegeben. Wie in der grafischen Oberfläche können auch über diese Schnittstelle mehrere POEs gleichzeitig zur Analyse übermittelt werden.

Nachdem das Programm auf diese Weise übertragen und die Statische Analyse darauf ausgeführt wurde, wird vom ARCADE.PLC-Server eine Antwort in Form der *Java Script Object Notation*, kurz JSON, generiert und an den Client zurück gesendet. Das Format entspricht dabei dem RFC-Standard-Entwurf 4627 [13]. Diese Antwort enthielt bisher die folgenden Informationen als JSON-Objekte in einem JSON-Array:

- Nachrichtentext zur Warnung bzw. zum Fehler
- Position der Nachricht im Quelltext mit Zeilen- und Zeichenindex
- Schweregrad der Nachricht (Warnung oder Hinweis)

Für ein Unterstützungssystem mit Anzeige der Wertemengen-Informationen, wie es in Anforderung 2 festgelegt wurde, müssen nun auch diese Informationen über die Webschnittstelle bereitgestellt werden. Dafür wurde die Nachrichtenübermittlung bzw. die Zusammenstellung der Informationen im Rahmen dieser Arbeit erweitert. Neben dem übertragenen Array für Warnungen und Hinweise wird ein JSON-Array für diese Wertemengen-Informationen für jede Zeile angelegt. Die JSON-Objekte enthalten dann die folgenden Informationen:

- Bezeichner inkl. Namensraumangabe der Variable
- Zeilenangabe, für die die Werte gelten
- Wert vor Ausführung der Zeile, der *rValue*
- Wert nach Ausführung der Zeile, der *lValue*

Beim Bezeichner ist der Namensraum der Variable wichtig, wenn mehrere POEs untersucht werden. So ist es möglich, dass eine sonst gleichnamige Variable aus einer anderen POE hier bekannt ist und annotiert wird. Es ist möglich, hier Variablenwerte für Zeilen zu übertragen, die in dieser Zeile nur bekannt sind, jedoch nicht verwendet werden. Diese Informationen werden dann wichtig, wenn eine Folgezeile annotiert werden soll, bevor eine erneute Analyse erfolgreich sein kann (vgl. Abschnitt 4.2.3). Ein Zeichenindex ist nicht nötig, da die Annotation auf eine konkrete Variable in einem späteren Schritt im Editor erfolgen kann. Die Unterscheidung zwischen linker und rechter Seite einer Zuweisung

```
{
   "Problems":[
      {
         "Type":"STATIC_ANALYSIS",
         "Message":"Variable Presentation.Z has the constant value 3 at
             the end of the cycle",
         "SourcePosition":{
            "BeginLine":7,
            "EndLine":7,
            "EndColumn":9,
            "BeginColumn":8
         },
         "Severity":"WARNING"
      }
   ],
   "Values":[
      {
         "lValue":"{}",
         "identifier":"Presentation.X",
         "line":11,
         "rValue":"{1}"
      },
      {
         "lValue":"{}",
         "identifier":"Presentation.Y",
         "line":11,
         "rValue":"{2}"
      },
      {
         "lValue":"{3}",
         "identifier":"Presentation.Z",
         "line":11,
         "rValue":"{0, 3}"
      }
   ],
   "EntryPoints":[
      "FUNCTION_BLOCK Presentation"
   ]
}
```

Abbildung 4.11: Gekürztes Beispiel einer JSON-Antwort vom ARCADE.PLC-Server mit einer Warnung und drei Wertemengen-Informationen

erfolgt über die *rValue*- und *lValue*-Angabe. Darin werden Wertemengen übertragen, die entsprechend vor bzw. nach dem Ausführen der Zeile aktuell sind. Die Formatierung der Wertemengen hängt von dem gespeicherten Wert ab. Wenige Werte werden als Menge mit einzelnen Elementen übertragen, darüber hinaus gibt es Wertemengen mit Minimum und Maximum, Bitvektor-Angaben und einem Stern für den Maximalwert, also den kompletten Wertebereich der Variable. Sind keine Informationen auf lValue-Seite verfügbar, bleibt die entsprechende Angabe im Objekt leer.

In Abbildung 4.11 ist exemplarisch die JSON-Antwort für das Codebeispiel aus Abbildung 4.1 dargestellt. Darin enthalten ist die Warnung, dass Variable `Z` nach Zyklusende immer den gleichen Wert besitzt. Darüber hinaus werden die Werte der Variablen `X`, `Y` und `Z` in Zeile 11 übertragen. Weitere Wertemengen für andere Zeilen wurden aus Platzgründen entfernt.

Mit dieser Erweiterung der JSON-Webschnittstelle wurde die Grundlage gelegt, auch anderen Editoren abseits der Eclipse-GUI neben den Warnungen und Hinweisen auch berechnete Wertemengen zur Verfügung zu stellen. Anforderung 7 ist damit erfüllt.

4.3 Codesys-Plugin

Anforderung 10 fordert die Darstellung der Informationen aus der Statischen Analyse in einer üblichen Entwicklungsumgebung für SPS-Code. Diese Anbindung ist von großer Bedeutung, um Hypothese 1 zu untersuchen, also ob die Informationen einem SPS-Programmierer beim Schreiben eines Programms helfen. Dafür wird die übliche Arbeitsumgebung benötigt, um sowohl externe Faktoren wie eine andere Gestaltung der grafischen Oberfläche auszuschließen als auch eine Kompilier- und Ausführungsumgebung bieten zu können, sodass das Programm auch auf konventionelle Weise getestet werden kann.

Für die vorliegende Arbeit wurde Codesys 3.5 als Entwicklungsumgebung ausgewählt. Das dafür notwendige SDK wurde freundlicherweise vom Hersteller 3S-Smart Software Solutions zur Verfügung gestellt. Darin enthalten ist auch die Beschreibung der API, um Erweiterungen gemäß der vorgesehenen Plugin-Infrastruktur strukturieren zu können.

Dieser Abschnitt wird sich mit der Architektur und Implementierung des Codesys-Plugins befassen. Es ist in zwei logische Teile unterteilt: Den Editor und die Anbindung an ARCADE.PLC mithilfe eines Websockets. Auf die Implementierung des Editors, die für diese Arbeit vorgenommen wurde, wird Abschnitt 4.3.2 eingehen. Die Anbindung des Editors an ARCADE war Teil der Abschlussarbeit von [Müllers, 2018] und wird in Abschnitt 4.3.1 beschrieben.

4.3.1 Architektur

ARCADE.PLC soll als Back-End für das Unterstützungssystem dienen und wurde gemäß der Anforderungen auf die Verwendung dafür vorbereitet. Eine direkte Integration des ARCADE-Quellcodes in ein Plugin für Codesys ist jedoch nicht möglich. Codesys wurde mithilfe des Microsoft .NET-Frameworks entwickelt und beschreibt daher das Schreiben von Plugins, die ebenfalls dieses Framework verwenden. ARCADE.PLC als Java-basierte Anwendung

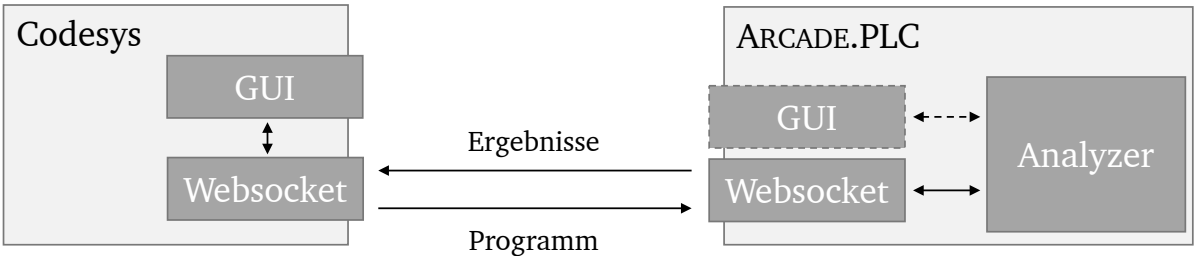


Abbildung 4.12: Die Kommunikations-Architektur zum Austausch zwischen ARCADE.PLC und Codesys. Die grafische Oberfläche auf ARCADE-Seite wird in diesem Fall nicht genutzt.

kann hier nicht ohne weiteres eingebunden werden, weshalb eine andere Lösung gefunden wurde, die eine Unabhängigkeit der beiden Programme garantiert und trotzdem einen einfachen und flexiblen Austausch von Programmcode und Analyse-Informationen erlaubt.

In Arcade steht als Interface neben der Eclipse-basierten grafischen Oberfläche auch eine Kommandozeilen- und eine Websocket-Schnittstelle zur Verfügung. Diese Schnittstelle wurde in Abschnitt 4.2.4 bereits dahingehend erweitert, dass auch Wertemengen übertragen werden können. Für die Integration der Analyseergebnisse wird daher ein Websocket-Protokoll verwendet. Dieses erlaubt die getrennte Entwicklung von ARCADE von einem Codesys-Plugin. Auch wäre so ein Aufbau möglich, bei dem die ARCADE-Instanz auf einem entfernten Server zur Verfügung steht und unabhängig von den Ressourcen des Anwender-PCs die Analysen durchführt. Von dieser Möglichkeit wurde im Rahmen dieser Arbeit jedoch kein Gebrauch gemacht.

Abbildung 4.12 zeigt die verwendete Architektur. ARCADE bietet wie beschrieben zwei Komponenten zur Nutzung der Analyse: Die grafische Benutzeroberfläche (GUI) in Form des Editors und einer Komponente für die Websocket-Kommunikation. Genutzt wird in diesem Szenario nur die Verbindung über den Websocket, für die ARCADE.PLC als Server dient. Das so empfangene Programm wird wie auch bei der Nutzung der grafischen Oberfläche in einer Analyzer-Instanz analysiert und die Ergebnisse über die bestehende Verbindung im JSON-Format zurück geschickt. Dabei werden Variableninformationen sowie die Warnungen und Hinweise jeweils mit ihren Positionen im Quellcode angegeben. Dieses in Abschnitt 4.2.4 beschriebene Format wird durch ein Codesys-Plugin geparst werden, um dann in einem Editor der Entwicklungsumgebung im Code annotiert zu werden.

Für die Planung der Implementierung auf Codesys-Seite muss das Plugin-System näher betrachtet werden. Der Hersteller bietet über eine Schnittstelle die Möglichkeit, einige gekapselte Komponenten in Codesys hinzuzufügen oder vorhandene Funktionen zu erweitern. Für eine auch zukünftig angestrebte Versionskompatibilität bei Änderung interner Komponenten ist die Einflussnahme auf diese herstellerseitig angebotenen Komponenten nur begrenzt möglich. Dieses Prinzip des Information-Hidings ist für die Anwendungsfälle des Herstellers sicherlich sinnvoll, erschwert jedoch die für diese Arbeit beabsichtigten Erweiterungen. So ist es zwar möglich, eigene Editor-Darstellungen für Projekt-Bestandteile wie POEs zu erstellen, eine direkte Einflussnahme auf Abläufe im vorgegebenen ST-Code-Editor war jedoch nicht möglich. Der folgende Abschnitt 4.3.2 wird darauf eingehen, welche

Einschränkungen es genau gibt und wie schließlich eine eigene Darstellung für den Editor integriert wurde.

Einen Ausschnitt der Softwarearchitektur in Codesys zusammen mit den für diese Arbeit entwickelten Erweiterungen bietet Abbildung 4.13. Codesys gliedert sich in die drei Bestandteile: *Systemkomponenten*, *gemeinsame Komponenten* und *Plugins*. Systemkomponenten stellen den Kern der Codesys-Anwendung dar und sind von elementarer Bedeutung für das Laden von Projektdateien und Sprachelementen. Der Component-Manager als Teil davon ist darüber hinaus für das Laden der Plugins zuständig. Für diese Arbeit von Bedeutung sind weiterhin der Message-Store, der Fehlermeldungen und Warnungen programmweit verwaltet. Diese Komponente wird später als Schnittstelle genutzt, um Meldungen aus ARCADE.PLC einzupflegen und in allen Bestandteilen des Programms verfügbar zu halten. Der Object-Manager ist für alle Objekt-Instanzen zur Laufzeit zuständig. Über ihn ist unter anderem der Zugriff auf geschriebenen Programmcode möglich. Dieser Programmcode wird von Codesys eigenständig compiliert, was mithilfe des Language-Model-Managers geschieht. Der Option Storage übernimmt die Speicherung von Einstellungen, was bei der Darstellung des Editors genutzt wird. Zuletzt ist die Komponente Engine für einen thread-sicheren Zugriff auf die verfügbaren Daten wichtig, da Codesys einige der internen Datenstrukturen in einem Haupt-Thread verwaltet, auf den über diese definierte Schnittstelle zugegriffen werden kann.

Der in Abbildung 4.13 dargestellte Bereich *Plugins* zeigt nur einen Teil der verfügbaren Plugins. Viele der Codesys-eigenen Komponenten wie Editoren, Visualisierungen und Code-Generatoren sind ebenfalls als Plugins angebunden. Für die Interaktion mit ihnen sind Interfaces vorgesehen, was bei Kenntnis des Interfaces auch eine Nutzung durch andere Plugins ermöglicht. Dem beschriebenen Ansatz des Information-Hidings nach sind die Möglichkeiten über die Interfaces jedoch üblicherweise auf eine Zusammenarbeit mit den herstellereigenen Komponenten ausgelegt und sind oft wenig dokumentiert.

InjEditor und *Arcade3S* sind die beiden Plugins, die im Folgenden genauer beschrieben werden. Sie implementieren die Interfaces für einen Editor bzw. für eine Kommando, um vom Component-Manager geeignet geladen zu werden. Die gemeinsamen Komponenten spielen für die vorliegende Arbeit keine zentrale Rolle.

Die Integration in Codesys wurde also in Form von zwei Plugins realisiert: Dem Editor und einer Kommunikationsschnittstelle. Diese Trennung wurde vorgenommen, da die Plugins auch für sich arbeiten können sollten. Der implementierte Editor ist so unabhängig von der Versorgung mit Informationen aus der Statischen Analyse und das Kommunikationsmodul kann je nach Bedarf ein- oder ausgeschaltet werden. Weiterhin war es so möglich, die Alternative des original-Editors parallel nutzen zu können und auch dort Meldungen der Statischen Analyse einzublenden. Diese Möglichkeit unterliegt jedoch Einschränkungen, die im Folgenden diskutiert werden. Für die Evaluation in Abschnitt 5 wurde ausschließlich der hier vorgestellte Editor unter dem Namen InjEditor verwendet. Beide Plugins wurden in C# mithilfe von Microsoft .NET und in Visual Studio 2015 entwickelt.

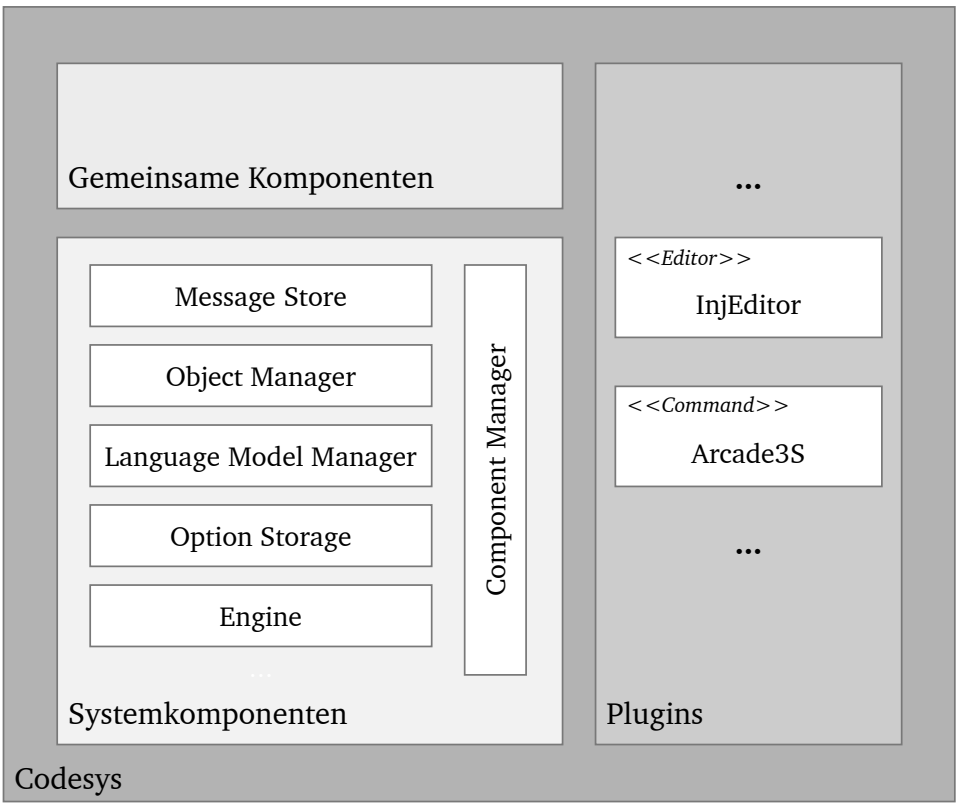


Abbildung 4.13: Ausschnitt aus der Architektur von Codesys. Die exemplarisch dargestellten Plugins sind Teil der hier vorgestellten Implementierung.
(Abbildung erstellt nach Informationen aus [46])

4.3.2 Editor

Der Structured-Text-Quellcode muss die Warnungen und Hinweise aus der Statischen Analyse darstellen können. Weiterhin muss es eine Möglichkeit geben, die Wertemengen aus der Statischen Analyse im Code anzeigen zu können. Diese Anforderungen wurden in Abschnitt 3.3 als Anforderungen 11 und 12 festgelegt. Der Codesys-eigene Editor trägt den internen Namen *SynEd* und beherrscht bereits Annotationen im Quelltext. Er wurde im Rahmen dieser Arbeit auf seine Eignung für die Evaluation untersucht, letztlich musste jedoch ein eigener Editor entwickelt werden. Die Vor- und Nachteile dieser Entscheidung werden im Folgenden dargelegt.

SynEd bezieht diese Annotationen aus der zentralen System-Komponente, dem Message-Store. Darin werden beispielsweise Compiler-Fehler gemeldet, die über diese Komponente an den Editor übergeben und dort annotiert werden. Diese Meldungen des Message-Stores werden üblicherweise auch noch in einem Fenster unterhalb des Programm-Editors tabellarisch dargestellt. Aber auch eigene Meldungen können durch Plugins in diesen Message-Store eingefügt werden. Auch andere Schweregrade der Meldungen sind möglich. So gibt es neben den genannten *Fehlern* auch die Klassifikationen *Warnung* und *Information*. Fehler werden von SynEd im Quellcode mit einer roten, gezackten Linie versehen, Warnungen

erhalten eine solche gelbe Linie. Beim Überfahren mit der Maus erscheint ein Hinweistext in Form eines Popups, mit der Meldung, die zu dieser Codestelle annotiert wurde. Meldungen der Klasse Informationen werden dagegen gar nicht annotiert und es erscheinen keine Popups mit dem enthaltenen Text. Sie erscheinen lediglich als Eintrag in dem Fenster unterhalb des Editors.

Für diese Arbeit wurde untersucht, ob der Editor auf nicht weiter dokumentierte Weise erweitert werden kann, um auch die Informations-Klasse darstellen zu können. Die Mouseover-Aktion des Editors löst beim Überfahren von Variablen-Identifiern bereits eine Aktion aus und zeigt den Datentyp der Variable. Es war nicht möglich, dieses Verhalten ohne Zugriff auf den Quellcode von Codesys zu ändern. Auch zusätzliche Annotations-Zeilen zwischen den Code-Zeilen, wie sie von Krämer et al. [27] verwendet wurden, ließen sich über die verfügbaren Schnittstellen nicht integrieren.

Die Annotationen für Warnungen und Fehler wären für den geforderten Anwendungsfall und Anforderung 11 gut geeignet. Aufgrund der nicht vorhandenen Erweiterbarkeit der Darstellung in SynEd, um Wertemengen an Variablen zu annotieren, scheidet dieser Editor für den vorgesehenen Anwendungsfall aus.

Es wurde daher die Entscheidung getroffen, einen neuen Editor für die Darstellung in Codesys zu entwickeln, der flexibel anpassbar ist und die Anforderungen erfüllt. Das Vorbild war dabei der Editor SynEd, um bei einer Nutzerstudie Effekte durch eine Eingewöhnungs- oder Umgewöhnungsphase zu vermeiden. Für den neuen Editor galten die folgenden Anforderungen:

- Der Editor soll visuell dem Editor SynEd ähnlich sein.
- Alle Warnungen und Hinweise aus dem Message-Store müssen angezeigt werden
- Wertemengen für Variablen müssen annotierbar sein
- Der Quellcode muss mit den Codesys-Systemkomponenten synchronisiert werden

Die ersten drei Punkte ergeben sich bereits aus der vorangegangenen Beschreibung benötigter Funktionen. Der letzte Punkt ist wichtig, um Quellcode aus der aktuell geladenen Projektdatei von Codesys laden zu können und darüber hinaus die Anbindung an den Compilier-Prozess zu erhalten. Nur so kann das Programm durch Codesys auch kompiliert und in Codesys simuliert werden, um einen realistischen Anwendungsfall für die Unterstützungsfunktionen durch die Statische Analyse schaffen zu können.

Der neu entwickelte Editor wurde *Injected Editor* (kurz *InjEditor*) genannt, da er sich nahtlos in die Umgebung in Codesys einfügen soll. Als Basis für diesen Editor wurde das Projekt AvalonEdit[7] verwendet. Dieses Projekt steht unter einer Open-Source-Lizenz, der MIT-Lizenz[8], und steht daher im komplett in C# geschriebenen Quellcode zur Verfügung. Ursprünglich entwickelt für die Entwicklungsumgebung SharpDevelop[9] ist AvalonEdit dennoch modular aufgebaut und kann somit an andere Entwicklungssprachen angepasst

[7] Projekt-Webseite: `http://avalonedit.net`
[8] Lizenzvereinbarung: `http://opensource.org/licenses/MIT`
[9] Projektwebseite: `http://www.icsharpcode.net/OpenSource/SD/Default.aspx`

werden. Es bietet einen Texteditor mit der Unterstützung von Syntaxhighlighting, Code-Faltung und alle üblichen Editor-Funktionen inklusive des Renderings.

AvalonEdit baut auf das Microsoft Grafik-Framework Windows Presentation Foundation (WPF) auf, welches seit Windows Vista die Standard-Methode für grafische Oberflächen unter Windows-Betriebssystemen ist. Für Codesys, das noch auf das Windows-Forms-Framework aufbaut, wurde dafür eine Adapter-Klasse implementiert, die jeweils zwischen beiden Frameworks vermittelt.

Der Editor selbst stand damit bereits zur Verfügung und musste noch an die Anforderungen und an die Umgebung in Codesys angepasst werden. Dafür wurden die folgenden Funktionen umgesetzt:

ST-Unterstützung Hierfür wurde ein Syntaxhighlighting konfiguriert, das die Ausdrücke und Datenformate in Structured Text hervorheben kann. Grundlage dafür sind reguläre Ausdrücke für Schlüsselwörter.

Deklarations-Teil Der Codesys-Editor SynEd teilt die Ansicht in einen Bereich für die Variablendeklaration und den Code-Teil. Diese Trennung wurde auch im InjEditor umgesetzt.

Schriftarten Für ein einheitliches Erscheinungs-Bild wurde eine Anbindung an den Option-Storage integriert. So werden die Größen- und Stileinstellungen vom original-Editor übernommen und sind gleichzeitig über das Einstellungsmenü von Codesys konfigurierbar.

Code-Faltung Hiermit ist es möglich, Code-Blöcke vorübergehend einzuklappen, um die Darstellung übersichtlicher zu machen. Dieses Feature ist auch im SynEd-Editor vorhanden. Ein Beispiel für einen Codeblock sind `FOR`-Schleifen oder Bedingungen mit `IF`

Warnungs-Marker Wird vom Message-Store eine Warnung oder ein Fehler gemeldet, so wird der entsprechende Bereich mit einer gelben bzw. roten gezackten Linie unterstrichen. Zusätzlich erscheint ein Warnungs-Symbol in Form eines entsprechend gefärbten Kreises am Rand zwischen Quelltext und Zeilennummer, um auf den Fehler in der Zeile aufmerksam zu machen. Diese Marker am Rand des Codes waren in AvalonEdit nicht vorgesehen, ließen sich jedoch einfach integrieren.

Variablenwerte Im Editor ist es möglich, Informationen für Codestellen zu hinterlegen, ohne diese weiter optisch zu annotieren. Genutzt wird dies, um die Wertemengen von Variablen zu annotieren.

Editor-Operationen Events wie Kopieren und Einfügen sowie Undo&Redo-Operationen, also Bearbeitungen im Editor zurücknehmen oder wiederholen, werden an Codesys weitergegeben. Die Umgebende IDE besitzt für diese Operationen Schaltflächen und Tastenkombinationen, die sich so auch im InjEditor intuitiv benutzen lassen.

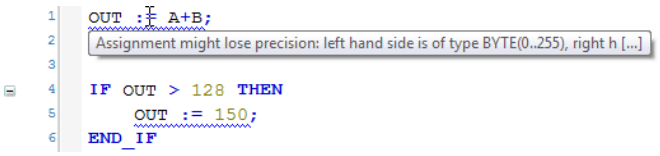


Abbildung 4.14: Original-Editor SynEd von Codesys mit einer Annotation

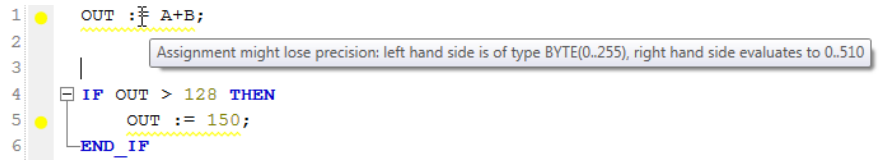


Abbildung 4.15: Entwickelter Editor InjEditor mit der gleichen Annotation

Datenmodell Der Quellcode für den Editor wird zu Beginn aus dem Object-Manager von Codesys geladen. Hier befinden sich alle Bestandteile des geladenen Projekts und so auch die Deklarations- und Programmcode-Texte für den Editor. Über Events werden Änderungen daran an beteiligte GUI-Komponenten propagiert. Der Programmcode wird hier nach einer Inaktivität von 0,5 Sekunden hin übertragen, um über den Language-Model-Manager das Kompilieren des Programms zu erreichen. Dieses Verhalten wurde an das des Codesys-eigenen Editors angelehnt.

Die Features wurden alle in der Implementierung des InjEditors umgesetzt. Erwähnenswert ist hier noch einmal die Trennung der Quellcode-Teile für Variablen und Programmcode. Diese Trennung ist in vielen Entwicklungsumgebungen für SPSen üblich, obwohl sie vom Standard IEC 61131-3 [16] nicht vorgesehen ist. Da ARCADE diese Trennung nicht vorsieht, müssen beide POE-Bestandteile vor der Analyse zunächst zusammengeführt werden, was später noch in Abschnitt 4.3.3 beschrieben wird. Für die getrennten Code-Ansichten wurden durch den InjEditor zwei AvalonEditor-Instanzen erzeugt, die sich den verfügbaren Platz durch eine verschiebbare Trennlinie aufteilen. Die Zeilennummerierung beginnt in jedem Fenster analog zu SynEd von 1 an.

Die Variablenwerte können im InjEditor nun angezeigt werden, indem mit der Maus über einen Variablenbezeichner gefahren wird. Diese Darstellungsform wurde gewählt, da die ständige, visuell sichtbare Darstellung aller berechneten Wertemengen überaus viel Darstellungsfläche benötigt und die Übersichtlichkeit für den Programmierer so verloren gehen kann. Eine Markierung der Variablen durch gezackte Linien ist ebenfalls nicht sinnvoll, da so jeder Variablenbezeichner markiert würde. Ohne optische Marker setzt die gewählte Lösung voraus, dass der Programmierer die Funktion kennt. Sie erlaubt es ihm dann jedoch, sie während seiner Tätigkeiten nach Bedarf zu nutzen und beeinträchtigt in der übrigen Zeit nicht die gewohnte Übersicht im Codeeditor.

In Abbildung 4.14 und 4.15 sind beide Editoren im Vergleich dargestellt. Die reine Darstellung des Quelltextes ist in beiden Editoren gleich. Sie unterscheiden sich in der Größe der Zeilennummerierung und der Position der Code-Faltungs-Bedienelemente. Auch fällt auf, dass SynEd für Warnungen, wie sie in Zeile 1 und 5 annotiert sind, eine andere Farbe und keinen Zeilenmarker auf der linken Seite einsetzt. Diese Unterscheidung wurde

bewusst gewählt, um die Aufmerksamkeit des Programmierers mehr auf die bereitgestellten Warnungen und Hinweise lenken zu können. Bei den anderen minimalen Unterschieden wird davon ausgegangen, dass sie nicht für eine Umgewöhnung des Programmierers sorgen werden.

In Abschnitt 4.2.3 wurde beschrieben, wie der Eclipse-Prototyp des Editors mit syntaktisch inkorrekten Programmen umgeht. Dieser Ansatz wurde auch im InjEditor gewählt. Aktualisierter Quellcode wird vom Editor nach 0,5 Sekunden Inaktivität an den Object-Manager weitergegeben. Von dort aus haben das Plugin zur Anbindung an ARCADE und der Language-Model-Manager Zugriff auf den Quellcode. Der Language-Model-Manager versucht den Quelltext zu kompilieren und gibt über den Message-Store Fehler aus, wenn syntaktische Fehler vorliegen. Andere Meldungen, die bis dahin über die aktive POE im Message-Store vorhanden waren, werden zu diesem Zeitpunkt gelöscht. Dies gilt ebenso für Meldungen, die von ARCADE generiert und dort eingebracht werden. Für annotierte Variablenwerte ist dies jedoch keine adäquate Lösung, da bei einem gelöschten Message-Store auch alle annotierten Wertemengen entfernt wurden. Es wurden daher zwei Funktionen eingebaut, die den Umgang mit syntaktisch nicht korrektem Code erleichtern:

1. Annotierte Wertemengen bleiben im InjEditor gespeichert, auch wenn der Message-Store geleert wird. Sie werden erst aktualisiert, wenn ARCADE.PLC neue Werte geliefert hat und nicht selbst Syntaxfehler meldet.

2. Neue Zeilen werden vom InjEditor ebenfalls Annotiert. Hierzu wird auf die berechneten Ergebnisse der vorausgehenden Zeile zugegriffen.

Das Verfahren zur Annotation neuer und syntaktisch nicht korrekter Zeilen funktioniert analog zur Implementierung im Eclipse Editor, die in Abschnitt 4.2.3 beschrieben wurde. Die zuletzt berechneten Wertemengen bleiben dazu gespeichert und werden erst beim nächsten erfolgreichen Compiliervorgang von ARCADE überschrieben. Wird nun eine neue, nicht analysierte Zeile hinzugefügt, wird auf die berechneten Werte der vorausgehenden Zeile zugegriffen. Werden mithilfe eines regulären Ausdrucks Variablenbezeichner in der neuen Zeile gefunden, deren Werte bekannt sind, so werden diese im Editor annotiert.

Im Rahmen der Abschlussarbeit [Müllers, 2018] wurde neben den hier genannten Erweiterungen in den Editor noch ein Logger integriert. Dieser ist in der Lage, Benutzereingaben mit Zeitstempeln zu protokollieren. Weiterhin wird protokolliert, wenn der SPS-Programmierer eine Mouseover-Aktion ausführt und sich so eine Warnung im Code anzeigen lässt.

Neben den implementierten Funktionen gibt es auch Einschränkungen, die der InjEditor im Gegensatz zum vorgegebenen Editor SynEd nicht unterstützt. So gibt es kein Kontextmenü, über das Refactoring-Operationen ausgeführt werden können. In der Evaluation sind jedoch keine Aktionen aus dem Kontextmenü notwendig, weshalb dieser Punkt vernachlässigt werden kann. Weiterhin ist keine Autovervollständigung verfügbar. Diese ist in Codesys ebenfalls als Komponente implementiert, konnte jedoch nicht durch ein öffentliches Interface genutzt werden. Die Syntax von Structured Text ist jedoch nicht so umfangreich, sodass dies nur eine kleine Einschränkung darstellt. Weiterhin wird in Abschnitt 5.2 beschrieben, dass die Evaluation ausschließlich mit dem entwickelten InjEditor

durchgeführt wird, sodass Effekte durch die fehlende Autovervollständigung bei einem Vergleich keine Rolle spielen.

Insgesamt wurde so ein Editor geschaffen, der den Anforderungen an einen Prototypen für die Evaluation des Unterstützungssystems entspricht. Abgedeckt wird Anforderung 10, da der Editor in die SPS-Entwicklungsumgebung Codesys integriert ist. Auch Anforderungen 11 und 12 sind erfüllt, denn Warnungen und Hinweise lassen sich im Code annotieren und es wurde eine Lösung gefunden, um die Wertemengen-Informationen weniger auffällig darzustellen.

4.3.3 Arcade3S

Der InjEditor ist einer von zwei Plugins, die für das Unterstützungssystem in Codesys entwickelt wurden. Das andere Plugin wurde Arcade3S genannt und übernimmt die Kommunikation zwischen dem ARCADE.PLC-Server und der Editor-Instanz. Es wurde als Teil von [Müllers, 2018] realisiert.

In Abbildung 4.12 wurde bereits die Architektur vorgestellt, mit deren Hilfe die Statische Analyse in Codesys integriert wird. Arcade3S übernimmt hier die Funktion des Websockets auf Codesys-Seite. Der Funktionsumfang des Plugins geht allerdings über die reine technische Bereitstellung eines Sockets hinaus. Das derzeit geschriebene Programm wird von diesem Plugin an ARCADE.PLC gesendet und die Ergebnisse der Analyse werden anschließend für die Anzeige im Editor bereitgestellt.

Damit ergeben sich die folgenden Aufgaben für das Arcade3S-Plugin:

- Überwachen des aktuellen Quellcodes
- Übertragung des Quellcodes an den ARCADE.PLC-Server inkl. Formatierung des Codes
- Parsen der Antwort des ARCADE.PLC-Servers
- Generieren von Meldungen aus den Ergebnissen über den Message-Store

Eine eigene grafische Oberfläche ist für das Plugin nicht nötig. Es arbeitet eigenständig im Hintergrund von Codesys und tauscht sich mit den verbundenen Komponenten in geeigneter Weise aus. Gestartet wird die Arbeit durch einen Menübefehl, der sich in der dynamisch konfigurierbaren Menüstruktur von Codesys nach dem erstmaligen Laden des Plugins in ein beliebiges Menü einfügen lässt. Für die Plugin-API ist Arcade3S damit ein *command* (vgl. Abbildung 4.13) und damit die einfachste Art, Funktionen in Codesys hinzuzufügen. Über den Menüeintrag kann das Plugin ein- und ausgeschaltet werden, um die Unterstützungsfunktionen entsprechend zu aktivieren oder zu deaktivieren. Weitere grafische Bedienelemente sind für die Funktion nicht nötig.

Einmal gestartet wartet das Plugin auf Änderungen des Quellcodes. Dieser wird im Editor geschrieben und von dort in den Object-Storage geschrieben. Der Zeitpunkt dafür obliegt dem Editor selbst und tritt im InjEditor ein, wenn der SPS-Programmierer 500 ms keine Eingaben mehr getätigt hat. Der SynEd-Editor verwendet ähnliche Auslöser für das Aktualisieren des Quelltextes in den Object-Storage, die jedoch nicht offen dokumentiert sind.

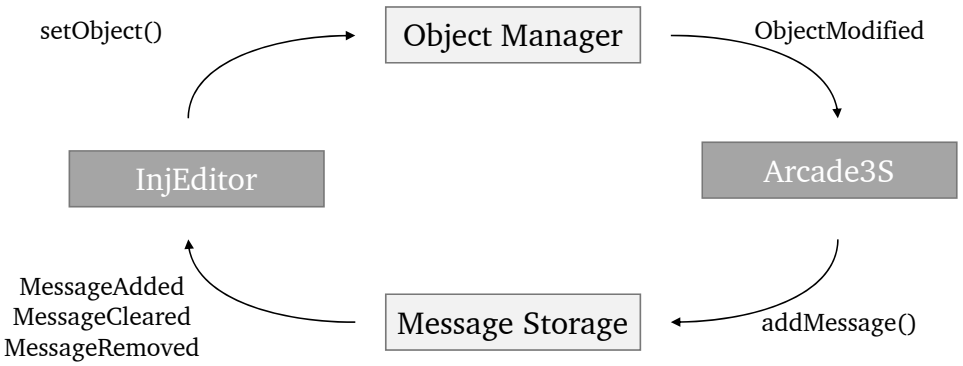


Abbildung 4.16: Kommunikationsschema zwischen Arcade3S und dem Editor InjEditor

Abbildung 4.16 zeigt das Zusammenspiel der entwickelten Plugins mit den Codesys-Systemkomponenten. Der InjEditor sendet die aktuelle Version des Quellcodes an den Object-Manager. Dazu wird die `setObject()`-Methode des Managers verwendet, um den Quellcode der aktuellen POE dort zu hinterlegen. Von technischer Seite gibt es hier eine Aufteilung in zwei Dokumente für die POE: Den Deklarations- und den Programm-Teil. Die `setObject()`-Methode wird daher für diese beiden Teile aufgerufen.

Um über Änderungen am Quellcode informiert zu werden, registriert sich das Arcade3S-Plugin zu Beginn beim Object-Manager. Wie andere Systemkomponenten in Codesys auch, verwendet dieser Manager ein Event-Modell. Sind andere Komponenten an bestimmten Änderungen interessiert, haben sie die Möglichkeit, mithilfe des Delegate-Pattern einen Eventhandler für die Reaktion darauf anzumelden. Das Arcade3S-Plugin wird über den registrierten `ObjectModified`-Delegaten dann informiert, wenn der Editor eine neue Iteration des Programms an den ObjectManager übergeben hat.

An diesem Punkt regelt das Plugin dann die empfangenen Analyseergebnisse und bereitet die Informationen für den Editor auf. Der Informationsaustausch dorthin erfolgt über die `addMessage()`-Methode des Message-Stores ähnlich dem Aufruf des Editors beim Object-Manager. Hier werden auch Prioritäten der Meldung angegeben, um sie im Editor entsprechend darstellen zu können. Diese Meldungen mit Prioritäten werden dann vom InjEditor aus dem Message-Store gelesen. Dabei kommen die drei Delegaten `MessageAdded` beim Hinzufügen einer Meldung, `MessageCleared` beim Löschen aller Meldungen oder `MessageRemoved` beim Löschen einzelner Meldungen zum Einsatz. Zum Abschluss werden die Meldungen vom Editor im Quellcode annotiert.

Nachdem die Einbettung und das Zusammenspiel im Editor betrachtet wurden, werden die Verarbeitungsschritte des Plugins nun erläutert. Das Senden des aktuellen Programms umfasst neben dem Übertragen über den Websocket die Zusammenstellung des Programms. Codesys zeigt Variablendeklarationen und den Programmcode getrennt an und speichert sie getrennt im Object-Store, ARCADE dagegen erwartet sie als eine zusammenhängende Einheit. Für die Übertragung werden also beide Texte einfach hintereinander gesetzt und mit einem Abschlusstoken beendet. Dieses Abschlusstoken ist ein vom Typ der POE abhängiges Schlüsselwort und lautet bei Programmen beispielsweise `END_PROGRAM`. Das so entstandene Programm liegt dann in der IEC-61131-3-konformen Form vor, die ARCADE erwartet. Zu-

sammen mit dem Attribut, dass der Structured-Text-Dialekt von Codesys verwendet werden soll, wird der so zusammengesetzte Code dann mittels eines HTTP-POST-Requests an die lokal laufende ARCADE.PLC-Instanz gesendet. Diese muss zuvor gestartet worden sein, kann jedoch ausgeblendet im Hintergrund bleiben.

Als Antwort auf das Programm versendet der ARCADE-Server nach erfolgter Analyse das Paket aus Warnungen, Hinweisen und den errechneten Wertemengen für die Variablen. Auf die Konstruktion dieser Antwort im JSON-Format ging bereits Abschnitt 4.2.4 ein. Zum Parsen wird im Projekt die Newtonsoft JSON.NET-Bibliothek[10] verwendet, die sich durch eine einfache und sehr performante Verarbeitung von JSON-Objekten auszeichnet. Die auf diesem Weg erhaltenen Meldungen müssen dann für die Verarbeitung im Message-Store vorbereitet werden. Die Meldungen bestehen üblicherweise aus dem Text, der annotiert werden soll und einer Positionsinformation, also einem Zeilen- und Zeichenindex für den Beginn und das Ende der markierten Quellcode-Stelle auf die sie sich bezieht. Während der Zeichenindex unproblematisch ist, muss der Zeilenindex angepasst werden. Die Zeilenindizes von ARCADE.PLC beziehen sich auf den Beginn des übertragenen Quellcodedokumentes, schließen also den Variablendeklarations-Teil wieder mit ein. Die Unterscheidung nach Deklarations- und Implementierungs-Teil kann anhand eines Vergleichs mit der Zeilenanzahl im Deklarationsteil leicht erfolgen. Für Annotationen am Implementierungsanteil ist diese Zeilenanzahl dann als Offset abzuziehen. Darüber hinaus arbeitet Codesys jedoch intern mit einer individuellen ID für jede Zeile, damit eine Identifikation einer Zeile eindeutig möglich bleibt, auch wenn Zeilen hinzugefügt, verschoben oder entfernt wurden. Die Umrechnung der Zeilenindizes in diese IDs kann in Codesys mithilfe des POE-Objektes erfolgen, das über den Objekt-Manager abgerufen wurde. Auf Seiten des Editors müssen diese IDs wieder in Zeilennummern der jeweiligen Darstellungen umgerechnet werden. Grund dafür ist, dass das AvalonEdit-Fenster wieder eigene Methoden bereit hält, um auch bei sich änderndem Quelltext eine eindeutige Adressierbarkit zu ermöglichen. Für eine einheitliche Nutzung in Codesys und die erwartungsgemäße Funktion von Meldungsfenstern und Darstellungen in Codesys sind jedoch alle genannten Formen der Zeilen-Adressierung nötig. Für die Warnungs- und Hinweismeldungen sind so alle benötigten Informationen bereit, um über den Message-Store mit `addMessage()` veröffentlicht zu werden.

Lediglich die Wertemengen-Annotationen müssen noch weiter behandelt werden. Hierfür stehen die folgenden Informationen bereit:

Zeilenindex Zeile des Eingabeprogramms, auf die sich die Meldung bezieht.

Variable Name der Variable, auf die sich die Wertemengen beziehen.

Wertemenge Eine Zeichenkette mit möglichen Werten. Beispiele: $[0-12]$ oder $0, 5, 12$.

Nach Zuweisung Wertemengen nach Ausführen der Zeile, falls eine Zuweisung erfolgt.

Im Gegensatz zu den Meldungen zuvor stehen also Zeilen- aber keine Zeichen-Informationen zur Verfügung. Die Position einer Variable in einer Zeile muss also zunächst mithilfe des Quellcodes und eines regulären Ausdrucks für den Variablennamen in der entsprechenden

[10] Projektwebseite: https://www.newtonsoft.com/json

Zeile gesucht werden. Falls die Variable mehrfach vorkommt, wird sie auch mehrfach in der Zeile annotiert. Dies gilt auch für Kommentare, die damit als zusätzliche Hilfestellung im Quellcode eingetragen werden können. Beinhaltet die Zeile jedoch eine Zuweisung (`:=`), so wird neben der normalen Wertemenge von ARCADE.PLC noch ein Wert nach der erfolgten Zuweisung berechnet und übermittelt. Er spiegelt die möglichen Werte nach erfolgter Zuweisung wieder. Es wird dann die Variable links des Zuweisungsoperators (engl. *left hand side*) mit einem Hinweistext versehen, der den alten und neuen Wert der Variable beinhaltet, um die Änderung der Wertemenge einfacher nachvollziehen zu können. Als Form wurde dafür „*{Alte Werte}* → *{neue Werte}*“ verwendet. Zeilennummern werden analog zu Warnungen und Hinweisen umgeformt.

Alle bisher beschriebenen Operationen des Arcade3S-Plugins werden vom Codesys-Haupt-Thread unabhängig ausgeführt. Diese Trennung ist sinnvoll, um die grafische Oberfläche kontinuierlich nutzen zu können auch wenn das Plugin derzeit auf die Antwort der ARCADE.PLC-Instanz wartet. Diese Antwort kann je nach Programmgröße zwischen einigen wenigen Millisekunden und einigen hundert Millisekunden liegen und würde das Nutzererlebnis negativ beeinflussen. Schwierig ist in diesem Fall der gemeinsame Zugriff auf Ressourcen bzw. Komponenten. Der lesende Zugriff auf den Quellcode mithilfe des Object-Managers ist noch möglich, das hinzufügen von Nachrichten in den Message-Store musste jedoch Threadsicher erfolgen. Für diesen Zweck sieht Codesys die Engine-Komponente vor. Dieser Komponente wird wieder mithilfe des Delegate-Entwurfsmusters eine Methode übergeben, die asynchron vom restlichen Plugin in Zukunft im Haupt-Thread ausgeführt wird. Zeitliche Garantien gibt Codesys hierfür zwar nicht, doch konnten in Testläufen des Plugins keine sichtbaren Verzögerungen hierdurch festgestellt werden. Weiter ist dies der einzige dokumentierte Weg, um Daten Threadsicher in den Speicherbereich des Haupt-Threads schreiben zu können.

Ein weiterer Aspekt muss hinsichtlich der zeitlich unabhängigen Analyse noch betrachtet werden. So ist es möglich, den Quellcode des SPS-Programms zu ändern, während dieses aktuell noch analysiert wird. Die Ergebnisse können so direkt veraltet sein oder es kann trotz der Umrechnungsfunktionen zu Fehlern bei der Zuordnung der Zeilennummern kommen. Die Analyseergebnisse sollen in diesem Fall nicht weiter im Editor angezeigt werden. Arcade3S prüft während ARCADE den Code analysiert, ob es aktualisierte Versionen des Programms gibt. Ist dies der Fall, wird die Analyse durch Abbruch des auf Antwort wartenden Threads gestoppt und das Programm in aktualisierter Form erneut verschickt. Erst diese Antwort wird dann im Editor angezeigt, wenn bis dahin keine erneute Änderung erkannt wurde.

Das Plugin kann mit den beschriebenen und implementierten Funktionen den aktuell betrachteten Quellcode an ARCADE.PLC senden und die Ergebnisse in eine Codesys-konforme und darstellbare Form überführen. Aktuell unterstützt das Arcade3S-Plugin nur die Analyse einer POE, nicht jedoch SPS-Programme, die aus mehreren POEs bestehen. Anforderung 3, die die Unterstützung mehrerer vom Benutzer geschriebener POEs fordert, ist somit in diesem Prototyp nicht erfüllt. Diese könnten dem übertragenen Programm für den ARCADE.PLC-Server angehängt werden, um dort für die Analyse zur Verfügung zu stehen. Das Analyse-Back-End unterstützt bereits das Analysieren von mehreren POEs in einem

Analyseschritt. Für die Evaluation in Abschnitt 5 wurde diese Funktion jedoch nicht benötigt und daher nicht integriert.

Abschließend betrachtet, ist das Arcade3S-Plugin das letzte Bindeglied, um die Unterstützungsfunktionen in einer SPS-Entwicklungsumgebung darstellen zu können. Es regelt die Übertragung des Programm-Quelltextes an ARCADE.PLC und die Aufbereitung der Analyseergebnisse für die Anzeige im InjEditor-Plugin. Eine Darstellung im Codesys-eigenen SynEd-Editor ist ebenso möglich, lediglich die Wertemengen-Informationen lassen sich hier nicht darstellen und die Übertragung des Programms von diesem Editor in den Object-Manager findet nach anderen Timeouts statt. Das Arcade3S-Plugin sorgt damit zusammen mit dem implementierten InjEditor dafür, dass Anforderungen 1 und 2 erfüllt sind, also die Annotation von Warnungen und Hinweisen sowie Wertemengen an Variablenbezeichnern an beliebigen Stellen im Programmquelltext. Anforderung 4, die automatische Neuberechnung der Analyse, wird als zentrale Aufgabe von diesem Plugin realisiert und ist damit erfüllt.

4.4 Inkrementelle Analyse

Die Anbindung der Statischen Analyse aus ARCADE.PLC an eine Benutzeroberfläche zur Darstellung von Meldungen und Wertemengen für die Unterstützung der SPS-Entwicklung ist ein Ziel der vorliegenden Arbeit. Um dieses Ziel zu erreichen, wurde bereits in Abschnitt 3.2 festgelegt, dass die Bereitstellung dieser Informationen geeignet schnell erfolgen muss, um keine deutlich sichtbar verzögerte Darstellung zu verursachen (Anforderung 8). Eine genauere Zeitangabe wurde zuvor nicht festgelegt. Die Beobachtung bei konventionellen IDEs zeigt allerdings, dass Annotationen am Quelltext üblicherweise in etwa einer Sekunde aktualisiert werden. Diese Dauer wird daher von einem SPS-Programmierer erwartet bzw. als Normalfall angesehen. Mit dieser Zielvorgabe wird in diesem Abschnitt untersucht, welche Berechnungszeit die Statische Analyse benötigt und wie mithilfe einer inkrementellen Analyse die Analysezeit verkürzt werden kann. Abschnitt 4.4.1 beschreibt dafür den Analyseprozess mit den jeweils nötigen Schritten genauer, woraus in Abschnitt 4.4.3 ein Konzept für eine Beschleunigung des Analyseprozesses vorgestellt wird. Im folgenden Abschnitt 4.4.4 wird die Implementierung beschrieben, die unter 4.4.5 evaluiert und in Abschnitt 4.4.6 diskutiert.

Eine erste Umsetzung einer inkrementellen Analyse wurde im Rahmen der Abschlussarbeit [Shaaban, 2016] implementiert. Ergebnisse daraus wurden jedoch nicht verwendet. Ein eigenes Konzept der für die vorliegende Arbeit implementierten inkrementellen Analyse wurde in der Publikation [37] veröffentlicht. Die Abschnitte 4.4.1 und 4.4.3 beziehen sich daher auf diese Veröffentlichung.

4.4.1 Beobachtung

Die in ARCADE implementierte Statische Analyse ist im Verhältnis zu anderen formalen Methoden, die auf Programmcode angewendet werden, verhältnismäßig schnell. Wie lange

Programm	Programmzeilen	POEs	Dauer
App1 / Programm1	233	3	< 1 s
App2 / Programm2	2776	100	11 s
App2 / Programm3	169	5	3 s
App2 / Programm4	2684	100	146 s
App2 / Programm5	206	12	< 1 s
App3 / Programm6	344	12	< 1 s
App4 / Programm7	3339	18	40 s

Tabelle 4.1: Teil der Fallstudie von Statischer Analyse durch ARCADE.PLC auf industriellem ST-Code (aus [48]). Programmnamen und deren Funktionen wurden aufgrund der Industriekooperation anonymisiert.

die Analyse tatsächlich dauert und in welchen Phasen der Analyse die meiste Rechenzeit benötigt wird, soll in diesem Abschnitt untersucht werden.

In der Publikation [48] wurde die Statische Analyse von industriell genutzten SPS-Programmen mit ARCADE.PLC untersucht. Evaluiert wurden sieben Programme mit einer Gesamt-Zeilenanzahl zwischen 169 und 3339 sowie zwischen drei und 100 POEs. Das Ergebnis der Evaluation ist in Tabelle 4.1 dargestellt. Gemessen wurde die Dauer der Statischen Analyse auf den jeweiligen Programmen, also einer kompletten Analyse inkl. der in dieser Arbeit besonders betrachteten VSA, der Wertemengenanalyse.

Während Analysezeiten von unter einer Sekunde für den in dieser Arbeit angestrebten Anwendungsfall akzeptabel sind, fallen die Programme 2, 3, 4 und 7 nicht mehr in den angestrebten Zeitrahmen.

Aus der Evaluation lässt sich vorab schon eine weitere Erkenntnis ziehen. So besteht kein direkter Zusammenhang zwischen der Anzahl der zu analysierenden Zeilen oder den POEs und der Analysedauer. Ein Grund dafür liegt in der Arbeitsweise der abstrakten Interpretation. Es ist vielmehr von Zusammenhängen innerhalb des Programms, sowie von Sprüngen und Schleifen abhängig, ob die Analyse einen Fixpunkt findet oder weiter über den Kontrollflussautomat iteriert. Weiterhin beeinflusst die Anzahl an Variablen im jeweiligen Kontext einer POE die Analysedauer. Insgesamt kann aus der Evaluation der Schluss gezogen werden, dass eine Optimierung der Laufzeit der Statischen Analyse sinnvoll ist, um Anforderung 8 zu erfüllen, also dass das Unterstützungssystem ohne sichtbare Verzögerungen arbeitet.

Der Prozess der Analyse und der abstrakten Interpretation im Speziellen wurde daher weiter untersucht. Das Testsetup wurde gegenüber der Evaluation in [48] geändert, da die dort verwendeten Programme nicht zur Verfügung standen. Stattdessen wurden andere SPS-Programme verwendet, um die Dauer der Bearbeitungsschritte einzeln erfassen zu können.

Zunächst werden die Schritte des Analyseprozesses dafür näher betrachtet. Abbildung 4.17 zeigt diese für das Beispiel des Structured-Text-Codes. Der Code wird in einen abstrakten Syntaxbaum geparst, der in die Intermediate Representation übersetzt wird. Aus diesem wird der Kontrollflussautomat erstellt. Bei diesen Schritten sollen getrennt voneinander

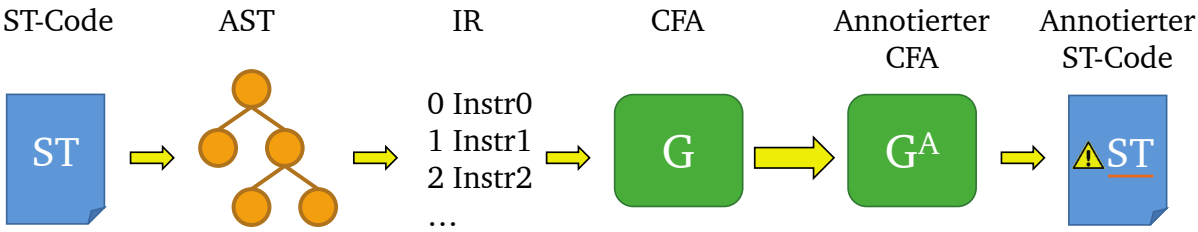


Abbildung 4.17: Schritte im ARCADE.PLC -Analyseprozess

Zeitmessungen durchgeführt werden, um über ihren Einfluss auf die gesamte Analysezeit Aussagen treffen zu können. Der Kontrollflussautomat wird schließlich durch die darauf operierenden Analyseverfahren annotiert mit dem Ergebnis des annotierten CFAs. Die Zeit für diese Analysen selbst wurde ebenfalls aufgenommen. Nachdem der CFA annotiert ist, werden darauf Prüfverfahren angewendet, die jeweils den Automaten durchlaufen und auf Auffälligkeiten hin überprüfen. Die so gefundenen Auffälligkeiten werden gesammelt und als Warnungen und Hinweise für Annotationen im Quellcode ausgegeben. Auch die Ausgabe der Wertemengen gehört in diesen Schritt.

Getestet wurden exemplarisch zwei Programme. Das Programm `Safety` besteht aus 23 Variablen und lediglich 13 Programmzeilen, in denen Safety-Funktionsblöcke der PLCopen aufgerufen werden. Die aufgerufenen Programmblöcke haben zusammen noch einmal 496 Zeilen Programmcode. Das zweite Testprogramm `Random` besteht aus nur einer POE mit 1000 Variablen, 600 Zeilen Programmcode aus zufällig generierten Zuweisungen, arithmetischen Operationen und `IF`-Anweisungen. Andere POEs werden nicht aufgerufen.

Die Zeitmessungen wurden auf einem Lenovo Notebook mit einem Intel Core i5 der dritten Generation und 16 GB Arbeitsspeicher durchgeführt. Als Betriebssystem kamen Windows 7 zusammen mit dem Java-Runtime-Environment von Oracle in der Version 1.8 zum Einsatz. Der Einfluss durch Hintergrundprozesse wurde weitestgehend ausgeschlossen und durch mehrfache Wiederholung der Tests relativiert. Um die Zeiten zu messen wurden in ARCADE an geeigneten Stellen Zeitstempel mithilfe der Java-Funktion `System.nanoTime()` aufgenommen und verglichen.

In Tabellen 4.2 und 4.3 sind die Ergebnisse der Zeitmessungen auf den beiden Programmen notiert. Sie bestehen jeweils aus zwei Messreihen für die erste Analyse und die darauf folgenden Analyse-Läufe. Der teils erhebliche Laufzeitunterschied ist durch folgende Einflussfaktoren zu erklären:

- Der Java Just-in-Time-Compiler (JIT) übersetzt Teile des Programmcodes bei der ersten Verwendung, um sie bei der folgenden Nutzung schneller ausführen zu können. Damit ist die erste Ausführung immer langsamer
- Speicher muss für AST, IR, CFA und weitere Datenstrukturen alloziert werden, der bei Folgeanalysen in der Java-VM bereits reserviert ist.
- Beim ersten Parsen werden Standard-Bibliotheken mit geladen, die danach nicht erneut geparst werden müssen.

Analyseschritt	Zeiten erste Analyse	Zeiten danach
Parsen	375 ms	6 ms
IR generieren	24 ms	2 ms
CFA generieren	18 ms	17 ms
Analysen	1426 ms	361 ms
Prüfverfahren	85 ms	30 ms

Tabelle 4.2: Zeitmessungen der Analyseschritte auf dem `Safety`-Programm (aus [37], © 2017 IEEE) mit gemittelten Zeiten über jeweils 10 Messungen

Analyseschritt	Zeiten erste Analyse	Zeiten danach
Parsen	580 ms	50 ms
IR generieren	52 ms	6 ms
CFA generieren	20 ms	4 ms
Analysen	4594 ms	3614 ms
Prüfverfahren	150 ms	60 ms

Tabelle 4.3: Zeitmessungen der Analyseschritte auf dem `Random`-Programm mit gemittelten Zeiten über jeweils 10 Messungen

Der letztgenannte Einflussfaktor wurde vor dem Vergleich als Optimierung implementiert, da Standardbibliotheken für grundlegende Rechenfunktionen wie `ADD()` und `MAX()` aber auch Timer und Bibliotheken der PLCopen beim ersten Lauf der Analyse zunächst geparst und angelegt werden. Die Objekte dafür werden gespeichert und müssen nicht erneut angelegt werden. Diese Änderung allein ist für eine Nutzung der Analyse im Unterstützungssystem bereits sinnvoll und im Hinblick auf die akzeptable Dauer der Analyse in Kombination mit den JIT-Optimierungen bei dem `Safety`-Programm bereits ausreichend.

Über die hier gemessenen Werte hinaus wurde eine genauere Aufschlüsselung der Berechnungszeiten einzelner Analysen für das `Safety`-Programm vorgenommen. Auffällig war dabei, dass die Wertemengenanalyse (VSA) bei diesem Programm den größten Teil der Berechnungszeit einnahm, während sich die anderen Analyseverfahren in deutlich kürzerer Zeit berechnen ließen. Zu den sonstigen Analysen gehören unter anderem die Live-Variable-Analyse (LVA), deren Ergebnisse von der VSA verwendet werden. Auf die VSA allein entfallen im ersten Analyselauf 1321 ms bzw. 332 ms in den späteren Wiederholungen. Die übrigen Analyseverfahren benötigen dagegen nur 105 ms bzw. 29 ms.

Aus den Messungen lassen sich Schlüsse ziehen. Erstens benötigt die VSA einen signifikanten Anteil an der Berechnungszeit. Insbesondere fallen die Schritte des Parsens, das Generieren von IR und CFA sowie die Prüfverfahren im Vergleich dagegen kaum ins Gewicht. Für ein Unterstützungssystem ist es daher sinnvoll, die VSA näher zu betrachten und nach Optimierungen dafür zu suchen, um eine schnellere Reaktion auf die Eingaben des SPS-Programmierers zu erreichen. Die zweite Erkenntnis ist, dass der Unterschied zwischen dem ersten und den folgenden Analyseläufen ebenfalls signifikant ist. Solange der Kontext

der Analyse jedoch erhalten bleibt, der ARCADE.PLC-Server also nicht beendet wird und die Analyse regelmäßig durchgeführt wird, kann auch weiterhin von den Optimierungen des JIT-Compilers und der vorhandenen Speicherallokation profitiert werden. Daher wird im folgenden lediglich die erste Erkenntnis weiter betrachtet und eine Optimierung der VSA vorgestellt.

Analyseläufe werden nun in zeitlicher Abfolge betrachtet. Wie in Abschnitt 4.2.3 beschrieben, werden während der Programmierung nur syntaktisch korrekte Programmversionen analysiert. Diese bezeichnen wir im Folgenden als Programmversion bzw. Iteration n und die nächste Version des Programms als $n + 1$, um über die zeitliche Entwicklung des Programmcodes Aussagen zu treffen.

Wir treffen an dieser Stelle die Annahme, dass zwischen zwei aufeinanderfolgenden Iterationen des Programms jeweils nur einzelne oder wenige Instruktionen hinzugefügt, geändert oder gelöscht werden. Im Ergebnis einer VSA bleiben dann möglicherweise viele Variablen, auf die die geänderten Instruktionen keinen Einfluss haben. Dennoch müssen diese komplett neu berechnet werden, da eine Übernahme berechneter Werte und die Betrachtung des Programms in einem zeitlichen Verlauf bisher nicht vorgesehen ist. Eine Neuberechnung nur der geänderten Wertemengen könnte die nötige Berechnungszeit der Analyse verkürzen. Aus dieser Kernidee wird im Folgenden das Verfahren der inkrementellen Analyse aufgebaut. Im Kontext dieser Arbeit bezieht sich die inkrementelle Analyse stets auf eine inkrementelle Variante der Wertemengenanalyse.

4.4.2 Formale Grundlagen des ARCADE.PLC Analyseframeworks

In diesem Abschnitt werden kurz formale Grundlagen der Analyseverfahren in ARCADE.PLC eingeführt, um im Konzept-Abschnitt 4.4.3 darauf zurückgreifen zu können. Die Statische Analyse in ARCADE.PLC wurde in erster Linie durch Sebastian Biallas im Rahmen seiner Dissertation [3] implementiert und in Studien evaluiert. Erläuterungen und Formalisierungen in diesem Abschnitt finden hier daher in Analogie zu [3, 6, 56] statt. Die grundlegenden Analysen haben sich im Vergleich zu den genannten Veröffentlichungen nicht geändert, operieren mittlerweile aber auf Kontrollflussautomaten (CFAs) statt auf Kontrollflussgraphen (CFGs). Der Unterschied besteht lediglich in der formalen Beschreibung der Datenstruktur und in Implementierungsdetails. Die darauf arbeitenden Verfahren wurden für die Nutzung des CFAs angepasst und besitzen weiterhin die gleichen Eigenschaften hinsichtlich ihrer Laufzeiten und Ergebnisse.

Ein CFA beinhaltet die Instruktionen des Programms in der Zwischendarstellung (IR). Im Unterschied zum Kontrollflussgraphen, bei dem die Instruktionen den Knoten des Graphen entsprechen und eine Kante die möglichen Nachfolgebeziehungen im Programm abbildet, trägt der Kontrollflussautomat die Instruktionen auf seinen Transitionen und besitzt Zustände, die den Programmzähler des Programms modellieren. Die Zustände des Automaten werden im Rahmen der Analyse mit weiteren Analysedaten annotiert, die bei Betrachtung einer Transition dem abstrakten Programmzustand vor bzw. nach dem Ausführen einer Instruktion entsprechen.

Eine Besonderheit beim CFA sind Transitionen für Bedingungen, wie sie bei `IF`-Anweisungen oder beim Überprüfen einer Schleifen-Invariante vorkommen. Diese Instruktionen

werden in zwei Transitionen übersetzt: Eine, in der die Bedingung erfüllt ist und eine, in der die Bedingung nicht erfüllt ist. Sie enden in zwei unterschiedlichen Zuständen des Automaten und teilen damit den Kontrollfluss in zwei verschiedene Pfade auf. In einem CFG entsprach dies zwei Nachfolgeknoten bzw. ebenfalls separaten Pfaden.

Ein Programm kann jeweils aus einer oder mehreren POEs bestehen, die in den CFA überführt wurden. Auch der Einfluss globaler Variablen auf andere POEs lässt sich analysieren, wenn POEs vom Typ `PROGRAM` analysiert werden. Die für SPSen übliche zyklische Ausführung von Programmen wird im CFA durch eine zusätzliche Transition vom Ende des Programms zum Startzustand modelliert.

ARCADE nutzt für mehrere Analysen einen Worklist-Algorithmus. Grundprinzip dieses Algorithmus ist eine Liste an Transitionen aus dem CFA, für die mithilfe der Datenflussanalyse Informationen (neu)berechnet werden müssen. Im Fall der VSA enthält diese Liste die Menge an Transitionen, für die aus ihrem Startzustand mittels abstrakter Interpretation Wertemengen für den nachfolgenden Zustand berechnet werden.

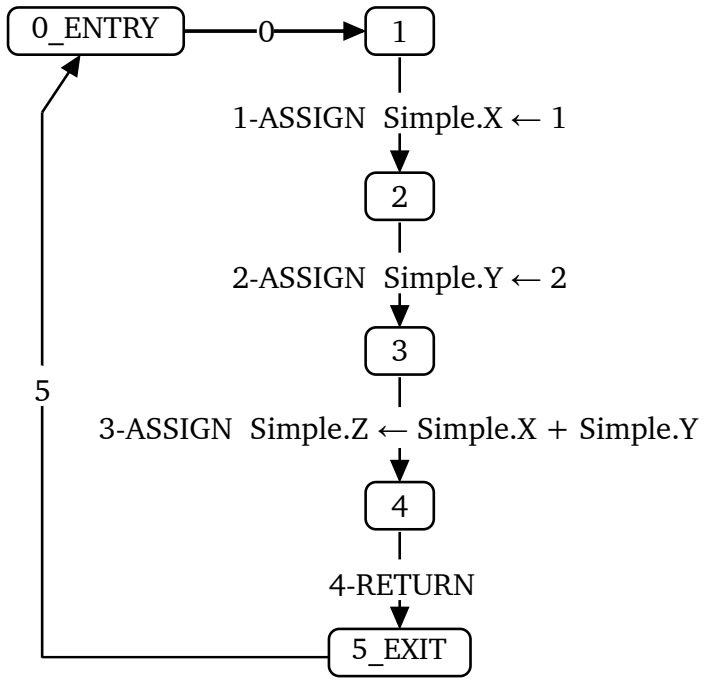
Um ein SPS-Programm zu analysieren, wird es so mithilfe der abstrakten Interpretation simuliert. Dafür wird die Variablenmenge $\mathcal{V}$ des Programms genutzt und auf einer abstrakten Domäne $\mathcal{D}$ interpretiert. Als Domänen kommen in der Implementierung je nach Datentyp konkrete Wertemengen, Intervalle oder Bitvektoren einzeln, meistens jedoch in Kombination zum Einsatz. Ein Element der Domäne steht dann für eine Menge an Werten, die die Variable annehmen kann. Werden beispielsweise Wertemengen natürlicher Zahlen $\mathbb{N}$ als Domäne verwendet, so ist $[2, 5]$ ein Element aus der Domäne und repräsentiert die Variablenwerte 2, 3, 4 und 5.

Abweichend zur Definition und Beschreibung von Biallas [3] wird nun jedem Zustand des CFA eine Belegung der Variablen auf ein Element der abstrakten Domäne erfolgen. Bezeichne $\mathcal{N}$ die Menge der Zustände des CFAs und $n \in \mathcal{N}$ einen einzelnen Zustand. Für jeden Zustand n weist dann die Abbildung $s_n : \mathcal{V} \to \mathcal{D}$ Variablen des Programms Elemente der Domäne, also Wertemengen zu. S bezeichne die Menge aller möglichen Abbildungen, die auf diese weise möglich sind. Diese Definitionsänderung gegenüber der der ursprünglichen Definition durch Biallas ist lediglich der Veränderung in ARCADE von CFGs zu CFAs geschuldet. Inhaltlich wird das Verfahren dadurch nicht geändert.

In Abbildung 4.18 sind Abbildungen s_n für den dargestellten Graphen aufgelistet. Die Abbildungen in der Tabelle sind dabei bereits das Ergebnis der VSA. Bei Initialisierung und während der schrittweisen Berechnung werden den Variablen entsprechend andere Werte zugewiesen, sodass an den jeweiligen Zuständen des Graphen andere Abbildungen $s'_n \in S$ gelten. Zur einfacheren Übersicht ist hier jedoch nur das Endergebnis dargestellt.

In den Domänen gibt es jeweils zwei besondere Elemente, die das Minimum und das Maximum der Menge repräsentieren. $\top$ ist das Maximum und entspricht dem kompletten Wertebereich einer Variable, also beispielsweise $[0 - 255]$ bei einer 8-Bit-Variable mit natürlichen Zahlen. $\bot$ ist das Minimum der Menge und entspricht einer leeren Menge $\{\}$. Zum Simulieren der Ausführung gibt es dann zwei wichtige Operationen:

$$\begin{aligned} \text{Überführungsfunktion für } \texttt{op} \quad & f^{op} : S \to S \\ \text{Vereinigungsoperator} \quad & \sqcup : S \times S \to S \end{aligned}$$



S	$\mathcal{V} \to \mathcal{D}$
s_{2-1}	$X = \{0,1\}$
	$Y = \{0,2\}$
	$Z = \{0,3\}$
s_{3-2}	$X = \{1\}$
	$Y = \{0,2\}$
	$Z = \{0,3\}$
s_{4-3}	$X = \{1\}$
	$Y = \{2\}$
	$Z = \{0,3\}$
s_{5-4}	$X = \{1\}$
	$Y = \{2\}$
	$Z = \{3\}$
s_{1-5}	$X = \{1\}$
	$Y = \{2\}$
	$Z = \{3\}$

Abbildung 4.18: Formale Darstellung der Variablenmengen-Zuweisungen $s_n : \mathcal{V} \to \mathcal{D}$ zu dem Beispiel aus Abbildung 4.5. Die Abbildungen s_{2-1}, s_{3-2}, etc. definieren für die Zustände 2-1, 3-2, etc. jeweils die Abbildung der Variablen auf Elemente der Domäne. In diesem Fall entspricht $\mathcal{D}$ Mengen ganzzahliger, natürlicher Zahlen.

Die Überführungsfunktion f^{op} wendet eine Operation op der IR auf eine Abbildung s an. Bei einer Zuweisung wird hier beispielsweise eine Berechnung des Ausdrucks auf der rechten Zuweisungsseite in der gewählten Domäne ausgeführt und das Ergebnis der Variable in s zugewiesen, die auf der linken Zuweisungsseite steht. Der Vereinigungsoperator $\sqcup$ nimmt eine element- bzw. variablenweise Vereinigung der Wertemengen vor und entspricht bei einfachen Mengen damit dem Operator $\cup$.

Schließlich ist noch die Relation $\sqsubseteq$ auf der Menge S definiert. Für zwei Elemente $s, s' \in S$ gilt die Relation $s \sqsubseteq s'$ genau dann, wenn jeder abstrakte Wert, der in s einer Variable zugewiesen ist, dieser auch in s' zugewiesen sein muss. Hat in s also eine Variable den Wertebereich $[0-128]$, so muss ihr in s' mindestens ebenfalls dieser Wertebereich zugewiesen sein. Der Vereinigungsoperator ist dann so konstruiert, dass stets gilt: $s^* = s \sqcup s'$ sodass $s \sqsubseteq s^*$ und $s' \sqsubseteq s^*$ gelten.

Mit diesen Operationen kann nun ein Worklist-Algorithmus auf dem CFA eingesetzt werden. Die Idee dabei ist es, den Automaten zu traversieren, an jedem Zustand jeweils die Operationen ausgehender Transitionen auf den gespeicherten Wertemengen zu simulieren und das Ergebnis mit dem Nachfolgezustand zu vereinigen.

Der Algorithmus beginnt auf einem CFA, bei dem die zu den Zuständen $\mathcal{N}$ assoziierten Zuweisungen s_n jeweils mit $\bot$ initialisiert sind. Alle unbekannten Eingangs- und globalen Variablen werden mit $\top$ initialisiert, da sie prinzipiell alle Werte annehmen können. Die Worklist selbst ist als Warteschlange (*engl.* Queue) implementiert und wird zu Beginn mit

allen Transitionen des Automaten initialisiert, beginnend mit den Transitionen aus dem Startzustand heraus. Unter Algorithmus 2 ist das verwendete Verfahren in Pseudocode festgehalten.

Daten: CFA und initialisierte Abbildungen $s_0 ... s_n$ für jeden Zustand
Ergebnis: Anotierter CFA

1 **Solange** *Worklist nicht leer*
2 | Entnehme Transition t aus der Worklist. n_i sei der Vorgänger-, n_j der Nachfolgezustand und op die Instruktion der IR auf der Transition t.
3 | Berechne $s'_1 = f^{op}(s_i)$
4 | Vereinige $s_j^{neu} = s_j \sqcup s'_j$
5 | **Wenn *nicht*** $s_j^{neu} \sqsubseteq s_j$ **dann**
6 | | Füge Nachfolgekanten von n_j der Worklist hinzu.
7 | **Ende**
8 | Weise dem Zustand seine neuen Werte zu $s_j = s_j^{neu}$
9 **Ende**

Algorithmus 2: Worklistalgorithmus auf einem CFA in Anlehnung an [3]

Dieser Algorithmus traversiert also alle Transitionen des CFAs und berechnet dabei jeweils aus Wertemengen am Vorgängerknoten und der Operation auf der Transition die Wertemengen des Nachfolgeknotens. Sind dort bereits Wertemengen hinterlegt, werden diese für jede Variable jeweils vereinigt. Dies ist bei jeder erneuten Traversierung der Transition und am Ende einer Verzweigung durch eine Bedingung oder Schleife der Fall. Zu Beginn einer Verzweigung von einer Bedingung kann diese Transition genutzt werden, um die Wertemengen zu präzisieren. Dabei werden solche Werte aus den abstrakten Mengen ausgenommen, die laut Bedingung auf dem Zweig nicht möglich sind.

Eine Besonderheit besitzt noch die Transition vom letzten CFA-Zustand des Programms, also dem Zustand nach der letzten oder den letzten Instruktionen. Die Transition verbindet diesen letzten Zustand mit dem Startzustand des CFA, was dem Zykluswechsel oder dem Start eines neuen Zyklus entspricht. Variablen wie `VAR`, die auch bei einem nächsten Zyklus ihren Wert behalten, bleibt die zuletzt berechnete Wertemenge erhalten und wird mit den Werten im Startzustand vereinigt. Eingangsvariablen werden wieder mit $\top$ und andere Variablen mit Initial- bzw. Defaultwerten belegt. Die abstrakten Wertemengen für nicht persistente Variablen beeinflussen die Mengen am Startzustand also nicht. Wertemengen von persistenten Variablen hingegen werden in den Startzustand übernommen und können die Berechnung der Werte im nachfolgenden Programmablauf beeinflussen.

Der genannte Algorithmus funktioniert und konvergiert durch die Vereinigung in Zeile 4 zu einem Fixpunkt. Dies ist der Fall, wenn nach jeder Vereinigung $s_j^{neu} \sqsubseteq s_j$ erfüllt ist, die neu berechneten Mengen also nicht mehr größer ist als die bereits vorhandenen Wertemengen. In diesem Fall werden keine neuen Transitionen mehr in die Worklist übernommen.

Das Konvergieren auf diese Weise kann unter Umständen lange dauern, wenn beispielsweise ein Wert in einer Schleife stets um 1 erhöht wird und dadurch s_j^{neu} für diese Variable jeweils nur minimal größer wird. ARCADE.PLC setzt daher das *Widening*-Verfahren ein, das

bereits von Cousot & Cousot [12] vorgestellt wurde und das seitdem breite Anwendung erfährt [11].

Die Grundidee des Widenings ist es, die Fixpunktiteration zu beschleunigen, indem eine Vereinigung wie im Algorithmus in Zeile 4 nicht mehr beliebig oft durchgeführt wird. In Experimenten hat sich eine Schranke von 5 Vereinigungen an einem Zustand als effiziente Lösung etabliert. Ist sie überschritten, wird der oder den betroffenen Variablen der Wert $\top$ zugewiesen. Durch dieses Verfahren werden nicht beliebig viele Vergrößerungen der (abstrakten) Wertemengen erlaubt, wodurch der Worklist-Algorithmus in jedem Fall terminiert. Ein Nachteil dieser Lösung ist jedoch eine möglicherweise weniger genaue Überapproximation im Ergebnis der Berechnung. ARCADE.PLC umfasst noch weitere Optimierungen zur Steigerung der Genauigkeit, zum Verringern des Berechnungsaufwandes und zum Sparen von Speicher, wozu auch die Sparse-Analyse gehört, die in Abschnitt 4.2.1 beschrieben wurde. Hierfür und für eine detaillierte Beschreibung der Algorithmen sei auf [3] verwiesen.

4.4.3 Konzept zur Beschleunigung der Wertemengenanalyse

Die Erkenntnisse aus 4.4.1 legen es nahe, die Wertemengenanalyse zu optimieren. In diesem Abschnitt wird ein Konzept vorgestellt, welches die fortschreitende Entwicklung des ST-Programms für eine Beschleunigung der Analyse ausnutzt. Es wird angenommen, dass ein Programm bereits analysiert wurde und die Ergebnisse der VSA dazu vorliegen. Wird diese Version des Programms nun Version n genannt, sollen Ergebnisse aus der VSA genutzt werden, um diese in der nächsten Version $n+1$ des Programms nicht neu berechnen zu müssen. Unter der Annahme, dass das Programm in kurzen Zeitabständen neu analysiert wird um die angezeigten Ergebnisse zu aktualisieren, ändern sich von Version n zu $n+1$ jeweils nur einzelne Instruktionen.

Diese geänderten Instruktionen können die berechneten Wertemengen aus der VSA beeinflussen, sodass diese neu berechnet werden müssen. Eine einzelne, geänderte Zuweisung kann einen nur sehr begrenzten Einfluss haben und nur die Neuberechnung der Wertemengen einer Variable notwendig machen. Durch Abhängigkeiten wie veränderte Bedingungen oder weitere Zuweisungen können jedoch auch noch weitere Neuberechnungen notwendig werden. Der folgende Ansatz besteht daher darin, die Unterschiede zwischen n und $n+1$ zu finden, dann den Einfluss der Änderungen zu berechnen und schließlich die notwendigen Neuberechnungen durchzuführen.

Diese Vorgehensweise bringt dann einen zeitlichen Vorteil, wenn sich das Programm von Version n zu $n+1$ nicht wesentlich verändert hat. Im geplanten Anwendungsfall der kontinuierlichen Annotation von SPS-Programmen ist dies gegeben. Das Verfahren soll inkrementelle Analyse genannt werden, da sich das SPS-Programm in kleinen Schritten ändert und damit schrittweise die Analyseergebnisse angepasst werden.

Für die Umsetzung einer solchen inkrementellen Analyse werden einige Informationen benötigt, die zum Teil schon vorliegen, zum Teil jedoch erst neu berechnet werden müssen. Die folgenden Informationen werden in diesem Konzept benötigt:

- Die Analyseergebnisse aus Analyse von Programmversion n.
- Ein Vergleich zwischen Version n und $n+1$ für die Information über gelöschte und hinzugefügte Instruktionen.
- Alle Instruktionen, die durch solche beeinflusst werden, die von Version n zu $n+1$ gelöscht wurden.
- Eine Zuordnung von Instruktionen aus Programmversion n zu Programmversion $n+1$.
- Alle Instruktionen, die durch solche beeinflusst werden, die von Version n zu $n+1$ neu hinzugekommen sind.
- Die VSA-Ergebnisse für die neu zu berechnenden Wertemengen.

Zunächst wird das Programm mittels einer konventionellen VSA analysiert oder die Ergebnisse liegen aus einem früheren Lauf der inkrementellen Analyse bereits vor. Wie in Abschnitt 4.4.1 untersucht wurde, haben die notwendigen Schritte im Analyseprozess vor der eigentlichen Analyse keinen wesentlichen Einfluss auf die Berechnungszeit. Dies umfasst das Parsen, Generieren der IR und Konstruktion des CFAs, die sowohl für Programmversion n wie zuvor generiert wurden als auch in $n+1$ normal generiert werden.

Ein Vergleich der beiden Versionen ist für den weiteren Verlauf wichtig, um die Unterschiede auswerten zu können. Ansätze zum Vergleichen von Quellcode gibt es bereits seit mehreren Jahrzehnten. Unter ihnen ist das Unix-Programm *diff* mit dem auf Myers [34] zurückzuführenden Algorithmus der Bekannteste. Im Anwendungsfall hier bietet es sich jedoch an, den Vergleich nicht auf Text- bzw. Zeichenebene vorzunehmen, wie es bei den Algorithmen üblicherweise gemacht wird, sondern durch Betrachtung der Programme in ihrer IR.

Würde der Vergleich auf Ebene der Texteingabe gemacht, müssten alle syntaktischen Änderungen verarbeitet und ihr Einfluss auf das Programm überprüft werden. Ebenso verhält es sich mit dem AST. Wird der Vergleich auf IR-Ebene durchgeführt, können die Änderungen bereits abstrahiert von der Syntax des Programms betrachtet werden und die Zuordnung zwischen Instruktion und CFA ist direkt gegeben. Syntaktische Änderungen wie das Hinzufügen von Leerzeilen oder das Umbrechen einzelner Zeilen im ST-Code sind in der Darstellung in IR abstrahiert und würden so keine neu-Analyse des Programms nötig machen.

Abbildung 4.19 zeigt zwei Programmversionen, die fortan als Beispiel genutzt werden sollen. Zeile 3 der Ausgangsversion wird in eine Assign-Instruktion übersetzt, die in Version $n+1$ nicht mehr vorhanden ist. Der Kommentar in Zeile 2 der geänderten Version kommt dagegen in der IR des Programms gar nicht vor und stellt somit keine Änderung dar.

Die beiden Programmversionen werden daher in ihrer IR-Form miteinander verglichen. Das dafür zu lösende algorithmische Problem wird als *Longest-Common-Subsequence*-Problem (LCS) bezeichnet [10, 2]. Im Grundsatz wird beim LCS-Problem eine Zuordnung zwischen zwei Listen vergleichbarer Objekte gesucht, sodass unter Einhaltung der Reihenfolge möglichst viele Objekte einander zugeordnet werden können. Nicht zugeordnet

```
1  A := 1;
2  B := 2;
3  A := 3;
4  C := A;
5  IF C>2 THEN
6     OUT := 1;
7  END_IF;
```

```
1  A := 1;
2  (*Kommentar*)
3  B := 2;
4  C := A;
5  IF C>2 THEN
6     OUT := 1;
7  END_IF;
```

Abbildung 4.19: Zwei Versionen eines Programms. In Ausgangsversion n (links) und geänderter Version $n+1$ (rechts) sind Unterschiede hervorgehoben.

werden dann genau die Objekte, die nur in einem der beiden Listen an den jeweiligen Stellen vorkommen. Die Änderungen an einem Programm werden dafür in zwei Klassen unterteilt: hinzugefügte und gelöschte Instruktionen. Mit diesen beiden Aktionen können alle Modifikationen des Programms von einer Programmversion n zur nächsten Version $n+1$ dargestellt werden. Hinzugefügte und gelöschte Instruktionen, sowie eine Zuordnung der gleich gebliebenen Instruktionen sind die Ausgabe des Algorithmus. Der hier verwendete Algorithmus wird in Abschnitt 4.4.4.1 beschrieben.

Wird eine Zeile oder Instruktion geändert, beschreibt der Algorithmus dies über Löschen und Hinzufügen einer Instruktion. Eine Erkennung einer reinen Änderung könnte weiter untersucht werden, um nur dadurch verursachte Auswirkungen zu untersuchen. Dies eröffnet Potenzial für nachfolgende Maßnahmen zur Effizienzsteigerung, wurde jedoch aus Komplexitätsgründen hier nicht weiter verfolgt.

Wurden die hinzugefügten und gelöschten Instruktionen durch den LCS-Algorithmus identifiziert, müssen die betroffenen und im CFA annotierten Wertemengen invalidiert bzw. bei einer nächsten Berechnung neu berechnet werden. Gelöschte und hinzugefügte Instruktionen werden hierzu jeweils getrennt betrachtet. Erstere hatten im CFA der Programmversion n einen Einfluss auf das Analyseergebnis, der nun nicht mehr besteht. Neben der Instruktion und dem nachfolgenden Zustand selbst müssen daher auch Wertemengen invalidiert werden, die davon direkt oder transitiv abhängig sind. Konkret bedeutet dies, dass beim Beschreiben einer Variable im darauf folgenden Programm nach lesenden Zugriffen auf diese Variable gesucht werden muss, also Instruktionen, bei denen der Bezeichner in einem Ausdruck vorkommt. Handelt es sich dabei um eine Zuweisung zu einer anderen Variable, so ist diese transitiv abhängig und ihre Werte müssen im nachfolgenden Programm ebenfalls invalidiert werden.

Im Beispiel 4.19 kann dies nachvollzogen werden. Durch das Löschen von Zeile 3 besitzt die Variable `A` im nachfolgenden Programm nicht mehr den Wert 3, was in der folgenden Zeile eine Rolle spielt. `C` ist direkt abhängig von `A`. In dem Beispielprogramm ist die Wertemenge für `A` nach der weggefallenen Zuweisung von `A` aus Zeile 3 nicht mehr gültig und die Wertemengen für `C` sind nach Zeile 4 ebenfalls nicht mehr gültig. Die Abhängigkeiten setzen sich fort, da auch die Bedingung in Zeile 5 und damit die Zuweisung in Zeile 6 neu geprüft werden müssen. In diesem Beispiel beeinflusst die gelöschte Zeile also vier Zeilen bzw. Instruktionen in IR. Eine geänderte Zuweisung in Zeile 1 oder 2 hätte dagegen

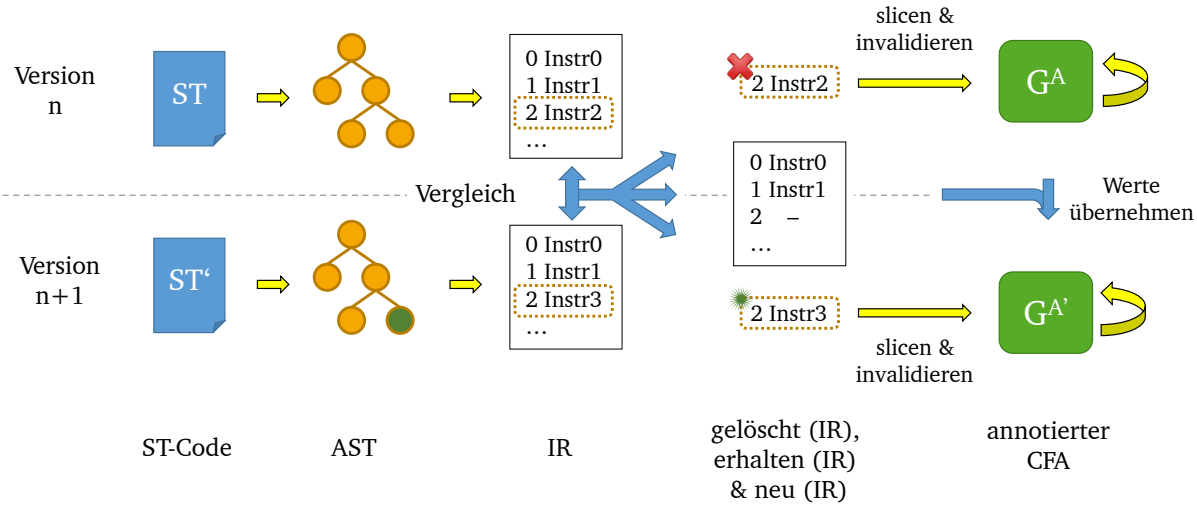


Abbildung 4.20: Verfahren zur Übernahme von Analyseergebnissen aus Programmversion n in $n+1$ (frühere Version der Abbildung erschienen in [37], © 2017 IEEE)

keine Auswirkungen auf das nachfolgende Programm. Lediglich die Wertemenge für die Instruktion bzw. die Zeile selbst hätte neu berechnet werden müssen.

Für diese Abhängigkeitsanalyse unterstützt ARCADE.PLC bereits die Technik des *Slicings*, die auch in Abschnitt 2.3 kurz eingeführt wurde. Mithilfe des *Forward-Slicings* lassen sich die benötigten direkten und transitiven Abhängigkeiten innerhalb des CFAs berechnen. Für die zum Slicing notwendinge Reaching Definition Analyse und zum Erstellen des Abhängigkeitsgraphen muss die VSA noch nicht ausgeführt worden sein. Auf dem für $n+1$ generierten Automat können diese Analysen daher ausgeführt werden, um dann analog zum Vorgehen bei gelöschten Zeilen den Einfluss neuer Instruktionen zu berechnen. Die so gefundenen Wertemengen dürfen dann nicht aus dem CFA von n übernommen werden. Die übrigen Wertemengen nicht betroffener Knoten und Variablen können dann aus dem alten in den neuen Automaten übernommen werden.

Im Ergebnis hat dieser neue CFA alle Wertemengen annotiert, die nicht durch Änderungen von geänderten Instruktionen beeinflusst werden konnten. Auf diesem (vor)annotierten CFA wird schließlich die Wertemengenanalyse ausgeführt. Sie kann ihren Fixpunkt nun schneller erreichen, was an der Konstruktion des Worklist-Algorithmus liegt. Algorithmus 2 wird also mit initialisierten $s_0 ... s_n$ gestartet, also mit eben jenen nicht beeinflussten Variablenmengen. Für das Finden eines Fixpunktes sind hier die Zeilen 4 und 5 wichtig. Dort werden die neu berechneten abstrakten Mengen des nächsten Zustandes mit den dort bereits bekannten Mengen vereinigt ($s_j^{neu} = s_j \sqcup s'_j$). In Zeile 5 wird dann mit $s_j^{neu} \sqsubseteq s_j$ überprüft, ob diese Berechnung tatsächlich neue Werte der Menge hinzugefügt hat oder nicht. Dies wird mit den initialisierten Mengen dann nur bei den Zuständen der Fall sein, bei dem die Wertemengen aufgrund von Änderungen an den Instruktionen invalidiert werden mussten. Diese werden dann neu berechnet.

Eine pauschale Aussage über die Einsparungen oder den verbleibenden Berechnungsaufwand lässt sich so nicht treffen, da dieser maßgeblich vom Einfluss der beeinflussten Zeilen

abhängt. Bei umfangreichen Programmen, die nicht stark zusammenhängen, ergibt sich jedoch ein Einsparungspotenzial.

Das hier erarbeitete Konzept ist in Abbildung 4.20 im Überblick dargestellt. Zusammengefasst werden die Informationen aus einer früheren Programmversion n damit auf folgende Weise übernommen:

1. Programmversion n liegt bereits analysiert vor
2. Ein Vergleich des Programms in Form der IR ergibt drei Arten von Instruktionen: Entfernte Instruktionen von n nach $n+1$, gleich gebliebene Instruktionen in beiden Programmen und in $n+1$ hinzugekommene Instruktionen.
3. Mithilfe des Slicings wird der Einfluss gelöschter Instruktionen im alten CFA berechnet, um dort die Wertemengen zu invalidieren.
4. Nicht betroffene Mengen werden in neuen CFA kopiert.
5. Dort wird mithilfe des Slicings der Einfluss neuer Instruktionen berechnet und ebenfalls entsprechende Wertemengen invalidiert.
6. Auf diesem (vor)annotierten CFA werden die üblichen Analysen inkl. VSA ausgeführt.

Auf diesem annotierten CFA können dann wie in der normalen Statischen Analyse die Prüfungen für Warnungen und Hinweise angewendet werden, mit deren Hilfe Annotationen im Quellcode des SPS-Programms erstellt werden. Die Umsetzung dieses Verfahrens ist im folgenden Abschnitt kurz beschrieben, bevor dann die auf diese Weise erreichten Berechnungszeiten mit der normalen VSA verglichen werden.

4.4.4 Implementierung

Das Verfahren zur inkrementellen Analyse, wie es in Abschnitt 4.4.3 beschrieben wurde, ist im Rahmen dieser Arbeit umgesetzt worden. Die inkrementelle Analyse wurde unabhängig von der verwendeten grafischen Oberfläche oder der Anbindung über die Webschnittstelle gehalten und ist vollständig im Back-End von ARCADE.PLC implementiert.

Zentraler Bestandteil ist dafür die Klasse `IncrementalPlcStaticAnalyzer`, die sich von der Klasse `PlcStaticAnalyzer` ableitet, über die bisher die Statische Analyse und die VSA aufgerufen wurden. Die Klasse `PlcStaticAnalyzer` umfasst bereits viele der nötigen Datenstrukturen inkl. des CFAs und den dazu gehörenden Annotationen. Der neu erstellten Klasse wird zur Initialisierung die bisherige Instanz von der Analyse des Vorgängers, also der Programmversion n übergeben, sofern diese vorliegt. Das Parsen in den AST, generieren der IR und des CFAs muss in dieser Klasse nicht aktiv gesteuert werden, da diese Datenstrukturen zum Zeitpunkt der Instanziierung bereits angelegt wurden. Mit dem Zugriff auf die Instanz der früheren Programmversion n ist dann hier der Vergleich zwischen den beiden Programmen anhand ihrer Repräsentanz in IR möglich.

Der Vergleichsalgorithmus für die beiden Programmversionen wird separat in Abschnitt 4.4.4.1 behandelt. Seine Eingabe sind die Listen mit Instruktionen von Programm n und $n+1$. Ausgabe sind die folgenden drei Datenstrukturen:

1. Liste mit neuen Instruktionen, die in n nicht enthalten waren.
2. Liste mit entfernten Instruktionen, die in $n+1$ nicht mehr enthalten sind.
3. Die Zuordnung von Instruktionen aus n zu gleichen Instruktionen in $n+1$ in Form einer HashMap.

Das Ändern einer Instruktion wird vom Algorithmus durch ein Löschen und Hinzufügen abgebildet und muss daher nicht separat betrachtet werden. Es gibt in der von ARCADE eingesetzten IR keine syntaktischen Strukturen zur Verschachtelung von Instruktionen – dies ist lediglich bei Ausdrücken durch Klammerung möglich, die innerhalb einer Instruktion genutzt werden. Sprünge für Bedingungen sind durch aufsteigende IDs im Programm realisiert, die einem Programmzähler in der Assemblerprogrammierung entsprechen. Entsprechend linear kann der Vergleich der Programme hier auf zwei Listen vorgenommen werden, ohne dabei Kontexte berücksichtigen zu müssen.

Die Liste der Zuordnungen zwischen den Programmversionen n und $n+1$ wird genutzt, um zwischen den CFAs einen semantischen Zusammenhang herstellen zu können. Auf diese Weise kann zu einer Instruktion jeweils die zugehörige Transition des CFAs gesucht und mit der Kante des anderen Graphen assoziiert werden. Unter Verwendung der so gewonnenen Zuordnung werden später auch die abstrakten Wertemengen an den Zuständen von Automat zu Programm n zum neuen Automat übernommen. Eine zusätzliche Berechnung eines Automaten/Graph-Matchings ist somit nicht nötig.

Für den Algorithmus in Abschnitt 4.4.4.1, ist es wie bei anderen Algorithmen zum Lösen des LCS-Problems erforderlich, dass die Elemente der Liste, in diesem Fall Instruktionen, vergleichbar sind. Die Java-Objekte konnten hier nicht komplett auf Gleichheit verglichen werden, da sie Verweise auf den AST und damit den Quelltext enthalten. Es wurde daher ein Vergleich für Instruktionen implementiert, der sich ausschließlich an der Semantik von Instruktionen, Ausdrücken und Struct-Datentypen orientiert. Diese Vergleichsmethode deckt alle in der IR generierten Instruktionen ab und erlaubt so eine einfache Formulierung des Algorithmus.

Wurden die Listen und Zuordnungen berechnet, wird mithilfe des Slicing-Algorithmus analysiert, welche Zustände in den Automaten von den jeweiligen Änderungen beeinflusst werden. Es kommt dabei das Forward-Slicing zum Einsatz, welches eben diese Abhängigkeiten untersucht. Eingabe für den Algorithmus sind Transitionen des Automaten, deren Einfluss untersucht werden soll. Diese können aus den beiden Listen des Vergleichsalgorithmus verwendet werden. Ausgabe ist eine Menge an Zuständen des CFAs, die durch die übergebenen Instruktionen beeinflusst werden. Genauer formuliert sind es Zustände nach Transitionen, in denen betroffene Variablen lesend genutzt werden. Im Automat müssen allerdings auch die Wertemengen am Zustand vor der Instruktion betrachtet werden, da diese im Worklist-Algorithmus der VSA zur Berechnung des so gemeldeten Wertemengen verwendet werden. In allen betroffenen Zuständen werden die berechneten abstrakten Wertemengen mit dem Initialwert versehen. Eine detaillierte Beschreibung des Slicings soll an dieser Stelle nicht erfolgen, da die Umsetzung nicht Teil dieser Arbeit war.

Nachdem auf diese Weise die Wertemengen des CFAs von Programmversion n teilweise invalidiert wurden, werden die übrigen Wertemengen mithilfe der Zuordnung von Instruk-

tionen aus n zu $n+1$ in den neuen CFA übertragen. Dieser enthält nun die abstrakten Wertemengen der Vorgängerversion ohne solche Mengen, die von gelöschten Instruktionen beeinflusst wurden. Auf diesem CFA wird nun ebenfalls Slicing für die neu hinzugekommenen Instruktionen ausgeführt und analog mit den Wertemengen verfahren. Resultat ist ein teil-annotierter CFA mit Wertemengen, die durch Änderungen in der aktuellen Programmversion nicht beeinflusst werden konnten. Auf diesem Automat wird schließlich die VSA ausgeführt.

Der grundlegende Worklist-Algorithmus der VSA wurde dafür nicht geändert. Lediglich die Initialwerte wurden geändert, sodass die inkrementelle Analyse selbst keinen Einfluss auf die Korrektheit der berechneten Analyseergebnisse hat.

Die Implementierung der inkrementellen Analyse ist derzeit nicht dafür ausgelegt, dass sich von einer Version des Programms zur Nächsten Variablendeklarationen im Programm ändern. Dies umfasst die Änderung von Datentypen bestehender Variablen sowie das einfache Hinzufügen oder Entfernen von Variablendeklarationen. Diese Änderungen haben Einfluss auf das Speichermodell, mit dessen Hilfe ARCADE die Instruktionen generiert und simuliert, weshalb Änderungen an Variablen derzeit nicht inkrementell verarbeitet werden können. Als Resultat werden solche Änderungen zu Beginn geprüft und ggf. das ST-Programm mit der konventionellen VSA analysiert.

Nachdem die inkrementelle Analyse abgeschlossen ist, wird die Instanz der Analyse-Klasse für die Vorgänger-Programmversion gelöscht. So kann der Garbage-Collector sie aus dem Speicher räumen und es kommt nicht zu einem wachsenden Speicherbedarf, indem schrittweise alle Instanzen von CFA, IR und AST im Speicher verbleiben.

4.4.4.1 LCS-Algorithmus

In diesem Abschnitt wird der im Rahmen dieser Arbeit entwickelte Algorithmus zum Vergleich zweier Programmversionen erläutert. Grundlage dafür sind die Problem- und Lösungsbeschreibungen von Wagner und Fischer [55]. LCS-Algorithmen wurden ursprünglich zum Vergleich zweier Zeichenketten entwickelt [1], um möglichst große übereinstimmende Teil-Ketten zu finden. In der Verwendung hier werden einzelne Instruktionen als Symbole verwendet und zwei Programme als Listen von Instruktionen bzw. Ketten von Symbolen verglichen. Eine Instruktion gleicht dann einer Anderen, wenn sie den gleichen Typ haben, also beide beispielsweise eine Zuweisung sind, und eventuelle Parameter wie linke und rechte Seite einer Zuweisung ebenfalls gleich sind.

Das LCS-Problem ist im Allgemeinen auf beliebig vielen Sequenzen beschrieben, unter denen die längste gemeinsame Teilfolge gesucht werden soll. Für zwei Teilfolgen lässt sich das Problem mittels dynamischer Programmierung lösen [55]. Weitere Optimierungen sind möglich, sofern die Alphabetgröße endlich ist [2]. Dies ist hier jedoch nicht der Fall, da Instruktionen im Prinzip beliebige Parameter bzw. Ausdrücke beinhalten können und diese dadurch jeweils verschiedene Elemente im Alphabet darstellen. Die Vorgehensweise mittels dynamischer Programmierung hat im Algorithmus den Nebeneffekt, dass neben der Länge des LCS und der konkreten Zuordnung der Instruktionen auch Mengen gebildet werden können, die nur in der ersten bzw. der zweiten Menge enthalten und daher nicht zugeordnet werden können. Dieser Teil ist in dieser Form nicht Bestandteil des LCS-Algorithmus, wie

er von Wagner und Fischer beschrieben wurde, sondern wurde für den Einsatzzweck im Rahmen dieser Arbeit ergänzt.

Als Eingabe werden für den Algorithmus zwei Listen mit Instruktionen erwartet, die auf gleiche Teilfolgen untersucht werden. Es wurde zuvor in 4.4.4 motiviert, dass Listen von Instruktionen keiner weiteren Hierarchie unterliegen und die Vergleichbarkeit einzelner Elemente durch die Implementierung nun gegeben ist. Die Reihenfolge der Instruktionen ist für die Semantik des Programms relevant und muss daher erhalten bleiben bzw. beim Vergleich der Listen beachtet werden. Die Instruktionslisten können im mathematischen Sinn daher als Sequenzen oder Folgen betrachtet werden, bei denen die Reihenfolge wichtig ist und Elemente auch mehrfach in der Sequenz vorkommen können. Eine gemeinsame Teilfolge ist eine Zuordnung von Elementen der ersten Sequenz auf die zweite Sequenz unter Beachtung der Reihenfolge. Dabei können sowohl in der ersten als auch in der zweiten Sequenz Elemente nicht zugeordnet sein. Eine *längste gemeinsame Teilfolge* (*engl.*: Longest Common Subsequence (LCS)) ist dann eine der längsten Teilfolgen unter allen möglichen Teilfolgen. Diese Lösung muss nicht eindeutig sein, es kann also für eine Eingabe mehrere gleichlange Teilfolgen geben, die jeweils die das Kriterium der längsten Teilfolge erfüllen.

Auch der Vergleich von Listen, Quellcode oder Zeichenketten im Allgemeinen kann auf dieses Problem reduziert werden, weshalb zur Veranschaulichung als Beispiel hier Zeichenketten verwendet werden. Die Beispielsequenzen „ABCDFGH“und „BACDEGH“haben als gemeinsame Teilfolgen unter anderem „ACD“, „GH“. Die LCS-Lösungen dieser beiden Sequenzen sind „ACDGH“und „BCDGH“. Wie an diesem Beispiel auffällt, muss die Lösung nicht eindeutig sein. Für den Anwendungsfall zum Vergleich von Programmen ist dies jedoch nicht problematisch, worauf später noch eingegangen wird.

Auf den verwendeten Algorithmus 3 soll nun kurz eingegangen werden. Die Verarbeitung der beiden Instruktions-Listen `a` und `b` besteht aus zwei Phasen. Zunächst wird in Zeilen 1-10 eine Tabelle `length` erzeugt, in der die Längen gemeinsamer Teilfolgen gespeichert sind. Der Spaltenindex entspricht dabei einer Eingabe, der Zeilenindex der zweiten Eingabesequenz. Die komplette Spalte an Index 0 und die Zeile mit Index 0 ist mit 0 initialisiert. Die Tabelle wird dann schrittweise nach der Definition von Wagner und Fischer [55] aufgebaut:

$$length[i,j] = \begin{cases} length(a[i-1], b[j-1]) + 1 & \text{wenn } a[i] = b[j] \\ \max(length[i][j-1], length[i-1][j]) & \text{wenn } a[i] \neq b[j] \end{cases}$$

Ein Eintrag an Stelle $length[i,j]$ entspricht der Länge einer längsten gemeinsamen Sequenz bis zur Position i der ersten Teilsequenz a und zur Position j der Teilsequenz b. Es werden also stets Präfixe der Sequenzen betrachtet. Auf diese Weise kann die Tabelle schrittweise aufgebaut werden, was es zu einem typischen Anwendungsfall der dynamischen Programmierung macht. Im zweiten Schritt, im Algorithmus 3 ab Zeile 11, wird die Tabelle mit einem Traceback-Ansatz nach den übereinstimmenden Instruktionen durchsucht. Begonnen wird in der Tabelle rechts unten, was dem maximalen Zeilen- und Spaltenindex entspricht. Von dort aus gibt es drei Möglichkeiten. Ist die aktuelle Länge gleich den Längen in der Zelle direkt darüber oder direkt daneben, so kann auf eine der beiden Felder vorgerückt werden. Die Elemente der Sequenzen wären am aktuellen Punkt unterschiedlich

Daten: Zwei Listen mit Instruktionen: $a[]$ und $b[]$
Ergebnis: Zwei Listen mit Instruktionen: *neu* und $entfernt$ sowie die Hashmap $zuordnung[]$

```
1  Für i = 0, i < a.size(), i++
2      Für j = 0, j < b.size(), j++
3          Wenn a[i] == b[j] dann
4              length[i+1][j+1] = length[i][j] + 1;
5          Ende
6          Sonst
7              length[i+1][j+1] = max(length[i+1][j], length[i][j+1]);
8          Ende
9      Ende
10 Ende
   // Matching erstellen durch Traceback-Ansatz
11 x = a.size()
12 y = b.size()
13 Solange x != 0 oder y != 0
14     Wenn nicht (length[x][y] == length[x - 1][y] oder
15                 length[x][y] == length[x][y - 1]) dann
           // Übereinstimmung gefunden
16         x = x - 1;
17         y = y - 1;
18         zuordnung[a[x]] = b[y];
19     Ende
20     Alternativ wenn y > 0 und (x == 0 oder length[x][y - 1] >= length[x - 1][y])
        dann
21         neu = b[y - 1] + neu; // Fügt neues Element vorne in die Liste ein
22         y = y - 1;
23     Ende
24     Sonst
25         entfernt = a[x - 1] + entfernt; // Fügt neues Element vorne in die Liste ein
26         x = x - 1;
27     Ende
28 Ende
```

Algorithmus 3: Der entwickelte LCS-Algorithmus mit Berechnung gleicher und geänderter Instruktionen

und könnten in die Liste `neu` oder `entfernt` aufgenommen werden. Es wird auf das Feld vorgerückt, das die längere Teilsequenz enthält. Bei Gleichstand wird nach oben vorgerückt. Falls beide dieser Werte kleiner bzw. in diesem Fall ungleich dem aktuellen Zellen-Wert sind, so liegt eine Übereinstimmung in den Sequenzen vor. An dieser Stelle kann die Zuordnung in der Hashmap `zuordnungen[]` gespeichert werden.

Der Algorithmus soll an einem Beispiel verdeutlicht werden. Die beiden Strings aus dem einführenden Beispiel „ABCDFGH“ und „BACDEGH“ werden dafür verwendet. Der erste String wird der Liste `a` mit dem Index `i` zugewiesen, der zweite String der Liste `b` mit dem Index `j`. In der Darstellung hier wird zuerst der Spaltenindex, dann der Zeilenindex für das Speichern in Tabelle `length[i][j]` genutzt. Diese Tabelle length wird durch den ersten Teil des LCS-Algorithmus wie folgt erstellt:

	ø	A	B	C	D	F	G	H
ø	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
A	0	1	1	1	1	1	1	1
C	0	1	1	2	2	2	2	2
D	0	1	1	2	3	3	3	3
E	0	1	1	2	3	3	3	3
G	0	1	1	2	3	3	4	4
H	0	1	1	2	3	3	4	5

Das Traceback-Verfahren startet dann unten rechts in der markierten Zelle und verläuft über alle markierten Zellen bis zur Position oben links. Zu Beginn unten rechts stimmen beide Eingabesequenzen überein. Das „H“wird daher in das Matching der beiden Sequenzen aufgenommen und auf das Feld oben links weiter gerückt. Dort wird analog verfahren, da auch „G“hier übereinstimmt. Im nächsten Eintrag sind beide Sequenzen unterschiedlich. Der default-Fall hierfür entscheidet für die Zelle darüber. Damit wird das „E“von der Sequenz konsumiert und als zusätzliches Element dort in die Liste `neu` als erstes Element aufgenommen. Der nachfolgende Fall ist analog, allerdings wird hier das Zeichen „F“von der Sequenz konsumiert und in der Liste `entfernt` als erstes Element eingetragen. Der Algorithmus fährt entsprechend fort bis er in der Tabelle oben links angekommen ist.

Ein Fall kann in diesem Beispiel noch betrachtet werden - wenn „A“und „B“als jeweils zweites Zeichen der Sequenz verglichen werden. Hier wählt der Algorithmus die Zelle darüber aus, es wird also das „B“als gemeinsame Instruktion zugeordnet. Hier wäre alternativ auch eine Zuordnung auf dem „A“möglich gewesen. Für die spätere Nutzung des Ergebnisses ist es jedoch unerheblich, welche der beiden Lösungen gewählt wird. Wenn A einen Einfluss auf B hat, werden durch das Slicing dort die Wertemengen invalidiert. Analog verhält es sich, wenn B das Ergebnis von A beeinflussen kann. Sind sie unabhängig voneinander, spielt die Reihenfolge der Beiden ohnehin keine Rolle.

Der hier gezeigte Algorithmus zur Lösung des LCS-Problems hat eine algorithmische Laufzeit-Komplexität von $O(2*m*n)$ und benötigt $O(m*m)$ Speicherplatz. Der Platzbedarf wird hier in erster Linie durch die zu bildende Längen-Matrix bestimmt.

In der Praxis benötigt der Vergleich auf den Programmen, mit denen die Messwerte für Tabellen 4.2 und 4.3 erstellt wurden, 0,33 ms für das `Safety`-Programm und 36 ms für das `Random`-Programm. Der Unterschied erklärt sich durch das an sich kurze `Safety`-Programm mit 13 Zeilen, welches interne Bibliotheken für Berechnungen verwendet. Verglichen wird derzeit nur das Haupt-SPS-Programm, von dem aus die Analyse gestartet wird bzw. an dem derzeit mit dem Unterstützungssystem gearbeitet wird. Das `Random`-Programm dagegen besitzt 600 Zeilen ST-Code. Die Laufzeit liegt damit jeweils deutlich unter der Laufzeit des Parsers auf diesen Programmen und fällt auch gegenüber der Laufzeit der Statischen Analysen nicht ins Gewicht.

4.4.5 Evaluation der inkrementellen Analyse

Die Inkrementelle Analyse wurde im Rahmen dieser Arbeit gegenüber der normalen Analyse evaluiert, um einen in Summe positiven oder negativen Einfluss auf die Gesamtlaufzeit des Analyseprozesses quantifizieren zu können. Dafür wurden sowohl im Fall der inkrementellen Analyse als auch bei der nicht-inkrementellen Analyse die Zeit der einzelnen Analyseschritte protokolliert. Die gemessenen Zeiten wurden dabei analog zu den Werten in Abbildungen 4.2 und 4.3 aus Abschnitt 4.4.1 mit der Funktion `System.nanoTime()` gemessen. Die Evaluation wurde jedoch auf einer anderen Hardware durchgeführt und mithilfe der Webschnittstelle bedient, um automatische Messreihen zu ermöglichen.

Ein Desktop-Computer mit einem Core i5 Prozessor mit 3,5 GHz Taktfrequenz sowie 16 GB DDR3-Ram wurde als Evaluationsgerät eingesetzt. Als Betriebssystem wurde ein 64-Bit Arch Linux zusammen mit dem Java 8 OpenJDK eingesetzt, welchem für die JVM 1024 MB-Ram zur Verfügung standen. Die Tests wurden zum Vergleich auch unter Windows 10 durchgeführt, wo sie zwischen 20 und 60 Prozent längere Laufzeiten in einer Oracle-Java-8-Umgebung zeigten. Hier dargestellt sind jedoch die gemessenen Werte aus den Testläufen unter Linux.

Drei Testprogramme sind für die Evaluation genutzt worden. Im Folgenden werden jeweils die gemittelten Messwerte aus 100 Analyseläufen dargestellt.

Die erfassten Werte schlüsseln jeweils auf, wie lange die Berechnung einzelner Teilschritte der Analyse benötigen. Aufgeführt sind die Berechnung des Programmvergleichs (LCS), die Live-Variable- (LVA) und Wertemengenanalyse (VSA) sowie die für das Slicing im inkrementellen Fall nötigen Reaching Definition-Analysen und das Erstellen des Dominator-Trees. Der Vorgang des Slicings ist nicht separat aufgeschlüsselt, da es sich dabei um eine Abfrage bestehender, bereits berechneter Datenstrukturen handelt. Für die konventionelle Analyse sind LCS, RDA und Domtree jeweils nicht notwendig. In der angegebenen Gesamtdauer sind weiter noch die übrigen Analyseschritte enthalten, die in 4.4.1 beschrieben wurden. Zusätzlich wird noch Zeit benötigt, um die Wertemengen zu extrahieren und die JSON-Ausgabe zu berechnen. Auch diese Schritte sind in der Gesamtdauer enthalten.

Da die inkrementelle Analyse nur sinnvoll beurteilt werden kann, wenn tatsächlich Änderungen erfolgt sind, wurden für jedes Programm zwei Versionen analysiert. Sowohl mit der konventionellen VSA als auch mit der inkrementellen Analyse wurden dann beide Programme analysiert und ebenfalls der Mittelwert daraus berechnet. So wurde sichergestellt,

```
1    MyVar := 3;
2    IF ( MyVar > 2 ) THEN
3    MyOutput := 2;
4    ELSE
5    MyOutput := 0;
6    END_IF
```

Abbildung 4.21: Codeabschnitt für Testprogramme zur Evaluation

dass die Änderung in keiner der Analysen zu einer Vereinfachung des Programms geführt hat, die die Analysezeit zugunsten einer der Verfahren verbessert hätte.

Für vergleichbare Ergebnisse wurde den Programmen jeweils der Codeabschnitt in Abbildung 4.21 eingefügt. Die genutzten Variablen sind jeweils vom Datentyp `INT`. Um die Programmbearbeitung zu simulieren wurde in diesem Codeblock jeweils die Variable `MyVar` in Zeile 1 zwischen den Werten 1 und 3 alterniert. Somit wurde der Kontrollfluss in der Bedingung geändert und die Variable `MyOutput` auf ihren entsprechenden Wert gesetzt. Die Ergebnisse der Berechnungen wurden jeweils überprüft.

Das erste Testprogramm ist das `Safety`-Programm aus der eingehenden Untersuchung in Abschnitt 4.4.1. Mit dem zusätzlichen Codeabschnitt aus Abbildung 4.21 ergeben sich die Werte in Tabelle 4.4.

Analyseschritt	**Konventionelle Analyse**	**Inkrementelle Analyse**
LCS berechnen	–	0,10 ms
RDA, DomTree	–	11,74 ms
LVA	2,33 ms	2,87 ms
VSA	139,43 ms	30,05 ms
Gesamtdauer	174,73 ms	79,49 ms

Tabelle 4.4: Zeitmessungen der Analyseschritte auf dem `Safety`-Programm

Das Resultat der gemessenen Werte zeigt eine deutliche Verkürzung der benötigten Analysezeit durch die inkrementelle Analyse. Dies liegt insbesondere daran, dass die VSA bei diesem Programm mit nur etwa einem Fünftel der sonst benötigten Zeit auskommt. Ein Grund dafür ist, dass die im Programm verwendeten Bibliotheksfunktionen, die miteinander in Abhängigkeit stehen, nicht mehr komplett neuberechnet werden müssen. Hierfür waren bisher viele Iterationen des Worklist-Algorithmus nötig. Im Vergleich der Gesamtdauer der Analysen bringt die inkrementelle Analyse hier einen Zeitvorteil von 54,5 %.

Das zweite Programm `Test1000` besteht aus 300 Variablen und 1000 Zeilen zufällig generiertem ST-Code mit arithmetischen Operationen und Blöcken aus `IF`-Anweisungen. Zusätzlich ist wieder der Codeabschnitt aus Abbildung 4.21 eingebaut. Tabelle 4.5 beinhalten die gemessenen Werte für dieses Programm.

Bei diesem Programm zeigt sich in der VSA ebenfalls eine Verkürzung von 51,05 % durch den inkrementellen Ansatz. Allerdings kompensieren andere Analyseschritte hier

Analyseschritt	Konventionelle Analyse	Inkrementelle Analyse
LCS berechnen	–	108,78 ms
RDA, DomTree	–	134,28 ms
LVA	1213,51 ms	1266,90 ms
VSA	1671,02 ms	817,98 ms
Gesamtdauer	4011,63 ms	3460,47 ms

Tabelle 4.5: Zeitmessungen der Analyseschritte auf dem `Test1000`-Programm

den Zeitgewinn, da die Berechnung des LCS und von RDA und DomTree ebenfalls Zeit benötigen. Der größere Zeitfaktor sind bei diesem Programm allerdings der Export aller Variablenwerte via JSON und die LVA, da es viele Variablen gibt, deren Werte über viele Instruktionen hinweg ermittelt werden müssen.

Das letzte Testprogramm `MagnetLifter` entspricht der Musterlösung eines zu entwickelnden Programms für die Nutzerstudie in Kapitel 5. Es verfügt über 40 Variablen des Datentyps `BOOL` sowie 174 Zeilen ST-Code. Zwei Varianten von Veränderungen wurden davon untersucht. Tabelle 4.6 zeigt die Ergebnisse der Messungen unter Verwendung des Codeblocks aus Abbildung 4.21. Für die Werte in Tabelle 4.7 wurde stattdessen eine Variable geändert, die große Teile des Codes unerreichbar macht.

Analyseschritt	Konventionelle Analyse	Inkrementelle Analyse
LCS berechnen	–	3,21 ms
RDA, DomTree	–	5,53 ms
LVA	1,63 ms	1,94 ms
VSA	30,28 ms	5,84 ms
Gesamtdauer	51,30 ms	35,20 ms

Tabelle 4.6: Zeitmessungen der Analyseschritte auf dem `MagnetLifter`-Programm mit kleiner Variation

Analyseschritt	Konventionelle Analyse	Inkrementelle Analyse
LCS berechnen	–	3,28 ms
RDA, DomTree	–	5,01 ms
LVA	1,64 ms	1,91 ms
VSA	15,86 ms	17,49 ms
Gesamtdauer	34,40 ms	46,03 ms

Tabelle 4.7: Zeitmessungen der Analyseschritte auf dem `MagnetLifter`-Programm mit Änderung, die große Codeteile betrifft

Auch bei den Messwerten in Tabelle 4.6 zeigt sich wieder eine positive Auswirkung der inkrementellen Analyse auf die VSA. Es können wieder Wertemengen genutzt werden, die

ein mehrfaches iterieren des Worklist-Algorithmus über den Code überflüssig machen. In der Gesamtdauer verbleibt ein Gewinn an Analysezeit von 31,39 %. Bei der zweiten Art von Variation im gleichen Programm, durch die ein Großteil des ST-Codes unerreichbar wird, ergibt sich ein anderes Ergebnis. Durch die starke Abhängigkeit eines großteils des Codes von der geänderten Variable müssen im Programm viele Wertemengen invalidiert bzw. neuberechnet werden. Es fällt dann in der Analysezeit im inkrementellen Fall auf, dass die Werte aus der vorherigen Programmversion zunächst in den Graph geladen werden müssen, bevor die VSA durchgeführt werden kann. Im Gesamtergebnis ergibt sich so eine 33,82 % längere Analysedauer bei Verwendung der inkrementellen Analyse.

Bei Betrachtung der dargestellten Messwerte fällt auf, dass die LVA in der konventionellen Analyse minimal schneller abgeschlossen ist, als im inkrementellen Fall. Diese Abweichungen wurden untersucht, der Grund hierfür liegt jedoch nicht in zusätzlichem Berechnungsaufwand, sondern liegt im Speichermanagement und Scheduling der eingesetzten Java VM. Durch die zusätzlich zu speichernden Daten auf dem Heap müssen im inkrementellen Fall mehr Objekte zwischengespeichert und gelöscht werden. Auch andere Schritte wie das Extrahieren von Variablenwerten aus dem Graph zeigten sich in Messungen im inkrementellen Fall minimal langsamer als mit der konventionellen Analyse. Das Speicher- und Performance-Analysetool Java VisualVM[11] legte im inkrementellen Fall vermehrt Aktivitäten des Garbage-Collectors und des Schedulers offen, der auf Seiten des ARCADE-Servers die Anfragen und damit die Analysen auf Threads verteilt. Diese systematischen Abweichungen sind somit in der Gesamtdauer enthalten.

4.4.6 Diskussion

In Abschnitt 4.4 wurde die inkrementelle Analyse als Beschleunigung der Wertemengenanalyse motiviert, vorgestellt und evaluiert. Unter der Annahme, dass Quellcode als Momentaufnahme regelmäßig aus dem Editor einer SPS-Entwicklungsumgebung für die Analyse verwendet wird, ergeben sich von einer Version eines Programms zur nächsten analysierten Version lediglich kleine Änderungen am Programm. Haben diese Änderungen nur einen lokalen Einfluss auf das restliche Programm, so kann die inkrementelle Analyse tatsächlich zu einer schnelleren Berechnungsdauer der VSA beitragen.

In der prototypischen Umsetzung des Verfahrens, die im Rahmen dieser Arbeit entstand, konnte so der komplette Analyseprozess um bis zu 50 % beschleunigt werden. Die Evaluation zeigt jedoch auch, dass die zusätzlichen Analysen, die für den Vergleich der Programme in intermediärer Darstellung und die Abhängigkeitsanalyse bzw. das Slicing verwendet werden, einen Mehraufwand darstellen, der sich in ungünstigen Fällen sogar negativ auf die Gesamtdauer der Analyse auswirken kann. Es ist nach derzeitigem Stand ungünstig für die Analyse, wenn kleine Veränderungen im Programmcode die Neuberechnung großer Teile der Analyseergebnisse erforderlich machen. In der Praxis kann dies beim Umgang mit Aktivierungs- oder State-Variablen passieren, sodass große Teile des Codes nicht mehr erreichbar sind. Weiterhin wird es derzeit nicht unterstützt, dass zusätzliche Variablen von einer Analyse zur nächsten hinzugefügt, geändert oder entfernt werden. Es wird dann

[11] Projekt-Webseite: `https://visualvm.github.io/`

automatisch eine konventionelle Analyse vorgenommen, bevor die nächste Version des Programms wieder inkrementell berechnet werden kann.

Diese beiden Einschränkungen kommen in der Praxis vor, sind jedoch keine Indikation, das Verfahren nicht einzusetzen, zumal der Mehraufwand in den getesteten Fällen absolut gesehen noch immer unter einer Sekunde liegt. In Anforderung 8 wurde jedoch gefordert, dass die Informationen im Back-End geeignet schnell berechnet werden sollen, was zu Beginn dieses Abschnitts 4.4 mit einer Sekunde quantifiziert wurde. Die Einhaltung dieser Schranke kann im Allgemeinen bei großen Programmen auch durch die inkrementelle Analyse nicht garantiert werden, wie die Messungen in Tabelle 4.5 belegen.

Weitere Optimierungen am vorgestellten Verfahren sind jedoch möglich. Es ist zu untersuchen, ob eine Vorauswahl der zu analysierenden Transitionen des CFAs im Worklist-Algorithmus ohne Einschränkung der Korrektheit der Analyse getroffen werden kann, um so den initialen Aufwand zu beschränken. In der vorgestellten Implementierung ist immer zumindest eine vollständige Traversierung des CFAs mit der damit verbundenen abstrakten Interpretation der Instruktionen vorgesehen. Auch der Algorithmus zum Export der berechneten Wertemengen und zum Erstellen der JSON-Ausgabe kann weiter optimiert werden. Weiterhin sind auch an anderen Analyseschritten wie der LVA Optimierungen denkbar, da diese in manchen Fällen ebenfalls maßgeblich zur Gesamtdauer des Analyseprozesses beitragen.

Abschließen muss noch auf eine Einschränkung des inkrementellen Verfahrens hingewiesen werden. Abschnitt 4.2.1 beschreibt die Extraktion der Wertemengen aus dem CFA. Dort wird darauf eingegangen, dass Wertemengen im CFA von ARCADE.PLC üblicherweise nur an Zuständen gespeichert werden, an denen die Variable dazu live ist. Auf diese Weise kann bei der Analyse großer Programme Speicher gespart werden. Für die Extraktion der Wertemengen für einen Editor wurde dieses Verhalten jedoch angepasst, um in jeder Zeile des Programmcodes auf alle Variablenwerte zugreifen zu können. Für die inkrementelle Analyse wurde diese Änderung nicht übernommen, da die Wertemengen sonst nicht durch das Slicing im CFA erfasst werden könnten.

Warnungen und Hinweise werden durch die inkrementelle Analyse auf einem SPS-Programm wie zuvor generiert. Wertemengen sind in der Ausgabe ebenfalls enthalten, allerdings nur an den Stellen, an denen Variablen auch tatsächlich im Quellcode vorkommen. Das Feature aus Abschnitt 4.2.3 zum Umgang mit unvollständigen Programmen, die auch zwischen zwei Analyseläufen neue Zeilen annotieren können, ist somit nicht mit der hier implementierten inkrementellen Analyse kompatibel. Daher und aufgrund der für die Nutzerstudie nicht kritischen Laufzeitunterschiede der Analyse wurde die inkrementelle Analyse nicht in der Nutzerstudie verwendet.

Kapitel 5

Evaluation

Bereits in Abschnitt 1.1 wurde die Hypothese H1 vorgestellt, nämlich, dass Informationen aus der Statischen Analyse eines bereits teilweise geschriebenen Programms bei der weiteren Entwicklung hilfreich sind. Um sie zu überprüfen, wurde der implementierte Prototyp auf Basis des Editors Codesys 3.5 von [Müllers, 2018] evaluiert. In diesem Abschnitt wird die Nutzerstudie dieser Abschlussarbeit erörtert und die Ergebnisse interpretiert.

5.1 Hypothesen

Für die Beantwortung der generell gehaltenen Hypothese 1 wurde diese in die folgenden konkreten Teil-Hypothesen zerlegt, die in der Studie untersucht werden sollten:

Hypothese 2 (H2): *Die Anzeige von Informationen aus der Statischen Analyse im Editor wird verwendet und führt zu weniger Fehlern im Programm.*

Hypothese 3 (H3): *Die Anzeige von Wertemengen zu Variablen verkürzt die Einlesezeit in ein existierendes Programm und einem anderen Entwicklungsverhalten.*

Hypothese 4 (H4): *Die angezeigten Wertemengen und Warnungen aus der Statischen Analyse sind für den Programmierer nützlich.*

Diese aufgestellten Teil-Hypothesen untersuchen jeweils einen erwarteten Nutzen, den der entwickelte Prototyp mithilfe der Statischen Analyse einem Programmierer bieten kann. H2 steht für die Vermutung, dass der SPS-Programmierer die angezeigten Warnungen aus der Analyse nutzt, um über die Semantik seines entwickelten Programms zu reflektieren und ggf. Korrekturen vorzunehmen, falls tatsächlich ein Fehler vorliegt. Weiterhin fassen wir unter H2 zusammen, dass die angezeigten Variablenwerte bzw. Wertemengen vom Programmierer genutzt werden um die Semantik des selbstgeschriebenen Programms sofort im Editor prüfen zu können und so weniger weiterführende Tests in einem Simulator oder auf echter Hardware durchgeführt werden müssen. Diese Hypothese ist testbar, indem das Auftreten und die Korrektur von Fehlern während der Evaluation überwacht wird. H3 nehmen wir an, da die Anzeige von Wertemengen auf einem existierenden Quellcode Hinweise über das spätere Laufzeitverhalten geben kann. Es wird erwartet, dass die

zusätzlichen Informationen zu einer anderen Entwicklungsstrategie führen, bei der weniger andere Testmethoden wie die Simulation notwendig sind. Testbar ist diese Methode über die Bearbeitungszeit von Programmieraufgaben sowie über ein Protokoll der gelesenen Annotationen. H4 ist eine offen gestellte Hypothese. Es wurde davon ausgegangen, dass angezeigte Wertemengen und Warnungen als positiv und produktivitätssteigernd wahrgenommen werden, wenn diese bereits beim Schreiben zur Verfügung stehen. Diese subjektiven Wahrnehmung wurde in der Studie mithilfe eines Fragebogens abgefragt.

Jede der aufgestellten Hypothesen stellt an sich bereits einen Vorteil für den SPS-Programmentwickler dar, sodass von diesen Hypothesen auf die grundlegende Hypothese H1 dieser Arbeit geschlossen werden kann.

5.2 Aufbau der Nutzerstudie

Der entwickelten Studie liegt ein experimentelles Design zugrunde, bei dem die Probanden zufällig einer von zwei Gruppen zugeordnet werden. Gruppe A ist dabei die Versuchsgruppe, die den entwickelten Prototyp mit eingeschalteter Unterstützung durch Statische Analyse nutzen konnte. Gruppe B diente als Kontrollgruppe und bekam diese Unterstützung nicht. Da der Editor des Prototyps nicht den gesamten Funktionsumfang des originalen Codesys-Editors bietet, wurde Gruppe B ebenfalls der Editor des Prototyps zur Verfügung gestellt. So wurden Unterschiede resultierend aus dem unterschiedlichen Funktionsumfang der beiden Editoren ausgeschlossen. In diesem Modus des entwickelten Prototyps wurden jedoch weiterhin Warnungen und Meldungen des Codesys-Backends angezeigt. Hierzu zählen zum Beispiel Syntaxfehler und Zuweisungen von inkompatiblen Datentypen, also Meldungen, die durch die Continuous Compilation von Codesys erzeugt werden.

5.2.1 Studienteilnehmer

Die Aussage von Hypothese 1 bezieht sich auf SPS-Programmierer, also auf Personen mit Erfahrung in der Entwicklung von Programmen für Steuerungen. Für die Nutzerstudie wurden daher Anwender gewählt, die diese Erfahrung mitbringen. Studentische und wissenschaftliche Mitarbeiter des Lehrstuhls Informatik 11 wurden lediglich für Probeläufe der Nutzerstudie herangezogen. Ihre Daten flossen nicht in die Auswertung ein. Geeignete Probanden sollten mit den Sprachen der IEC 61131-3 vertraut sein und regelmäßig Codesys oder eine andere Entwicklungsumgebung für speicherprogrammierbare Steuerungen verwenden.

5.2.2 Aufgaben

Für die Nutzerstudie wurden zwei Aufgaben entwickelt. Zu Beginn wurden die Probanden mit einer kleinen Aufgabe an die Nutzung des Prototypen herangeführt. Eine zweite Aufgabe war etwas umfangreicher. Hier sollte ein bereits vorgegebenes Programm erweitert werden.

Aufgabe 1 – SumOrMax In der ersten Aufgabe sollten die Probanden eine einfache Funktion zur Addition zweier Werte und einer zusätzlichen Maximalwertbildung implementieren. Die Eingangs-Variablen `A` und `B` sowie die Ausgangsvariable `OUT` wurden in einem Codegerüst vorgegeben und hatten den Typ `BYTE`. Die Werte von `A` und `B` waren nun zu addieren. Übersteigt die Summe 128, sollte `OUT` konstant der Wert 150 zugewiesen werden. Die Probanden wurden gebeten, den Programmcode für diese Aufgabe in der vorbereiteten Codesys-Umgebung zu schreiben, vorgegebene Variablen als Schnittstelle jedoch nicht zu ändern. Weitere lokale Variablen durften verwendet werden, ebenso wie weitere Funktionen, um beispielsweise Datentypen zu konvertieren.

Aufgabe 2 – MagnetLifter In dieser zweiten Aufgabe war ein Programm zur Steuerung eines Krans mit Magnetgreifer vorgegeben. Dieses Programm bewegte den Kran nach unten, um eine bereitstehende Kiste aufzunehmen, fährt damit zu einer Entladestelle über einem Fließband und fährt nach dem dortigen Absenken und Ablegen wieder in die Ausgangsposition. Eine Grafik erläuterte den Ablauf des vorgegebenen Verhaltens. Zusätzlich stand auf dem Testsystem eine animierte Version zur Erläuterung zur Verfügung. Der für dieses Programm zugrundeliegende Zustandsautomat wurde durch ein `case`-Statement und eine zugehörige Zustandsvariable vorgegeben. In der Aufgabe sollten die Probanden nun zwei zusätzliche Funktionalitäten implementieren. Ein `Enabled`-Eingang vom Datentyp `BOOL` sollte jegliche Bewegung des Krans ein- bzw. ausschalten, jedoch keinen Einfluss auf den Magneten nehmen. Ein weiterer `AV_EmergencyStopOk`-Eingang mit gleichem Datentyp sollte den Kran bei der Belegung mit `FALSE` ebenso anhalten. Das Wiedereinschalten sollte jedoch erst möglich sein, wenn am `Enabled`-Eingang eine steigende Flanke erkannt wurde. Den Probanden wurde der vorgegebene Code bereitgestellt. Sie konnten selbst wählen wie und an welcher Stelle sie die Funktionalität implementieren. Die Verwendung von eigenen lokalen Variablen war wie in Aufgabe 1 erlaubt und für die Flankenerkennung auch nötig. Auch hier sollte die Schnittstelle nach außen nicht geändert werden.

Diese beiden Aufgaben wurden aufgrund ihres Anwendungsbezugs und dem für die Studie verfügbaren Zeitrahmen ausgewählt. Eine kurze Codeimplementierung in einer eigenständigen POE zum Verarbeiten von Werten in Abhängigkeit eines Grenzwertes ist eine typische Aufgabe, die im Zusammenhang mit der Funktionsbausteinsprache entsteht und wenn sich die Funktionalität sonst nur umständlich durch andere existierende Bausteine ausdrücken lässt. Die Nutzung von `BYTE`-Variablen ist in diesem Kontext ebenfalls üblich, da sie bei SPSen zum Zusammenfassen von mehreren digitalen Ein- oder Ausgängen genutzt werden können.

Aufgabe 1 bietet sich mit dieser Aufgabenstellung in der Studie besonders an, da es zu einem Fehlverhalten bei einer zu einfachen Implementierung kommen kann. Werden die beiden Eingangsvariablen zuerst addiert und einer Variable vom Typ `BYTE` zugewiesen, kommt es bei der Ausführung des Codes zu einem Überlauf der zugewiesenen Variable, wenn `A+B` in Summe den Wertebereich von 0–255 überschreiten. Eine anschließende Prüfung auf den Grenzwert von 128 führt so nicht mehr zum gewünschten Verhalten.

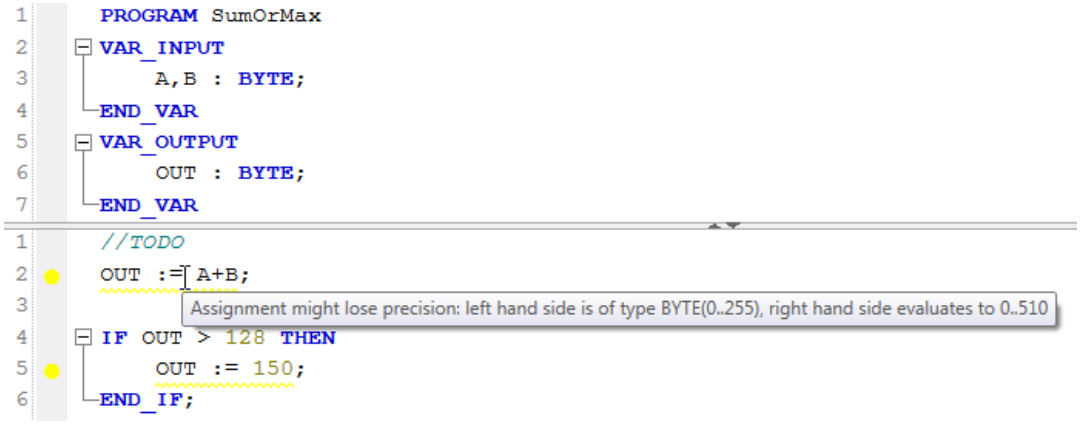


Abbildung 5.1: Eine Warnung weist in Zeile 2 darauf hin, dass die Addition der zwei BYTE-Variablen A+B in Summe größer als der Wertebereich der Variable auf der linken Seite der Zuweisung ist. Die Warnung in Zeile 5 warnt vor der erneuten Zuweisung einer Ausgangsvariable, was hier ein gewolltes Verhalten ist.

Die Statische Analyse durch ARCADE.PLC ist jedoch in der Lage, auf dieses, in diesem Fall unerwünschte, Verhalten hinzuweisen. Die beschriebene Zuweisung wird daher im Prototyp mit einer Warnung versehen. Abbildung 5.1 zeigt diese Annotation im Code mit der problematischen einfachen Implementierung. Für die Studie kann der Proband anhand dieser Aufgabe die Darstellung der Warnungen kennenlernen und aufgrund der geringen Komplexität der Aufgabe auch schnell die Ursache der Warnung nachvollziehen.

Für Aufgabe 2 wurde ein bei SPSen üblicher Zustandsautomat vorgegeben, der eine Schrittkette implementiert. Hier war eine etwas größere Programmieraufgabe vorgesehen, um das Verhalten des Programmierers an einer komplexeren Aufgabe untersuchen zu können. Die komplett eigenständige Implementierung der Schrittkette und des Abschalt-Verhaltens hätte einer ersten Schätzung nach 2–3 Stunden gedauert. Für diese Versuchsdauer standen jedoch keine Probanden zur Verfügung bzw. es gab keine adäquate Aufwandsentschädigung. Es wurden daher nur die Nothalt- und Abschalt-Funktionalität Teil der zu implementierenden Aufgabenstellung. Weiterhin war es so möglich, eine Aussage über Hypothese 3 treffen zu können. Um die Komplexität der Aufgabe weiter gering zu halten, waren die Ein- und Ausgänge binär bzw. vom Datentyp BOOL. Die Anzeige von möglichen Variablenwerten mithilfe des Prototyps war während der Implementierung möglich. Verhalten und Eindrücke der Probanden in dieser komplexeren Aufgabe waren dann Indizien für Hypothese 4.

In Probeläufen wurden mehrere Varianten der Lösung erarbeitet. So ist es möglich, den zusätzlichen Programmcode hauptsächlich am Ende des vorgegebenen Codes zu implementieren oder zwischen diesem Code. Eine Vorgabe, welche Art der Implementierung zu bevorzugen ist, war in der Aufgabenstellung nicht gegeben.

Die Einlese- und Bearbeitungszeit wurde vor der Bearbeitung durch die Probanden mit etwa 10 Minuten für Aufgabe 1 und 30 Minuten für Aufgabe 2 mit Hilfe von zwei Probeläufen abgeschätzt. Damit lagen die Aufgaben im Zeitfenster von etwa einer Stunde, die die Studie inklusive Einführung, Erläuterungen und Fragebogen dauern sollte.

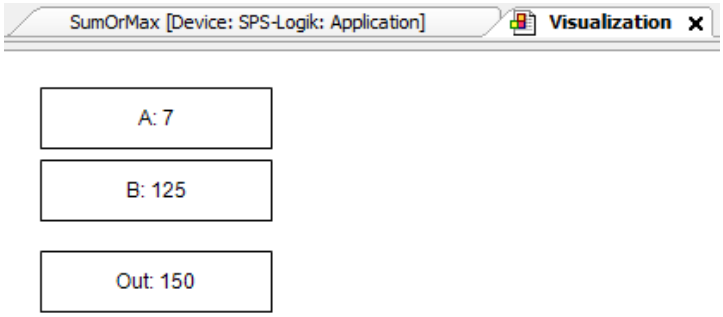


Abbildung 5.2: Visualisierung zu Aufgabe 1 (Abbildung erstellt zur Arbeit von [Müllers, 2018])

5.2.3 Rahmenbedingungen

Die Aufgabenbeschreibung wurden den Probanden zusammen mit den Variablen inkl. der verwendeten Datentypen in Papierform zur Verfügung gestellt. Als Hilfsmittel standen ihnen neben dem Codeeditor in der Codesys-IDE inklusive der Onlinehilfe mit Funktionsreferenz, Papier und Stift für Notizen und der Zugang zum Internet für eventuelle Recherchen zur Verfügung. Die Probanden wurden aufgefordert, keine externen Tools zur Analyse oder Verarbeitung des Quellcodes zu nutzen.

Die Entwicklung von SPS-Programmen erfolgt üblicherweise zunächst in einer Entwicklungsumgebung. Um die Funktionalität des Programms oder einzelner POEs zu überprüfen, wird dieses Programm dann auf die SPS in einer produktiven oder einer Testumgebung übertragen und dort getestet [51]. Eine solche Testmöglichkeit bestand bei der Evaluation nicht, da die konkrete Ausführung des Programms nicht zentraler Untersuchungsgenstand war. Stattdessen wurde eine Simulation genutzt, sodass das Programm in der Codesys-internen Soft-SPS ausgeführt werden konnte.

Die Ein- und Ausgänge des Programms wurden dafür in einer grafischen Ansicht, einer Visualisierung, dargestellt. Abbildungen 5.2 und 5.3 zeigen diese vorkonfigurierten Visualisierungen. Ausgangsvariablen der simulierten Programmausführung wurden für Aufgabe 1 als Zahl oder für Aufgabe 2 in Form von Lampen dargestellt. Eingänge wurden für Aufgabe 1 als Eingabefelder und in Aufgabe 2 als Schalter für boolesche Variablen oder Radio-Buttons für Endlage-Zustandsübermittlungen bereitgestellt.

Diese Visualisierung stand beiden Gruppen, also der Versuchs- und Kontrollgruppe zur Imitation des normalen Entwicklungsprozesses zur Verfügung. Beiden wurden zu Beginn der Aufgabenbearbeitung die Simulation und Visualisierung erklärt. Für Aufgabe 2 wurde zusätzlich der Ablauf des vorimplementierten Programms vorgeführt, damit der Einarbeitungsaufwand gering bleibt.

Um die Funktionsfähigkeit der Visualisierung zu gewährleisten, mussten die Variablenbezeichner der beiden Codegerüste jeweils gleich bleiben. Die Probanden wurden gebeten, daran keine Änderungen durchzuführen. Der originale Codeeditor in Codesys bietet während des Testens die Möglichkeit, einzelne konkrete Variablenwerte der Simulation anzuzeigen und zu manipulieren. Dieses Feature stand beiden Gruppen nicht zur Verfügung.

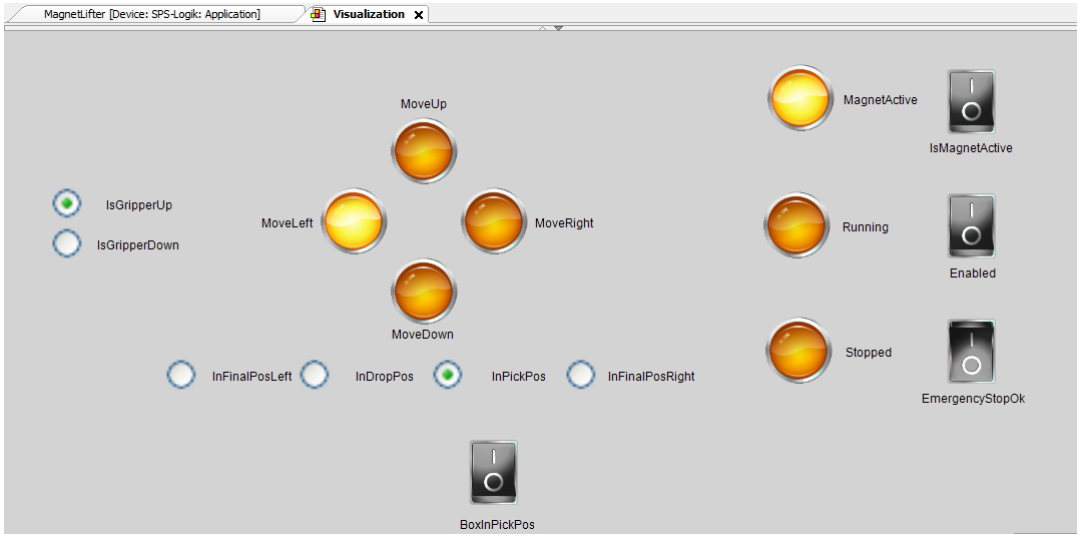


Abbildung 5.3: Visualisierung zu Aufgabe 2 (Abbildung erstellt zur Arbeit von [Müllers, 2018])

Da der Fokus der Studie auf der Untersuchung des reinen Entwicklungszeitraumes liegt und sich die Ergebnisse weiterhin zwischen den Gruppen vergleichen lassen, ist diese Einschränkung für die Studie nicht relevant.

Während der Aufgabenbearbeitungen durften die Probanden Fragen stellen, insbesondere zur Aufgabenstellung und zur Nutzung von Codesys. Auch Tipps zur Bearbeitung der Aufgabe wurden gegeben, wenn ein Proband Schwierigkeiten mit einem Ansatz dafür hatte. Die Testgruppe wurde zu Beginn sehr kurz auf die in ihrem Editor zusätzlich vorhandenen Features hingewiesen und wie die zusätzlichen Informationen angezeigt werden können. Darüber hinaus wurde dieser Gruppe zu diesem Feature keine Hilfestellung geleistet, um das Ergebnis der Studie nicht zu beeinflussen. Speziell zur Interpretation der Meldungen wurden keine Hinweise gegeben. Es wurde kein zeitliches Limit für die Studie vorgesehen, allerdings wurden die Probanden darauf hingewiesen, dass sie die Teilnahme jederzeit abbrechen können, wenn sie bei der Bearbeitung Probleme haben sollten. Von dieser Möglichkeit wurde jedoch kein Gebrauch gemacht.

Die Evaluation wurde auf einem Notebook mit 14"-Bildschirm, einem Intel Core i7 der dritten Generation und 16 GB Arbeitsspeicher durchgeführt. Die Bedienung erfolgte über die eingebaute Tastatur und eine zusätzliche externe Maus.

Alle Eingaben der Benutzer in dem von uns entwickelten Prototyp wurden bei beiden Gruppen durch einen eingebauten Keylogger aufgezeichnet. Darüber hinaus wurde ein Bildschirmvideo während der Bearbeitung aufgenommen. Die Probanden wurden über diese Maßnahmen zuvor mündlich aufgeklärt und gaben dazu eine schriftliche Einverständniserklärung ab. Die Aufzeichnungen wurden genutzt, um später die Hypothesen H2 und H3 untersuchen zu können.

Als Aufwandsentschädigung erhielten alle Probanden nach der Teilnahme an der Studie einen Einkaufsgutschein in Höhe von 10,– Euro.

5.2.4 Fragebogen

Nach der Aufgabenbearbeitung erhielten die Probanden einen Fragebogen um weitere Daten zu erheben. Abgefragt wurden Geschlecht, Alter, Programmiererfahrung und Vorerfahrungen speziell hinsichtlich der SPS-Programmierung und Statischer Analyse. Im weiteren Verlauf wurden die Probanden nach ihren Einschätzungen gefragt, inwiefern sie Annotationen und Meldungen hilfreich fanden oder wie sehr diese genutzt werden konnten, um Fehler zu beheben. An dieser Stelle wurde die Unterscheidung zwischen Annotationen im Code und Meldungen, also Einträgen im Meldungsfenster von Codesys gemacht.Einschätzungen der Probanden wurden jeweils mithilfe einer fünfwertigen Likert-Skala erfasst. Dieser Fragentyp bildet einen anerkannten Standard in Nutzerstudien, bei denen eine quantifizierbare Aussage zu Nutzereinschätzungen abzufragen [30]. Ein Beispiel für ein solches Likert-Item ist:

> "*Wie sehr stimmen Sie der folgenden Aussage zu? Die vom Editor zur Verfügung gestellten Annotationen bei der Bearbeitung waren hilfreich. (1 überhaupt nicht, 5 sehr)*" (aus: [Müllers, 2018])

Der Fragebogen war für beide Gruppen gleich. Die Kontrollgruppe, die keine Unterstützung durch Statische Analyse erhielt, konnte sich an dieser Stelle lediglich auf Meldungen des Codesys-Compilers beziehen, der ebenfalls für Annotationen und Meldungen sorgte.

Beide Gruppen hatten im Anschluss noch die Möglichkeit, den entwickelten Prototyp mit eingeschalteter Unterstützung ohne weitere Aufgabenstellung zu nutzen und frei damit zu experimentieren. Es folgte abschließend noch einmal ein kurzer Fragebogen zur Einschätzung, inwiefern sie die Features einzeln, also nur die Annotationen aus der Statischen Analyse oder nur die Informationen über Wertebereiche der Variablen als nützlich einschätzen. Dieser Fragebogen wurde unabhängig vom Versuchsleiter am eigenen Computer nach dem Versuch durchgeführt. Auf diese Weise bestand für die Probanden Bedenkzeit, um über den Prototyp und das Unterstützungssystem zu reflektieren und eine eventuelle wahrgenommene Erwartungshaltung während des Versuchs konnte so vermieden werden. Beide Fragebögen sind im Anhang A dieser Arbeit angehängt.

5.3 Ergebnisse und Diskussion

Sieben Teilnehmer aus zwei unterschiedlichen Unternehmen konnten für die Nutzerstudie gewonnen werden. Sie waren zwischen 25 und 49 Jahre alt (Mittelwert 31,71 Jahre, Standardabweichung σ =8,01). In Abbildung 5.4 sind die Vorkenntnisse der Teilnehmer dargestellt. Fünf von ihnen gaben ihre wöchentliche Programmiertätigkeit mit mehr als 20 Stunden an und alle Teilnehmer sind mit den Sprachen der IEC 61131-3 vertraut. Generell schätzen jedoch alle ihre Programmiererfahrung als durchschnittlich bis überdurchschnittlich ein. Zwei Probanden arbeiten hauptsächlich mit Siemens STEP 7 bzw. dem TIA-Portal und haben wenig Erfahrung mit Codesys. Bei den fünf Anderen ist der Kenntnisstand genau andersherum - sie sind auf Codesys spezialisiert. Darüber hinaus hatten fast alle Erfahrungen mit Visual Studio, einige auch mit Eclipse. Dies spiegelt sich auch in den

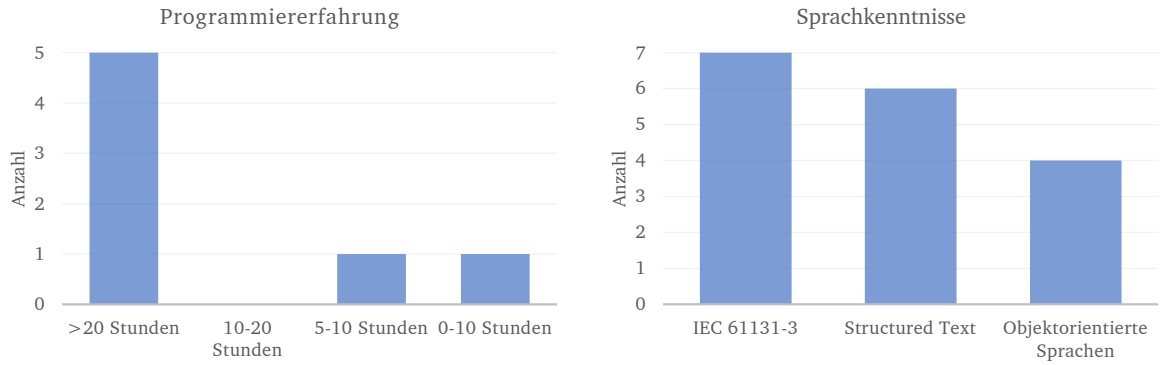


Abbildung 5.4: Vorerfahrung der Probanden – links die wöchentliche Programmierpraxis, rechts die Programmiersprachen-Kenntnisse (Daten erhoben in [Müllers, 2018])

Programmiersprachen wieder. Sie alle beherrschen Sprachen der IEC 61131-3 und Structured Text im Speziellen. Darüber hinaus kennen sich vier Teilnehmer mit objektorientierter Programmierung aus.

Nachdem die Probanden die Aufgaben bearbeitet hatten, wurden sie im Fragebogen nach der Schwierigkeit der zu lösenden Aufgabe gefragt. Auf einer Skala von 1 bis 5 wurden die zwei Aufgaben im Mittel mit einer 2 mit einer Standardabweichung σ von 0,75 bewertet. Ein Zusammenhang zwischen der Bewertung der Aufgabe und der Selbsteinschätzung wäre hier zu erwarten gewesen, allerdings sprechen die Antworten der Teilnehmer teilweise dagegen.

Von den Probanden wurden vier zufällig der Testgruppe zugeteilt. Die übrigen drei waren in der Kontrollgruppe ohne Unterstützung durch Statische Analyse. Sie alle absolvierten die Studie komplett und ihre Daten konnten ausgewertet werden. Auch technische Probleme traten während der Durchführungen nicht auf. Der geringe Stichprobenumfang erlaubt jedoch keine repräsentativen Aussagen. In der folgenden Auswertung werden daher Erfahrungsberichte aus der Studie [Müllers, 2018] präsentiert und diskutiert.

5.3.1 Beobachtungen

Die während der Aufgabenbearbeitung aufgenommenen Bildschirmvideos, Logdateien und erarbeiteten Lösungen wurden gesammelt und ausgewertet. Auf diese Weise soll Hypothese 2 untersucht werden, also ob durch durch die Anzeige von Informationen der Statischen Analyse weniger Fehler im Programm gemacht werden und ob diese tatsächlich genutzt werden. Dies betrifft sowohl die angezeigten Wertemengen als auch die Warnungen und Hinweise der Statischen Analyse, durch die mögliche Fehlerquellen im SPS-Programm aufgedeckt werden. Auch Hypothese 3 soll mithilfe der gesammelten Daten untersucht werden, also ob die angezeigten Wertemengen die Einlesezeit des Programmierers verkürzen. Bei dieser Hypothese sind zusätzlich die Antworten der Fragebögen relevant, um die subjektive Einschätzung zu untersuchen.

Ergebnis	Testgruppe	Kontrollgruppe
Direkt fehlerfrei gelöst	3	1
Nach Meldung der Statischen Analyse korrigiert	1	×
Mit Simulator korrigiert		
Fehler nicht gefunden		2

Tabelle 5.1: Bearbeitungsergebnisse zu Aufgabe 1 (Daten erhoben in [Müllers, 2018])

Um mit vertretbarem zeitlichen Aufwand Aussagen über Programmierfehler treffen zu können, wurde Aufgabe 1 als Einstieg in die Evaluation konstruiert. Diese Aufgabe verleitet dazu, die maximal möglichen Wertebereiche der vordefinierten Variablen falsch einzuschätzen und damit einen semantischen Fehler zu erzeugen. Durch die Unterstützung mit ARCADE.PLC werden in diesem Fall Warnungen im Quellcode annotiert, die die Testgruppe nutzen konnte, um den Fehler zu beheben. Der Kontrollgruppe stand dieses Mittel nicht zur Verfügung, da Codesys selbst diese Warnung nicht ausgibt. Abbildung 5.1 zeigt beispielhaft eine Lösung der Aufgabe mit einem solchen Fehler und der Annotation durch die Statische Analyse.

In der Durchführung des Versuchs implementierten vier Probanden die Aufgabe auf Anhieb korrekt und waren damit nicht auf ein Unterstützungssystem angewiesen. Sie überprüften im Anschluss ihr Ergebnis in der Simulation mit der Visualisierung. Zwei Probanden der Kontrollgruppe und einem Probanden der Testgruppe unterlief wie vorgesehen der Fehler im Bezug auf die Wertebereiche. Der Proband aus der Testgruppe wurde durch die Meldungen der Statischen Analyse auf den Fehler hingewiesen und behob ihn anschließend, wie auf dem Bildschirmvideo sichtbar ist. Ob hierfür jedoch die Anzeige im Meldungsfenster ausschlaggebend war oder die Annotation im Quellcode, kann nicht zweifelsfrei bestimmt werden. Insgesamt wurden die Meldungen in den Annotationen über ein überfahren mit der Maus in der Testgruppe bei Aufgabe 1 nur wenige Male verwendet.

Die zwei Probanden der Kontrollgruppe, die den Fehler ebenfalls machten, hatten zur Fehlersuche lediglich die Simulation zur Verfügung. Sie beide nutzten diese Möglichkeit zwar, entdeckten im simulierten Verhalten jedoch keinen Fehler. Er blieb daher in ihrer Lösungsversion jeweils enthalten. Tabelle 5.1 fasst diese Beobachtungen zusammen.

Für Hypothese 2 wäre es auch interessant, ob die Unterstützungsfunktionen einen Vorteil im Bezug auf die Bearbeitungszeit bringen können. Da Aufgabe 1 jedoch die erste zu bearbeitende Aufgabe war und die Bearbeitungszeit ohnehin relativ kurz ausfiel, sind Aussagen darüber in diesem Rahmen nicht möglich. Die Testgruppe benötigte für die Bearbeitung 5:22 Minuten (σ = 1:45 min), während die Kontrollgruppe lediglich 2:52 Minuten (σ = 0:49 min) benötigte. Es ist jedoch davon auszugehen, dass die längere Bearbeitungszeit der Testgruppe auf das Kennenlernen des Prototyps zurückzuführen ist, da während des Versuchs hierzu auch Fragen gestellt wurden. Weiterhin sollte beachtet werden, dass die Kontrollgruppe die Aufgabe zwar im Mittel schneller bearbeitete, zwei der drei Lösungen jedoch einen Fehler enthielten.

Auch Aufgabe 2 wurde auf gleiche Weise in der Nachbearbeitung untersucht. Die Schwierigkeit dieser Aufgabe bestand in erster Linie darin, die spezifizierte Funktionalität um-

zusetzen. Dies gelang jedoch allen Teilnehmern mit den jeweils verfügbaren Mitteln. Programmierfehler wie in Aufgabe 1 tauchten während der Bearbeitung nicht auf, da die verwendeten Variablen hauptsächlich den Datentyp `BOOL` besaßen. Generell traten in der Testgruppe nur sehr wenige Fehler auf, durch die Warnungen und Hinweise von ARCADE.PLC generiert wurden.

Die Ergebnisse aus der Studie zu Aufgabe 2 lassen daher keine Rückschlüsse auf Hypothese H2 zu, also ob weniger Fehler durch die Unterstützungsfunktionen gemacht werden. Wohl aber lässt sich aus den Logdateien ablesen, ob Annotationen im Programmcode zum Ablesen der Wertemengen genutzt wurden. Erfasst wurde dabei jeweils, wenn eine Annotation mit der Maus überfahren wurde, um sie im Popup-Fenster zu lesen. Unterschieden wurde zwischen Meldungen von Codesys aus der Übersetzung des Programms und ARCADE-Meldungen aus der Statischen Analyse.

Abbildung 5.5 zeigt den Bearbeitungsverlauf für alle Teilnehmer. Die Probanden sind von 1-7 nummeriert, wobei 1, 3 und 5 in der Kontrollgruppe nur die Meldungen des Codesys-Compilers nutzen konnten während 2, 4, 6 und 7 als Teilnehmer der Testgruppe die Meldungen aus der Statischen Analyse nutzen konnten. In der Abbildung sind die Probanden entsprechend gruppiert. Direkt auf den Linien sind das Schreiben von neuen Zeilen sowie das Korrigieren vorhandener Zeilen aufgetragen. Jeweils leicht versetzt darüber zeigen unterschiedliche Markierungen an, wenn im Verlauf der Bearbeitung Annotationen gelesen wurden. Nicht explizit in der Grafik erfasst sind die Abschnitte, in denen die Simulation mit Visualisierung zum Testen der Implementierung genutzt wurde. Diese Phasen fanden in der Regel in den Pausen statt, in denen keine Modifikationen am Programmcode vorgenommen und keine Annotationen angeschaut wurden.

Nicht überraschend ist die messbar stärkere Nutzung von Annotationen durch die Testgruppe im Vergleich zur Kontrollgruppe, da bei letzterer lediglich Compilerwarnungen zur Verfügung standen. Die Probanden der Testgruppe nutzten die Annotationen intensiv und betrachteten im Schnitt 36 Meldungen bei der Bearbeitung, von der die deutliche Mehrheit Wertebereichsmeldungen aus der Statischen Analyse von Codesys waren. In Test- und Kontrollgruppe wurden zudem auch Codesys-Meldungen in etwa vergleichbarem Maß in Form der Annotationen genutzt. 7,33 Mal war dies bei Teilnehmern der Kontrollgruppe der Fall. Zu beachten ist hier jedoch die große Streuung insbesondere in der Testgruppe. Dies weist auf persönliche Unterschiede mit Präferenzen zu unterschiedliche Arbeitsweisen oder auf einen anderen Erfahrungsstand hin.

Diese Unterschiede zwischen den Probanden sind auch der Grund, weshalb die Bearbeitungszeit sich auch bei dieser Aufgabe nicht sinnvoll auswerten lässt. Im Mittel ist die Bearbeitungszeit gleich, allerdings weisen insbesondere die Ausreißer mit den Nummern 2 und 7 mit sehr kurzen bzw. langen Bearbeitungszeiten darauf hin, dass die vorliegenden Zeiten stärker von den Eigenschaften und Fähigkeiten der Probanden abhängt als von der verwendeten Methode selbst.

Bei näherer Betrachtung lässt sich bei den Probanden 2 und 4 der Testgruppe in Abbildung 5.5 erkennen, dass die Annotationen besonders zu Beginn der Bearbeitung genutzt wurden. Sie wurden somit für die Einarbeitung in das bestehende Codegerüst genutzt, was ein Indikator für Hypothese 3 wäre. In der Einarbeitungsphase selbst wurde das Unterstützungssystem von den Probanden 4 und 6 nicht genutzt. Sie setzten es jedoch wie Proband

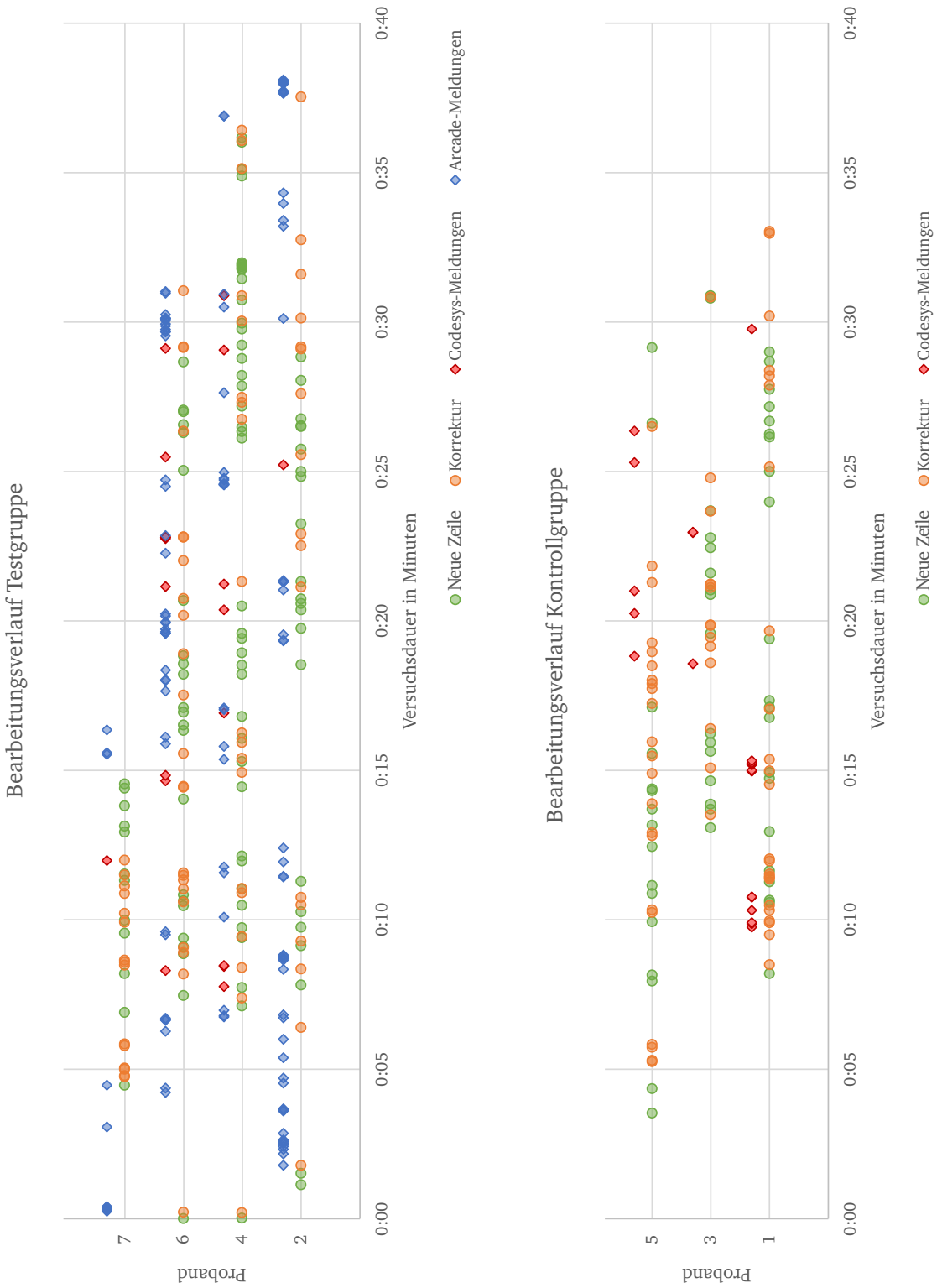


Abbildung 5.5: Bearbeitungsverlauf von Aufgabe 2 aller sieben Probanden. Teilnehmer 2, 4, 6, 7 gehörten zur Testgruppe, 1, 3 und 5 zur Kontrollgruppe. Neu erstellte Zeilen und Korrekturen bestehender Zeilen sind jeweils separat erfasst. Darüber ist im Zeitverlauf das Lesen von Codesys- und ARCADE-Meldungen aufgetragen (Daten erhoben in [Müllers, 2018]).

2 während der Ansicht im Editor bei der Programmierung ein, bevor das Programm im Simulator getestet wurde. Proband 7 zeigte kein direkt vergleichbares Programmierverhalten, da er das Programm zunächst komplett schrieb und nach wenigen gelesenen Annotationen im Simulator testete. Hier war keine Nachbearbeitung mehr nötig.

Erwähnenswert ist noch eine Beobachtung aus dem Versuch selbst. So nutzte Proband 6 Meldungen von ARCADE, dass eine Variable einen konstanten Wert über die Laufzeit behalte, als eine Aufgabenliste. Die Variablen wurden dafür zu Beginn deklariert, wodurch sie von der Statischen Analyse mit einem entsprechenden Hinweis annotiert wurden. Durch den schrittweisen Einbau im Programm wurden dann auch diese Hinweise abgebaut.

Eine Aussage über Hypothese H3 kann mit den aufgezeichneten Beobachtungen nicht gemacht werden. Die Bearbeitungsverläufe waren dafür zu unterschiedlich und die Einlesezeiten in das Programm nicht klar abgrenzbar. Zwei der Probanden nutzten die Annotationen der Wertemengen im Quellcode zwar direkt zu Beginn der Aufgabenbearbeitung, jedoch erlauben es die starken Schwankungen bis zum Beginn gehäufter Editiervorgänge nicht, diese Zeiten statistisch zu interpretieren. Es bleibt jedoch die Erkenntnis, dass die Annotationen sowohl beim Einlesen in den bestehenden Quelltext als auch während der Entwicklung genutzt wurden. Die Abschnitte, in denen die Visualisierung in der Testgruppe genutzt wurden, erscheinen kürzer als in der Kontrollgruppe, jedoch reicht dies noch nicht für eine belastbare Aussage.

5.3.2 Ergebnis der Fragebögen

Nach der Bearbeitung der beiden Aufgaben wurden die Probanden gebeten, in einem Fragebogen ihre gemachten Erfahrungen mit dem Prototyp zu bewerten. Zwar ist eine statistische Auswertung der Daten durch die geringe Teilnehmeranzahl auch hier nicht sinnvoll, allerdings lassen sich zu manchen Fragen Tendenzen erkennen, die hier vorgestellt werden.

Drei Arten von Programmierhilfen wurden im Fragebogen separat abgefragt.

- Meldungen im Nachrichtenfenster unterhalb des Quellcode-Editors
- Annotationen im Quellcode
- Visualisierung bei der Simulation des Programms

Zur Erinnerung: Die Meldungen im Nachrichtenfenster waren zusätzlich als Annotation im Quelltext sichtbar, wenn der Mauszeiger darüber gehalten wurde. Zusätzlich ließen sich so die Wertemengen im Quelltext anzeigen. Die Visualisierung der Simulation entspricht dem gewohnten Prozess der Programmierer, in dem sie zunächst Code entwickeln und diesen dann in der Simulation testen.

In der Kontrollgruppe werden Annotationen im Quellcode überwiegend als sinnvoll angesehen. Diese Annotationen wiesen bei ihnen auf Compiler-Warnungen von Codesys hin. Im Meldungsfenster wurden genau diese Meldungen jedoch nicht als hilfreich empfunden. Ein Grund dafür könnte sein, dass die Meldungen zum Teil inhaltlich erwartbar sind, da diese syntaktisch bedingten Fehler während der Entwicklung immer gemeldet werden. Die

Visualisierung gibt dann Aufschluss darüber, dass ein Fehler vorliegt, den der Programmierer auch ohne weitere Betrachtung des Nachrichtenfensters beheben kann.

In der Testgruppe ist die Meinung über Annotationen und Nachrichtenfenster ausgeglichener. Annotationen werden hier als durchschnittlich hilfreich eingestuft, die Meldungen im Nachrichtenfenster dagegen etwas besser. Ein Grund dafür könnte jedoch ein Missverständnis in den Fragebögen sein, da hier nicht noch einmal explizit definiert wurde, welche Formen von Annotationen gemeint waren. Dieser Punkt wurde daher später noch einmal differenzierter abgefragt.

Im Bezug auf Aufgabe 1 wurden die Probanden befragt, ob sie durch die Entwicklungsumgebung auf Fehler im Programm hingewiesen wurden. Fast alle Teilnehmer aus beiden Gruppen beantworteten diese Frage mit Ja, bezogen in einem Freitextfeld ihre Antwort jedoch vor allem auf syntaktische Fehler, die nicht spezifisch für die Statische Analyse durch ARCADE.PLC sind.

Die Visualisierung wurde von allen Probanden überwiegend nützlich für die Entwicklung und Fehlersuche wahrgenommen. Diese Aussage deckt sich auch mit den Aufzeichnungen aus den Versuchen, dass beide Gruppen jeweils phasenweise im Wechsel programmierten und ihren Code in der Visualisierung testeten. Dies spricht für einen Gewöhnungseffekt, da diese Entwicklungsmethode den Programmierern bekannt ist und in der Praxis angewendet wird. Auch aus der Literatur ist dieses Verhalten abseits der Entwicklung für SPSen untersucht worden [8].

In diesem Teil des Fragebogens wurden die Probanden noch abschließend gefragt, ob sie Probleme mit der Handhabung des Prototypen hatten oder es zu Einschränkungen und Programmfehlern bei der Nutzung kam. Einzelne Teilnehmer merkten hier an, dass gewohnte Kontextmenüs nicht wie gewohnt vorhanden sind und dadurch einzelne Features nicht genutzt werden können. Auch verhielt sich der Codeeditor teilweise etwas anders und konnte in beiden Gruppen keine Autovervollständigung anbieten. Als Einschränkung wurden diese Punkte jedoch nicht wahrgenommen. Die Aufgaben konnten problemlos bearbeitet werden und es sind keine Programmfehler aufgefallen.

Auch wurden die Probanden gefragt, ob die zusätzlichen Annotationen sie abgelenkt hätten, die Meldungen unerwartet oder falsch gewesen seien. Dies wurde von den Probanden jedoch klar verneint. Die einzelnen Antworten in den Fragebögen auf diese ausgewählten Fragen sind in Tabelle 5.2 zusammengefasst. Auf eine weitere Auswertung wurde aufgrund des kleinen Stichprobenumfangs verzichtet. Die detailliertere Zusammenfassung der Antwortbögen befindet sich Anhang B.

Nach der Beantwortung dieser Fragen hatten die Probanden beider Gruppen die Gelegenheit, den Prototyp frei zu testen. So konnten sie sich vom Funktionsumfang der aktiven Unterstützungsfunktionen abseits der Aufgabenstellung ein Bild machen. Nach der Präsenzzeit wurde dann noch einmal in einem Onlinefragebogen abgefragt, wie einzelne Features des Unterstützungssystems wahrgenommen wurden. Dieser zeitliche Versatz gab den Teilnehmern die Möglichkeit, über den Versuch zu reflektieren. Auch standen sie nicht mehr in einer Test- oder Kontrollsituation, in der eine Erwartungshaltung seitens des Versuchsleiters die Beurteilung beeinflussen könnte. Im Fragebogen war noch einmal explizit nach Annotationen im Quelltext gefragt und mit zwei Beispielen der Unterschied zwischen

	Testgruppe				Kontrollgruppe		
Frage	2	4	6	7	1	3	5
Wahrgenommene Annotationen waren hilfreich	3	2	3	3	5	1	4
(SA-)Meldungen sind verständlich	4	4	4	4	3	3	4
Annotationen stören Arbeitsfluss	1	1	2	1	1	4	1
(SA-)Meldungen sind inkonsistent	1	1	2	1	1	1	1
(SA-)Meldungen zeigen tatsächliche Fehler	4	4	3	4	1	2	4
Visualisierung decken Fehler auf	4	5	4	4	4	2	2

Tabelle 5.2: Auszug aus den Antworten der Probanden direkt nach der Aufgabenbearbeitung. Die Fragen wurden als Lickert-Item gestellt (1 stimme nicht zu, 5 stimme voll zu). Die Kontrollgruppe hatte noch keinen Zugang zum Prototyp (Daten erhoben in [Müllers, 2018]).

Warnungen und Hinweisen auf der einen und den Wertemengen auf der anderen Seite erklärt.

Bis auf eine Ausnahme waren die Teilnehmer der Meinung, dass die Annotationen im Quellcode mit Warnungen und Hinweisen auf tatsächliche Problemstellen im Quellcode hinweisen. Sie waren daher überzeugt, dass diese Meldungen bei der Programmierung hilfreich sind. Die Bewertung der annotierten Wertemengen fiel noch etwas positiver aus. So halten die meisten Teilnehmer die Darstellung für eine Hilfe bei der Einarbeitung in existierenden Quellcode und ebenso bei der Fehlersuche während der Programmierung. Ob die Meldungen auch konkret bei der Eingabe von Quelltext hilft wurde von den Teilnehmern neutral bewertet. Hieraus lässt sich weder eine Zustimmung noch Ablehnung der Aussage ablesen.

5.3.3 Diskussion

Ziel der Evaluation war die Untersuchung der Hypothesen H2, H3 und H4. Bereits erwähnt wurde der geringe Stichprobenumfang durch die geringe Teilnehmeranzahl, durch den eine statistische Auswertung der Daten nicht sinnvoll ist. Auch kann eine Unabhängigkeit der Probanden letztlich nicht bewiesen werden, da sie teilweise persönlich zu der Studie eingeladen wurden oder in einem kooperierenden Unternehmen der vorgestellten Arbeit grundsätzlich positiv gegenüber eingestellt sind. Dagegen spricht, dass durchaus auch negatives Feedback beispielsweise über Eigenschaften des Editors geäußert wurden. Qualitative Ergebnisse können daher sehr wohl aus dieser Studie gezogen werden.

Die aufgestellte Hypothese H2 sagt aus, dass durch die Anzeige von Informationen der Statischen Analyse weniger Fehler im Programm gemacht werden und die Informationen auch genutzt werden. Eine abgesicherte Aussage zu dieser Hypothese liefert die Studie nicht, allerdings kann resümiert werden, dass alle Probanden der Testgruppe die Informationen während der Bearbeitung beider Aufgaben genutzt haben. Auch ist es einem Probanden aus der Testgruppe gelungen, durch Warnungen der Statischen Analyse einen Flüchtigkeitsfehler in Aufgabe 1 zu korrigieren, während zwei Andere aus der Kontrollgruppe den gleichen Fehler machten, jedoch nicht korrigierten. Durch die Gestaltung von Aufgabe 2 war diese

weniger anfällig für Flüchtigkeitsfehler, die durch die Statische Analyse erkannt werden können. Ein Effekt konnte daher hier nicht beobachtet werden.

Die Aussage von Hypothese H3, dass die annotierten Wertemengen die Einlesezeit verkürzen und zu einem anderen Entwicklungsverhalten führen, kann ebenfalls nicht bewiesen werden. Auch hier gibt es Tendenzen und unterschiedliche Muster im Bearbeitungsverlauf, wie Abbildung 5.5 zeigt. Einige Probanden nutzen die Annotationen in Bearbeitungspausen, um den Verlauf des Programms nachzuvollziehen, statt in den Simulationsmodus zu wechseln, was in diesem Fall einen Zeitvorteil bringt. Auch während der Einlesezeit wurde diese Funktion von Probanden genutzt, um sich über den existierenden Quellcode einen Überblick zu verschaffen. Dies deckt sich mit der Aussage des nachgelagerten Fragebogens, in dem fast alle Teilnehmer der Aussage zustimmten, dass die Annotation von Wertemengen für diese Zwecke eine Unterstützung bietet. Der Unterschied zwischen eingeschätztem Nutzen und messbaren Ergebnissen ist ein Anhaltspunkt dafür, dass das Aufgabendesign oder der Umfang möglicherweise nicht optimal war, um aus den Unterstützungsfunktionen einen größtmöglichen Nutzen ziehen zu können. Persönliche Unterschiede zwischen den Probanden führen darüber hinaus dazu, dass das Programmierverhalten nicht im Vergleich betrachtet werden kann. Dieser Zusammenhang wurde bereits von Krämer [26] untersucht, dessen Ergebnisse hier nun weder bestätigt noch widerlegt werden konnten.

Schließlich bleibt die Aussage von Hypothese H4, die zusammenfasst, dass Wertemengen sowie annotierte Warnungen und Hinweise einen Nutzen für den Programmierer darstellen. Hierfür können drei Aspekte zusammengeführt werden: Die zwei zuvor genannten Hypothesen sowie die Auswertung der Fragebögen. Ergebnis des nachgelagerten Fragebogens ist es, dass die Versuchsteilnehmer fast alle der Meinung sind, dass Warnungen und Hinweise der Statischen Analyse eine Hilfe bei der Entwicklung sind. Bei der Anzeige der Wertemengen zeigte sich ein noch positiveres Bild, da die Teilnehmer die Eignung zur Unterstützung während des Einlesens und der Fehlersuche als durchgehend positiv einschätzten. Lediglich in den Schreibphasen, in denen nur Programmcode eingegeben wird, waren die Teilnehmer neutral eingestellt, ob ihnen die verfügbaren Wertemengen hier einen Vorteil bieten.

Der letzte Punkt könnte an einem nötigen Wechsel des Eingabegerätes liegen. Während die Eingabe des Programms ausschließlich über die Tastatur erfolgt, muss für eine Anzeige verfügbarer Wertemengen auf die Maus gewechselt werden, um den Mauszeiger über einen Variablenbezeichner zu bewegen. Diese Einschränkung könnte durch eine andere Darstellung der Wertemengen optimiert werden, auf die auch ohne Wechsel zwischen Tastatur und Maus zugegriffen werden kann.

Die Erkenntnisse und Indizien der Hypothesen H2, H3 und H4 lassen sich zur anfangs aufgestellten Hypothese H1 vereinigen. Ihre Aussage war es, dass die Bereitstellungen von Informationen aus der Statischen Analyse eines bereits teilweise geschriebenen Programms eine Hilfe bei der Entwicklung von SPS-Programmen darstellt. H2, H3 und H4 haben hierzu jeweils einen Teilaspekt konkretisiert, für den im Rahmen dieser Studie Indizien für die Validität der Hypothesen gefunden werden konnten. Auch wenn eine statistisch belastbare Grundlage dafür für H1 damit noch nicht gegeben ist, so deuten die mit dieser Studie gefundenen Anhaltspunkte auf die Validität der Hypothese hin.

Kapitel 6

Schluss

In der vorliegenden Arbeit wurde ein Unterstützungssystem für die Entwicklung von SPS-Programmen mittels Statischer Analyse vorgestellt und evaluiert. Dafür wurden zunächst die Anforderungen an ein solches System anhand des geplanten Nutzungsszenarios aufgestellt. Die anschließende Implementierung umfasst zwei Editoren, mit denen die entwickelten Unterstützungsfunktionen genutzt werden können. Ein erster Prototyp auf Basis der Eclipse-IDE kann editierbare Programme in der Sprache Structured Text analysieren. Warnungen und Hinweise der Statischen Analyse können im Text annotiert werden, und es lassen sich Mengen von Werten an Variablen anzeigen, die diese bei der Ausführung annehmen können.

Um diese Informationen generieren zu können, wurde das Analyse-Framework ARCADE.PLC angepasst und erweitert. Somit konnten Wertemengen aus den internen Datenstrukturen der Analyse exportiert und für die Annotation in einer IDE aufbereitet werden.

Diese Anbindung wurde in einem zweiten Prototyp umgesetzt, der sich nahtlos in die Codesys-3-Entwicklungsumgebung einfügt. Ein dafür entwickelter Editor ist in der Lage, sowohl sichtbare Markierungen für Fehler und Hinweise im Text vorzunehmen, als auch nicht permanent sichtbare Annotationen für die Anzeige von Wertemengen vorzuhalten. Beide Editor-Prototypen können Wertemengen auch an Teilen des Quellcodes annotieren, die noch nicht analysiert werden konnten, da sie während der Eingabe vorübergehend nicht syntaktisch korrekt sind.

Um die benötigte Berechnungszeit der Statischen Analyse insgesamt zu verkürzen, wurde die Wertemengenanalyse in ARCADE.PLC hin zu einer inkrementellen Analyse erweitert. Die Erweiterung baut auf die Beobachtung auf, dass minimale Änderungen am Quelltext, wie sie während der Entwicklung häufig auftreten, meist nur einen kleinen Teil der Analyseergebnisse beeinflussen. Die Ergebnisse einer vorherigen Programmiteration werden dafür herangezogen und durch das vorgestellte Verfahren nur an den Stellen neuberechnet, die von der Änderung auch betroffen sind. Performance-Messungen zeigten bei einigen Programmen einen Gewinn der Berechnungszeit der VSA von etwa 50 %. Allerdings muss dieses Ergebnis in Relation zum zusätzlichen Aufwand gesehen werden, den Abhängigkeitsanalysen vor der VSA nötig machen. So gibt es auch Programmänderungen bei denen die Gesamtdauer der Analyse von dieser Erweiterung nicht oder nur wenig von der inkrementellen Arbeitsweise profitiert.

Schließlich wurde das Unterstützungssystem im Rahmen einer kleinen Nutzerstudie evaluiert. Aufgrund der niedrigen Teilnehmerzahl sind hier keine statistisch belastbaren Auswertungen möglich. Dennoch konnten zwei elementare Beobachtungen gemacht werden: Die Nutzer nahmen die zusätzlichen Annotationen in der Studie nicht als Ablenkung von ihrer eigentlichen Tätigkeit wahr und sie nutzten Wertemengen-Angaben während ihrer Aufgabenbearbeitung. Auch war es möglich, bei einem Probanden durch den Prototypen einen Flüchtigkeitsfehler in der Implementierung zu verhindern. Eine Aussage über geändertes Arbeitsverhalten lässt sich aufgrund der Unterschiede zwischen den Probanden jedoch nicht formulieren. Fast alle Teilnehmer der Studie bewerteten die Unterstützungsfunktionen als hilfreich, allerdings konnte kein valider Vergleich zwischen der Test- und Kontrollgruppe durchgeführt werden, da die Gruppengrößen dafür zu klein waren.

Ausblick

Die Integration und Präsentation der Ergebnisse der Statischen Analyse in eine Entwicklungsumgebung sind ein wichtiger Schritt, um die Akzeptanz von Statischer Analyse bei SPS-Programmierern bzw. Entwicklern zu verbessern und Hemmschwellen für die Nutzung generell abzubauen. Optimierungspotenzial gibt es an mehreren Stellen der Visualisierung und Gestaltung des implementierten Editors. Hier gibt es bereits Ansätze, die Annotation von Wertemengen optisch auffälliger und permanent sichtbar zu gestaltet, um damit den Wechsel zwischen Maus und Tastatur für den Programmierer überflüssig machen.

So sollte es in Zukunft möglich sein, Meldungen der Statischen Analyse zu quittieren oder automatisch zu verbergen, wenn diese bereits ein Mal vom Programmierer wahrgenommen wurden. Somit besteht weniger die Gefahr, dass die Unterstützung durch Statische Analyse als Störfaktor wahrgenommen wird und die Meldungen letztlich ignoriert werden.

Hinsichtlich der Benutzerstudie ist es in Zukunft sinnvoll, diese mit mehr Probanden zu wiederholen und die Komplexität der Aufgaben so abzustimmen, dass der Programmierer von den Unterstützungsfunktionen mehr Gebrauch machen kann. Mit mehr erfassten Daten könnten dann auch mithilfe statistischer Auswertungen Aussagen über die gesammelten Daten getroffen werden.

In dieser Studie wurde das Szenario auf die Entwicklung von SPS-Programmen in Structured Text begrenzt. Es ist darüber hinaus interessant, auch grafische Programmiersprachen wie FBS zu analysieren und Annotationen daran anzuzeigen. Hier müssten möglicherweise auch andere Prüfverfahren gewählt werden, da Warnungen wie unerreichbarer Code innerhalb eines genutzten Funktionsblocks nur von geringem Wert sind. Im Fall von Enable-Eingängen ist dies häufig sogar ein gewünschtes Verhalten. Andere Überprüfungen mit Kenntnis möglicher Wertebereiche können dagegen wertvoller für den Entwickler sein, um Seiteneffekte und unerwünschtes Verhalten schon beim Verbinden von Funktionsblöcken signalisiert zu bekommen und Wertebereiche an den Schnittstellen von POEs zu prüfen.

Das Potenzial von Statischer Analyse mit abstrakter Interpretation liegt neben diesen Einsatzszenarien jedoch besonders auch in der Einarbeitung in bzw. Modifikation von fremdem Programmcode. Im Lebenszyklus eines Automatisierungsprojektes ist dies ein realistisches Szenario, da Revisionen oft viele Jahre nach Inbetriebnahme einer Anlage

stattfinden und der Entwickler, das Wissen und die Dokumentation über die SPS-Programme meist nicht mehr zur Verfügung stehen. Die vorgestellte Evaluation untersuchte in einem kleinen Beispiel bereits die Modifikation von existierendem Programmcode und zeigte erste Anzeichen, dass sich der vorgestellte Ansatz für dieses Einsatzszenario eignet. Das Abbauen der Einstiegsbarrieren durch Werkzeugintegration und minimieren von Konfigurationsaufwänden kann hier einen großen Beitrag leisten, Statische Analyse in der Praxis nutzbarer zu machen. Neben der Ausweitung der Nutzerstudie an sich ist in Zukunft auch von Interesse, ob in diesem Fall andere Ergebnisdarstellungen hilfreich sind. So sind Darstellungen denkbar, die dem Entwickler Zusammenhänge und Auswirkungen geplanter Editieroperationen anzeigen.

Hinsichtlich der Laufzeit der Analysen gibt es für die Praktische Nutzung Optimierungsbedarf. Die vorgestellte inkrementelle Analyse befasst sich ausschließlich mit der VSA, doch tragen nach dieser Optimierung auch andere Schritte des Analyseprozesses zur Gesamtlaufzeit bei, die möglicherweise ebenfalls optimiert werden können. Ein großer Gewinn liegt sicherlich in der Neuimplementierung des ARCADE.PLC -Frameworks in C++. Dieser Umbau befindet sich derzeit in Arbeit und verspricht aktuell deutlich kürzere Analysezeiten, wodurch möglicherweise Effizienz-steigernde Maßnahmen wie inkrementelle Analysen in Zukunft neu bewertet werden müssen.

Thematisch relevante Abschlussarbeiten

[Müllers, 2018] Müllers, T. H. (2018). Integration von Statischer Analyse in Entwicklungsumgebungen für Steuerungscode. Bachelorarbeit, RWTH Aachen University, Aachen.

[Rath, 2017] Rath, D. (2017). Syntactic Analysis of PLC Software Projects. Bachelorarbeit, RWTH Aachen University, Aachen.

[Shaaban, 2016] Shaaban, M. (2016). Iterative Static Analysis for PLC Code During Development. Masterarbeit, RWTH Aachen University, Aachen.

[Wolf, 2015] Wolf, B. (2015). A Structured Text Editor by Use of the Meta Programming System. Bachelorarbeit, RWTH Aachen University, Aachen.

Kommentar zur Neutralität der Evaluation

Im Rahmen der Abschlussarbeit von Herrn Till Müllers [Müllers, 2018] wurde die Codesys-Integration aus dieser Arbeit in einer Nutzerstudie evaluiert. Den Aufbau und Umfang hatte ich für diese Arbeit vorgeschlagen. Diesem Vorschlag folgte er in seiner Arbeit grundsätzlich. Ich motivierte Herrn Müllers, die Evaluation der Integration aus einem neutralen Standpunkt heraus durchzuführen und die Rahmenbedingungen seiner Arbeit nicht als gegeben hinzunehmen sondern die Integration als Untersuchungsgegenstand zu sehen. Um sowohl ihn als auch die Versuchsteilnehmer nicht zu beeinträchtigen, führte Herr Müller die Nutzerstudie ohne meine Präsenz durch. Es war von Beginn an klar, dass das Ergebnis der Evaluation keinerlei Einfluss auf die Bewertung seiner Arbeit haben würde. Die Auswertung der Ergebnisse wurde von mir nicht beeinflusst und das entsprechende Ergebnis-Kapitel von mir vor der Abgabe nicht korrigiert.

Literaturverzeichnis

[1] Alberto Apostolico and Zvi Galil. *Pattern matching algorithms*. Oxford University Press, New York, 1997.

[2] Lasse Bergroth, Harri Hakonen, and Timo Raita. A survey of longest common subsequence algorithms. In *Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000*, pages 39–48. IEEE Comput. Soc, 2000.

[3] Sebastian Biallas. Verification of Programmable Logic Controller Code using Model Checking and Static Analysis. Dissertation, RWTH Aachen University, Aachen, 2016.

[4] Sebastian Biallas, Jörg Brauer, and Stefan Kowalewski. Counterexample-guided abstraction refinement for PLCs. In *5th International Workshop on Systems Software Verification (SSV 2010), Vancouver, Canada*, pages 2–12, Berkeley, CA, USA, 2010. USENIX Association.

[5] Sebastian Biallas, Volker Kamin, Stefan Kowalewski, Bastian Schlich, Stephan Sehestedt, and Stefan Stattelmann. Verifikation von sicherheitsgerichteten SPS-Programmen mit Hilfe von Safety-Automaten. In *Automation 2013 : 14. Branchentreff der Mess- und Automatisierungstechnik ; Kongresshaus Baden-Baden, 25. und 26. Juni 2013 / VDI/VDE-Gesellschaft Mess- und Automatisierungstechnik*, volume 2209 of *VDI-Berichte*, pages 75–79. VDI-Verl., Düsseldorf, 2013.

[6] Sebastian Biallas, Stefan Kowalewski, Stefan Stattelmann, and Bastian Schlich. Efficient Handling of States in Abstract Interpretation of Industrial Programmable Logic Controller Code. *IFAC Proceedings Volumes*, 47(2):400–405, 2014.

[7] Sébastien Bornot, Ralf Huuck, Ben Lukoschus, and Yassine Lakhnech. Utilizing Static Analysis for Programmable Logic Controllers. In Sebastian Engell, editor, *ADPM 2000 conference proceedings*, Berichte aus der Automatisierungstechnik, pages 183–187, Aachen, 2000. Shaker.

[8] Joel Brandt, Philip J. Guo, Joel Lewenstein, Scott R. Klemmer, and Mira Dontcheva. Writing Code to Prototype, Ideate, and Discover. *IEEE Software*, 26(5):18–24, 2009.

[9] Fabien Campagne. *The MPS language workbench*. Campagne Laboratory, New York, NY, USA, third edition, version 1.5.1, march 2016 edition, 2016.

[10] Vaclav Chvatal, David Anthony Klarner, and Donald Ervin Knuth. *Selected combinatorial research problems*. Report (Stanford University. Computer Science Department). Computer Science Department, Stanford University, 1972.

[11] Agostino Cortesi. Widening Operators for Abstract Interpretation. In *2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*, pages 31–40. IEEE, 2008.

[12] Patrick Cousot and Radhia Cousot. Abstract interpretation. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '77*, pages 238–252, New York, New York, USA, 1977. ACM Press.

[13] Douglas Crockford. The application/json Media Type for JavaScript Object Notation (JSON): RFC 4627.

[14] Pierpaolo Degano and Corrado Priami. LR techniques for handling syntax errors. *Computer Languages*, 24(2):73–98, 1998.

[15] Deutsches Institut für Normung e.V. *DIN EN 61131-3:2003: Speicherprogrammierbare Steuerungen - Teil 3: Programmiersprachen*. 2 edition, 2002-12-01.

[16] Deutsches Institut für Normung e.V. *DIN EN 61131-3:2013: Speicherprogrammierbare Steuerungen – Teil 3: Programmiersprachen*. 2014-06.

[17] Deutsches Institut für Normung e.V. *DIN EN 61131-10: Speicherprogrammierbare Steuerungen – Teil 10: XMLAustauschformat für IEC 61131-3 (Entwurf)*. 2017-10-20.

[18] Alpana Dubey. Evaluating software engineering methods in the context of automation applications. In *2011 9th IEEE International Conference on Industrial Informatics*, pages 585–590. IEEE, 2011.

[19] Martin Fowler. *Refactoring: Improving The Design Of Existing Code*. Addison-Wesley object technology series. Addison-Wesley, Boston, 1999.

[20] Keith Gallagher and David Binkley. Program slicing. In *2008 Frontiers of Software Maintenance*, pages 58–67, 2008.

[21] Jennifer Gluck, Andrea Bunt, and Joanna McGrenere. Matching attentional draw with utility in interruption. In Mary Beth Rosson and David Gilmore, editors, *Proceedings of the SIGCHI conference on Human factors in computing systems - CHI '07*, page 41, New York, New York, USA, 2007. ACM Press.

[22] Sebastian Hack, R. Wilhelm, and Helmut A. Seidl. *Compiler design: Code generation and machine-level optimization*. Springer, Berlin and London, 2011.

[23] International Electrotechnical Commission. *IEC 61508:1998: Functional Safety of Electrical, Electronic and Programmable Electronic Safety-Related Systems*.

[24] International Electrotechnical Commission. *IEC 61511:2003: Functional safety - Safety instrumented systems for the process industry sector*.

[25] Benjamin Kormann and Birgit Vogel-Heuser. Automated test case generation approach for PLC control software exception handling using fault injection. In *IECON 2011 - 37th Annual Conference of the IEEE Industrial Electronics Society*, pages 365–372. IEEE, 2011.

[26] Jan-Peter Krämer. Interacting with code: Observations, models, and tools for usable software development environments. Dissertation, RWTH Aachen University, Aachen, 2016.

[27] Jan-Peter Krämer, Joachim Kurz, Thorsten Karrer, and Jan Borchers. How live coding affects developers' coding behavior. In *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 5–8. IEEE, 2014.

[28] Nicolas Lopez and André van der Hoek. The code orb. In Richard N. Taylor, Harald Gall, and Nenad Medvidović, editors, *Proceeding of the 33rd international conference on Software engineering - ICSE '11*, page 824, New York, New York, USA, 2011. ACM Press.

[29] Arndt Lüder, Nicole Schmidt, Heinrich Steiniger, and Stefan Biffl. Analyse von Anforderungen an Software-Systeme zum Steuerungsentwurf. In Inst. für Automation und Kommunikation e.V., editor, *Entwurf komplexer Automatisierungssysteme : EKA 2014*, 2014.

[30] I. Scott MacKenzie. *Human-computer interaction*. Elsevier Science, Burlington, 2012.

[31] Emerson Murphy-Hill and Andrew P. Black. Refactoring Tools: Fitness for Purpose. *IEEE Software*, 25(5):38–44, 2008.

[32] Emerson Murphy-Hill and Andrew P. Black. An interactive ambient visualization for code smells. In Alexandru Telea, Carsten Görg, and Steven Reiss, editors, *Proceedings of the 5th international symposium on Software visualization - SOFTVIS '10*, page 5, New York, New York, USA, 2010. ACM Press.

[33] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How We Refactor, and How We Know It. *IEEE Transactions on Software Engineering*, 38(1):5–18, 2012.

[34] Eugene W. Myers. AnO(ND) difference algorithm and its variations. *Algorithmica*, 1(1-4):251–266, 1986.

[35] Sreeja Nair, Raoul Jetley, Anil Nair, and Stefan Hauck-Stattelmann. A static code analysis tool for control system software. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 459–463. IEEE, 2015.

[36] Namur. *NAMUR-Arbeitsblatt NA35 - Abwicklung von PLT-Projekten*. NAMUR-Arbeitskreis 1.1 "Planungsmethodik“, Leverkusen, 24.3.2003.

[37] Mathias Obster and Stefan Kowalewski. A live static code analysis architecture for PLC software. In *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–4. IEEE, 2017.

[38] Chang Mok Park, Sangchul Park, and Gi-Nam Wang. Control logic verification for an automotive body assembly line using simulation. *International Journal of Production Research*, 47(24):6835–6853, 2009.

[39] Herbert Prähofer, Florian Angerer, Rudolf Ramler, and Friedrich Grillenberger. Static Code Analysis of IEC 61131-3 Programs: Comprehensive Tool Support and Experiences from Large-Scale Industrial Application. *IEEE Transactions on Industrial Informatics*, 13(1):37–47, 2017.

[40] Herbert Prähofer, Florian Angerer, Rudolf Ramler, Hermann Lacheiner, and Friedrich 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 2012)*, pages 1–8. IEEE, 2012.

[41] Rudolf Ramler, Werner Putschögl, and Dietmar Winkler. Automated testing of industrial automation software: Practical receipts and lessons learned. In Anil R. Nair, Herbert Prähofer, Alois Zoitl, Raoul Jetley, Alpana Dubey, and Atul Kumar, editors, *Proceedings of the 1st International Workshop on Modern Software Engineering Methods for Industrial Automation - MoSEMInA 2014*, pages 7–16, New York, New York, USA, 2014. ACM Press.

[42] Susanne Rösch, Dmitry Tikhonov, Daniel Schütz, and Birgit Vogel-Heuser. Model-based testing of PLC software: Test of plants' reliability by using fault injection on component level. *IFAC Proceedings Volumes*, 47(3):3509–3515, 2014.

[43] David Saff and Michael D. Ernst. Reducing wasted development time via continuous testing. In *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003*, pages 281–292. IEEE, 2003.

[44] David Saff and Michael D. Ernst. An experimental evaluation of continuous testing during development. In George Avrunin and Gregg Rothermel, editors, *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis - ISSTA '04*, page 76, New York, New York, USA, 2004. ACM Press.

[45] Nicole Schmidt, Arndt Lüder, Heinrich Steininger, and Stefan Biffl. Analyzing requirements on software tools according to the functional engineering phase in the technical systems engineering process. In *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, pages 1–8, 2014.

[46] Smart Software Solutions GmbH. CoDeSys Automation Platform: Concept and Tutorial, Document Version 1.0, SDK-Dokumentation. 03-2007.

[47] Rob St. Amant, John Riedl, and Anthony Jameson, editors. *Proceedings of the 10th international conference on Intelligent user interfaces - IUI '05*, New York, New York, USA, 2005. ACM Press.

[48] Stefan Stattelmann, Sebastian Biallas, Bastian Schlich, and Stefan Kowalewski. Applying static code analysis on industrial controller code. In *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, pages 1–4. IEEE, 2014.

[49] Steven L. Tanimoto. VIVA: A visual language for image processing. *Journal of Visual Languages & Computing*, 1(2):127–139, 1990.

[50] Steven L. Tanimoto. A perspective on the evolution of live programming. In *2013 1st International Workshop on Live Programming (LIVE)*, pages 31–34. IEEE, 2013.

[51] Michael Tiegelkamp and Karl Heinz John. *SPS-Programmierung mit IEC 61131-3*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[52] Michael L. van de Vanter, Susan L. Graham, and Robert A. Ballance. Coherent user interfaces for language-based editing systems. *International Journal of Man-Machine Studies*, 37(4):431–466, 1992.

[53] Birgit Vogel-Heuser, Thomas Bauernhansl, and Michael ten Hompel, editors. *Handbuch Industrie 4.0 Band 2: Automatisierung*, volume 2 of *Springer Reference Technik*. Springer Reference Technik, Berlin, 2 edition, 2017.

[54] Valeriy Vyatkin. Software Engineering in Industrial Automation: State-of-the-Art Review. *IEEE Transactions on Industrial Informatics*, 9(3):1234–1249, 2013.

[55] Robert A. Wagner and Michael J. Fischer. The String-to-String Correction Problem. *Journal of the ACM*, 21(1):168–173, 1974.

[56] Reinhard Wilhelm, Helmut Seidl, and Sebastian Hack. *Übersetzerbau*. EXamen.press. Springer, Berlin, 2007-2012.

Anhang A

Fragebögen der Nutzerstudie

Es folgen die Fragebögen, die während der Evaluation in Kapitel 5 verwendet wurden, um Rückmeldungen der Probanden strukturiert aufzunehmen.

A.1 Vorwissen und Erfahrungen bei der Bearbeitung

Dieser erste Teil des Fragebogens wurde von [Müllers, 2018] erstellt. Er wurde den Probanden bis Frage 14 vor und ab Frage 15 nach der Durchführung der Aufgaben vorgelegt. Details zu den Aufgaben und zum weiterem Aufbau sind in Abschnitt 5.2 beschrieben.

Probandeninformationen

* Erforderlich

1. **Probandennummer:** *

2. **Gruppe:** *
   Markieren Sie nur ein Oval.
   - ◯ A
   - ◯ B

Persönliche Informationen

3. **Alter:**

4. **Geschlecht**
   Markieren Sie nur ein Oval.
   - ◯ Männlich
   - ◯ Weiblich
   - ◯ Keine Angabe

Programmiererfahrung und Vorkenntnisse

5. **Geschätzte Programmierpraxis (Stunden pro Woche)**
   Markieren Sie nur ein Oval.
   - ◯ 0 - 5
   - ◯ 5 - 10
   - ◯ 10 - 20
   - ◯ mehr als 20

6. **Wie schätzen sie ihre Fähigkeiten als Programmierer ein?**
   Markieren Sie nur ein Oval.

	1	2	3	4	5	
Anfänger	◯	◯	◯	◯	◯	Profi

Wie sehr stimmen sie der folgenden Aussage zu?

7. **Ich bin mit dem Tia Portal oder Step 7 vertraut.**
   Markieren Sie nur ein Oval.

	1	2	3	4	5	
überhaupt nicht	◯	◯	◯	◯	◯	sehr

8. **Ich bin mit CoDeSys 3.X vertraut.**
   Markieren Sie nur ein Oval.

	1	2	3	4	5	
überhaupt nicht	◯	◯	◯	◯	◯	sehr

9. **Ich bin mit Visual Studio vertraut.**
   Markieren Sie nur ein Oval.

	1	2	3	4	5	
überhaupt nicht	◯	◯	◯	◯	◯	sehr

10. **Ich bin mit Eclipse vertraut.**
    Markieren Sie nur ein Oval.

	1	2	3	4	5	
überhaupt nicht	◯	◯	◯	◯	◯	sehr

11. **Haben sie sich schon mit statischer Analyse beschäftigt?**
    Markieren Sie nur ein Oval.

- ◯ Ja
- ◯ Nein

12. **Wenn ja, mit welchem Tool?**

_____

13. **Haben sie statische Analyse bereits produktiv eingesetzt?**
    Markieren Sie nur ein Oval.

- ◯ Ja
- ◯ Nein

14. **Welche der folgenden Programmiersprachen beherrschen sie?**
    Wählen Sie alle zutreffenden Antworten aus.

- ☐ Sprachen der IEC 61131-3
- ☐ Objektorientierte Programmiersprachen (Java, C#, C++ ...)
- ☐ Structured Text / Structured Control Language

Aufgabe

15. **Wie bewerten sie den Schwierigkeitsgrad der Aufgabe?**
    Markieren Sie nur ein Oval.

	1	2	3	4	5	
sehr einfach	◯	◯	◯	◯	◯	sehr schwer

Wie sehr stimmen sie der folgenden Aussage zu?

16. **Die vom Editor zur Verfügung gestellten Annotationen bei der Bearbeitung waren hilfreich.**
    Markieren Sie nur ein Oval.

	1	2	3	4	5	
überhaupt nicht	◯	◯	◯	◯	◯	sehr

17. **Die von der Entwicklungsumgebung bereitgestellten Meldungen haben mich bei der Erkennung und Behebung von Fehlern unterstützt.**
    Markieren Sie nur ein Oval.

	1	2	3	4	5	
überhaupt nicht	◯	◯	◯	◯	◯	sehr

18. **Die Visualisierung hat mich bei der Erkennung und Behebung von Fehlern unterstützt.**
    Markieren Sie nur ein Oval.

	1	2	3	4	5	
gar nicht	◯	◯	◯	◯	◯	sehr

19. **Kam es während der Bearbeitung zu Fehlern, die ihnen schon im Editor aufgefallen sind?**
    Wählen Sie alle zutreffenden Antworten aus.

- ☐ Ja
- ☐ Nein

20. **Welche Art von Fehlern war das?**

21. **Kam es während der Bearbeitung zu Fehlern, die erst im Test mit der Visualisierung aufgefallen sind?**
Wählen Sie alle zutreffenden Antworten aus.

- [ ] Ja
- [ ] Nein

22. **Welche Art von Fehlern war das?**

Entwicklungsumgebung

Wie sehr stimmen sie den folgenden Aussagen zu?

23. **Die Meldungen der Entwicklungsumgebung waren klar und verständlich formuliert.**
Markieren Sie nur ein Oval.

	1	2	3	4	5	
überhaupt nicht	◯	◯	◯	◯	◯	sehr

24. **Die Meldungen der Entwicklungsumgebung haben mir das Verständnis des Quellcodes erleichtert.**
Markieren Sie nur ein Oval.

	1	2	3	4	5	
überhaupt nicht	◯	◯	◯	◯	◯	sehr

25. **Die Meldungen der Entwicklungsumgebung waren stets nachvollziehbar.**
Markieren Sie nur ein Oval.

	1	2	3	4	5	
überhaupt nicht	◯	◯	◯	◯	◯	sehr

26. **Die Meldungen der Entwicklungsumgebung haben mich überrascht.**
Markieren Sie nur ein Oval.

	1	2	3	4	5	
überhaupt nicht	◯	◯	◯	◯	◯	sehr

27. **Mein Arbeitsfluss wurde durch die Entwicklungsumgebung unterbrochen oder gestört.**
Markieren Sie nur ein Oval.

	1	2	3	4	5	
überhaupt nicht	◯	◯	◯	◯	◯	sehr

28. **Die Meldungen der Entwicklungsumgebung waren inkonsistent oder fehlerhaft.**
Markieren Sie nur ein Oval.

	1	2	3	4	5	
überhaupt nicht	◯	◯	◯	◯	◯	sehr

29. **Meldungen der Entwicklungsumgebungen haben mich auf tatsächliche Fehler hingewiesen.**
Markieren Sie nur ein Oval.

	1	2	3	4	5	
überhaupt nicht	◯	◯	◯	◯	◯	sehr

30. **Schildern sie ihre Erfahrungen mit den generierten Meldungen. Wo haben sich diese als Hilfreich erwiesen? Wo nicht?**

31. **Traten bei der Bearbeitung der Aufgabe Programmfehler auf, die deren Durchführung erschwerten?**
Markieren Sie nur ein Oval.

- ◯ Ja
- ◯ Nein

32. **Wenn ja, welche Programmfehler sind aufgetreten?**

Wie sehr stimmen sie den folgenden Aussagen zu?

33. **Programmfehler haben mich an der produktiven Arbeit mit dem Prototypen gehindert.**
Markieren Sie nur ein Oval.

	1	2	3	4	5	
überhaupt nicht	◯	◯	◯	◯	◯	sehr

34. **Die Funktion des Quellcode-Editors entsprach meinen Erwartungen.**
Markieren Sie nur ein Oval.

	1	2	3	4	5	
überhaupt nicht	◯	◯	◯	◯	◯	sehr

35. **Wo hatten sie Schwierigkeiten? Haben sie Verbesserungsvorschläge?**

A.2 Eindrücke zum Prototyp von allen Teilnehmern

Nach der regulären Aufgabenbearbeitung hatten alle Teilnehmer der Studie die Gelegenheit, den Prototyp der Testgruppe, also der ST-Editor mit Unterstützung durch Statische Analyse, auszuprobieren. Darüber wurden sie in einem nachgelagerten, kurzen Fragebogen befragt, der im Rahmen dieser Dissertation erstellt wurde.

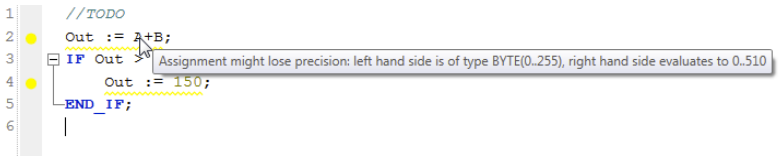
Intergration von statischer Analyse in CoDeSys

Vielen Dank noch einmal für Ihre Teilnahme an der Benutzerstudie. Sie hatten im Anschluss an die Aufgabenbearbeitung die Gelegenheit, den entwickelten Prototyp mit eingeschalteter Unterstützung eigenständig an Ihrem Quellcode zu testen. Im Folgenden bitten wir Sie dazu noch um einen abschließendes Feedback.

1. **Probandennummer**
   Bitte geben Sie Ihre Probandennummer ein, sofern Sie diese zur Hand haben. Falls nicht, lassen Sie dieses Feld bitte leer.

Im folgenden sehen Sie noch einmal die beiden Features, die Ihnen im Prototyp zur Verfügung standen und jeweils Aussagen dazu. Bitte beziehen Sie Ihre Aussagen jeweils auf eine generelle Verwendung auch abseits der gestellten Aufgaben.

Meldungen der statischen Analyse direkt im Quellcode



Wie sehr stimmen sie der folgenden Aussage zu?

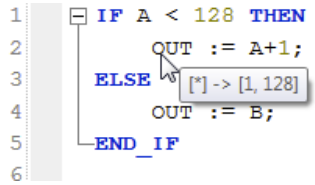
2. **Die Meldungen der statischen Analyse weisen auf potenzielle Fehlerquellen hin**
   Markieren Sie nur ein Oval.

	1	2	3	4	5	
überhaupt nicht	◯	◯	◯	◯	◯	sehr

3. **Die Meldungen der statischen Analyse sind nützlich bei der Programmentwicklung**
   Markieren Sie nur ein Oval.

	1	2	3	4	5	
überhaupt nicht	◯	◯	◯	◯	◯	sehr

Anzeige von Variablenwerten. Hier: Wertemengen vor und nach der Zuweisung



Wie sehr stimmen sie der folgenden Aussage zu?

4. **Die angezeigten Variablenwerte sind nützlich beim Verständnis von existierendem Quellcode**
   Markieren Sie nur ein Oval.

	1	2	3	4	5	
überhaupt nicht	◯	◯	◯	◯	◯	sehr

5. **Die angezeigten Variablenwerte sind nützlich bei der Programmentwicklung**
   Markieren Sie nur ein Oval.

	1	2	3	4	5	
überhaupt nicht	◯	◯	◯	◯	◯	sehr

6. **Die angezeigten Variablenwerte sind nützlich bei der Fehlersuche**
   Markieren Sie nur ein Oval.

	1	2	3	4	5	
überhaupt nicht	◯	◯	◯	◯	◯	sehr

Abschluss

7. **Haben Sie noch abschließende Anmerkungen oder Hinweise zum Prototyp oder zur Studie?**

Im Namen des Lehrstuhls Informatik 11 der RWTH und des i11Freunde e.V. danken wir für Ihre Teilnahme.

Anhang B

Daten aus der Nutzerstudie

Für diese Arbeit wurde eine Nutzerstudie durchgeführt (Siehe Abschnitt 5). In diesem Anhang sind die Antworten der Probanden auf beide Teile des Fragebogens aufgeführt. Aus den Daten wurde zur Wahrung der Anonymität das Alter, Geschlecht und Freitext-Antworten der Probanden entfernt. Durch den geringen Stichprobenumfang wäre eine Anonymisierung der Daten bzw. der Probanden sonst nicht sichergestellt. Diese Informationen flossen jedoch in die Ergebnisbeschreibung in Abschnitt 5.3.2 mit ein. Die Fragen sind in den folgenden Tabellen verkürzt dargestellt, die Reihenfolge ist jedoch erhalten geblieben und entspricht der des jeweiligen Fragebogens aus Anhang A.

Proband	2	4	6	7	1	3	5
Gruppe		Test				Kontroll	
Programmierpraxis in Stunden pro Woche	>20	0 - 5	>20	>20	>20	5 - 10	>20
Programmiererfahrung (1=wenig, 5=viel)	5	3	4	3	3	4	3
Siemens Erfahrung (1=wenig, 5=viel)	5	1	1	2	4	2	1
Codesys Erfahrung (1=wenig, 5=viel)	1	4	5	5	2	5	4
Visual Studio Erfahrung (1=wenig, 5=viel)	2	4	4	3	3	4	3
Eclipse Erfahrung (1=wenig, 5=viel)	1	2	4	1	2	4	1
Erfahrung Statische Analyse	Nein	Ja	Ja	Ja	Nein	Ja	Nein
Wenn ja, welche?		Mitgelieferte Tools in VS	Static Analysis Light (Codesys)	Static Analyser von CodeSys		[mc]²	
Statische Analyse Produktiv eingesetzt	Nein	Ja	Nein	Nein	Nein	Nein	Nein
Programmiersprachen	61131-3, ST	61131-3, OOP, ST	61131-3, OOP, ST	61131-3, OOP	61131-3, ST	61131-3, OOP, ST	61131-3, ST
Schwierigkeit Aufgabe	3	2	1	3	2	2	1
Annotationen hilfreich	3	2	3	3	5	1	4
Meldung Erkennen und Beheben	4	3	2	4	1	1	3
Visu zum Debuggen hilfreich	4	5	4	4	4	2	2
Programmierfehler gemacht	Ja	Ja	Ja	Ja	Ja	Ja	Nein
Fehler in Vis gefunden	Ja	Ja	Ja	Nein	Ja	Nein	Ja

Proband	2	4	6	7	1	3	5
Gruppe		Test				Kontroll	
Schwierigkeit Aufgabe (5=schwer)	3	2	1	3	2	2	1
Annotationen hilfreich (5=stimme zu)	3	2	3	3	5	1	4
Meldung Erkennen und Beheben (5=stimme zu)	4	3	2	4	1	1	3
Visu zum Debuggen hilfreich (5=stimme zu)	4	5	4	4	4	2	2
Programmierfehler gemacht	Ja	Ja	Ja	Ja	Ja	Ja	Nein
Fehler in Vis gefunden	Ja	Ja	Ja	Nein	Ja	Nein	Ja
Meldungen verständlich (5=stimme zu)	4	4	4	4	3	3	4
Meldungen helfen beim Verständnis (5=stimme zu)	4	1	2	3	3	1	3
Meldungen nachvollziehbar (5=stimme zu)	3	5	2	4	3	4	3
Annotationen überraschend (5=stimme zu)	2	1	4	2	3	2	2
Arbeitsfluss gestört (5=stimme zu)	1	1	2	1	1	4	1
Meldungen inkonsistent (5=stimme zu)	1	1	2	1	1	1	1
Meldungen tatsächliche Fehler (5=stimme zu)	4	4	4	3	1	2	4
Programmierfehler in IDE	Nein	Nein	Nein	Nein	Nein	Nein	Nein
Programmfehler waren hinderlich (5=stimme zu)	2	1	1	1	1	1	1
Editor entspricht erwartungen (5=stimme zu)	4	4	4	4	4	2	4

Abbildung B.1: Antworten aller Teilnehmer auf Fragebogen Teil 1 wie abgebildet in A.1 (Daten erhoben in [Müllers, 2018])

Proband	2	4	6	7	1	3	5
Gruppe	Test				Kontroll		
Meldungen weisen auf Fehler hin (5=stimme zu)	5	4	4	4	5	2	1
Meldungen nützlich (5=stimme zu)	4	5	4	4	5	2	3
Variablenwerte zum Verständnis (5=stimme zu)	5	4	3	3	5	3	4
Variablenwerte zur Entwicklung (5=stimme zu)	4	3	3	2	5	3	3
Variablenwerte zur Fehlersuche (5=stimme zu)	5	4	2	4	5	3	4

Abbildung B.2: Antworten aller Teilnehmer auf Fragebogen Teil 2 wie abgebildet in A.2 (Daten wurden für diese Arbeit erhoben)

Aachener Informatik-Berichte

This list contains all technical reports published during the past four years. A complete list of reports dating back to 1987 is available from:

`http://aib.informatik.rwth-aachen.de/`

To obtain copies please consult the above URL or send your request to:

Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen,
Email: `biblio@informatik.rwth-aachen.de`

2017-01 *	Fachgruppe Informatik: Annual Report 2017
2017-02	Florian Frohn and Jürgen Giesl: Analyzing Runtime Complexity via Innermost Runtime Complexity
2017-04	Florian Frohn and Jürgen Giesl: Complexity Analysis for Java with AProVE
2017-05	Matthias Naaf, Florian Frohn, Marc Brockschmidt, Carsten Fuhs, and Jürgen Giesl: Complexity Analysis for Term Rewriting by Integer Transition Systems
2017-06	Oliver Kautz, Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe: CD2Alloy: A Translation of Class Diagrams to Alloy
2017-07	Klaus Leppkes, Johannes Lotz, Uwe Naumann, and Jacques du Toit: Meta Adjoint Programming in C++
2017-08	Thomas Gerlitz: Incremental Integration and Static Analysis of Model-Based Automotive Software Artifacts
2017-09	Muhammad Hamad Alizai, Jan Beutel, Jó Ágila Bitsch, Olaf Landsiedel, Luca Mottola, Przemyslaw Pawelczak, Klaus Wehrle, and Kasim Sinan Yildirim: Proc. IDEA League Doctoral School on Transiently Powered Computing
2018-02	Jens Deussen, Viktor Mosenkis, and Uwe Naumann: Ansatz zur variantenreichen und modellbasierten Entwicklung von eingebetteten Systemen unter Berücksichtigung regelungs- und softwaretechnischer Anforderungen
2018-03	Igor Kalkov: A Real-time Capable, Open-Source-based Platform for Off-the-Shelf Embedded Devices
2018-04	Andreas Ganser: Operation-Based Model Recommenders
2018-05	Matthias Terber: Real-World Deployment and Evaluation of Synchronous Programming in Reactive Embedded Systems
2018-06	Christian Hensel: The Probabilistic Model Checker Storm - Symbolic Methods for Probabilistic Model Checking
2019-02	Tim Felix Lange: IC3 Software Model Checking
2019-03	Sebastian Patrick Grobosch: Formale Methoden für die Entwicklung von eingebetteter Software in kleinen und mittleren Unternehmen

2019-05 Florian Göbe: Runtime Supervision of PLC Programs Using Discrete-Event Systems

2020-02 Jens Christoph Bürger, Hendrik Kausch, Deni Raco, Jan Oliver Ringert, Bernhard Rumpe, Sebastian Stüber, and Marc Wiartalla: Towards an Isabelle Theory for distributed, interactive systems - the untimed case

2020-03 John F. Schommer: Adaptierung des Zeitverhaltens nicht-echtzeitfähiger Software für den Einsatz in zeitheterogenen Netzwerken

2020-04 Gereon Kremer: Cylindrical Algebraic Decomposition for Nonlinear Arithmetic Problems

2020-05 Imke Drave, Oliver Kautz, Judith Michael, and Bernhard Rumpe: Pre-Study on the Usefulness of Difference Operators for Modeling Languages in Software Development

* These reports are only available as a printed version.
Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.