Entwicklung eines Compilers zur Programmierung von Sequenz-Generatoren

BACHELORARBEIT

für die Prüfung zum

Bachelor of Engineering

des Studienganges Infomationstechnik

an der Dualen Hochschule Baden-Württemberg Mannheim

von

Christoph Prinz

14. September 2015

Bearbeitungszeitraum	12 Wochen
Matrikelnummer, Kurs	2537478, TINF12ITIN
Ausbildungsfirma	Deutsches Zentrum für Luft- und Raumfahrt, Göttingen
Betreuer der Ausbildungsfirma	Dr. Robert Konrath
Gutachter der Dualen Hochschule	Prof. Dr. Rainer Colgen

Eidesstattliche Erklärung

gemäß § 5 (3) der „Studien- und Prüfungsordnung DHBW Technik“ vom 22. September 2011.

Ich habe die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Göttingen, den 11. September 2015

Zusammenfassung

Um Messgeräte in Echtzeit zu synchronisieren, werden beim Deutschen Zentrum für Luft- und Raumfahrt (DLR) programmierbare Sequenz-Generatoren eingesetzt. Diese werden mit Hilfe einer Software angesteuert, die die Steuerbefehle auf Grundlage einer Konfigurationsdatei erzeugt. Bei der Verwendung der Software treten jedoch Probleme auf, die nicht auf Basis der bisherigen Entwicklung behoben werden können.

Aus diesem Grund wird in dieser Bachelorarbeit ein Compiler für die Programmierung der Sequenz-Generatoren entwickelt. Bei der Entwicklung der Software liegt der Fokus auf Praktikabilität, Erweiterbarkeit und Qualitätssicherung.

Zur Programmierung der Sequenz-Generatoren wird eine domänenspezifische Sprache entwickelt. Auf Basis von praktischen Anwendungsfällen wird eine Sprachsyntax entworfen, welche die Problemdomäne der Ablaufsteuerung abstrahiert. Die Sprache wird in einer formalen Grammatik definiert, auf deren Basis ein zugehöriger Parser generiert wird.

Die Ausgabe des Parsers wird in einem Objektmodell verwaltet, welches die abstrakt programmierte Ablaufsteuerung in eine generische Form transformiert. Der konkrete Übersetzungsvorgang ist hardwareabhängig und wird im Rahmen dieser Arbeit anhand des Hardsoft Sequenz-Generators v5.1 demonstriert. Während des Übersetzungsvorgangs wird ebenfalls die technische Ausführbarkeit der programmierten Sequenz überprüft.

Um die Softwarequalität sicherzustellen, werden parallel zur Entwicklung Komponenten- und Integrationstests entworfen und durchgeführt. Das Ergebnis der Arbeit wird anhand eines Akzeptanztests evaluiert.

Abstract

To synchronize measurement equipment in real-time environments, the German Aerospace Center uses programmable sequence generators. They are triggered by software that generates control commands based on a configuration file. The usage of this software causes issues that cannot be solved by improving the current implementation.

The purpose of this bachelor thesis is the development of a new compiler to program sequence generators. The focus of the development is centered on practicability, extensibility and quality assurance.

To programm sequence generators, a domain specific language is developed. The specific syntax is designed based on practical demands and provides an abstract view on event-sequences. The language is specified in a formal grammar that is used to generate the compiler's parser.

An object model manages the parsed data and transforms it into a generic event list. The specific translation process depends on the target plattform. To demonstrate the compiler, the process is implemented for a Hardsoft sequence generator v5.1. The technical validation of an input sequence is performed during the translation process.

To address quality assurance, integration and component tests are executed simultaneously with the development process. Furthermore, the result of this thesis is evaluated on the basis of an acceptance test.

Inhaltsverzeichnis

Abbildungsverzeichnis **IX**

Tabellenverzeichnis **X**

Quellcodeverzeichnis **XI**

Abkürzungsverzeichnis **XII**

1 Einleitung **1**

1.1 Überblick . . . 1
1.2 Arbeitsumfeld . . . 2
1.2.1 Optische Messtechnik PIV . . . 2
1.2.2 Ablaufsteuerung . . . 3
1.3 Anforderungen . . . 4
1.3.1 Motivation . . . 4
1.3.2 Konkrete Anforderungen . . . 5
1.3.3 Rahmen der Arbeit . . . 6
1.4 Aufbau der Arbeit . . . 6

2 Stand der Technik **8**

2.1 Etabliertes Messkonzept . . . 8
2.1.1 Hardsoft Sequenz-Generator v5.1 . . . 8
2.1.2 Vorhandene Software . . . 11
2.2 Vergleichbare Arbeiten . . . 14
2.2.1 Proprietäre Komplettlösungen zur Sequenz-Generierung . . . 14
2.2.2 Objektmodelle zur Repräsentation von Sequenzen . . . 15
2.2.3 Domänenspezifische Sprachen zur Definition von Sequenzen . . . 17
2.2.4 Fazit . . . 20

3 Theoretische Grundlagen **21**

3.1 Formale Sprachen . . . 21
3.1.1 Grammatiken . . . 21
3.1.2 Sprachhierachie . . . 22
3.1.3 Darstellung einer Sprache in EBNF . . . 22

3.2 Compilerbau . . . 23
3.2.1 Übersetzungsschritte . . . 23
3.2.2 Werkzeugunterstützung . . . 28
3.3 Domänenspezifische Sprachen . . . 29
3.3.1 Einsatzgebiete . . . 30
3.3.2 Entwicklungsprozess . . . 31

4 Konzept **33**
4.1 Vorgehensmodell . . . 33
4.2 Aufteilung der Problemlösung in Teilsysteme . . . 34
4.3 Grammatik der Sprache . . . 35
4.3.1 Analyse der Problemdomäne . . . 36
4.3.2 Design der Sprachsyntax . . . 39
4.3.3 Globale Definitionen . . . 40
4.3.4 Definition der Sequenzen . . . 45
4.3.5 Aufbau der Sequenzdatei . . . 48
4.4 Parser . . . 48
4.4.1 Auswahl eines Parsergenerators . . . 50
4.4.2 Generieren des Objektmodells . . . 52
4.5 Laufzeitumgebung . . . 54
4.6 Optimierer und Übersetzter . . . 54
4.7 Hardwareansteuerung . . . 55
4.8 Zusammenfassung . . . 55

5 Umsetzung **57**
5.1 Parser . . . 57
5.1.1 Definition der Grammatik mit PyParsing . . . 57
5.1.2 Aufbau des Objektmodells . . . 58
5.2 Laufzeitumgebung . . . 59
5.2.1 Generieren der Symboltabellen . . . 60
5.2.2 Aktivieren von Menüeinträgen . . . 60
5.2.3 Auflösen von Ereignissen . . . 61
5.2.4 Auflösen von mathematischen Ausdrücken . . . 63
5.2.5 Anwenden der wait- und offset-Parameter . . . 64

5.3 Optimierer . . . 64
5.4 Übersetzer . . . 66
5.5 Hardwareansteuerung . . . 66
5.5.1 Einbindung der Hardwarefunktionen . . . 67
5.5.2 Verbinden eines Sequenz-Generators . . . 68
5.6 Hauptanwendung . . . 69

6 Test **71**
6.1 Grundlagen . . . 71
6.1.1 Fundamentaler Testprozess . . . 71
6.1.2 Arten von Tests . . . 72
6.2 Testkonzept . . . 72
6.2.1 Planung . . . 73
6.2.2 Analyse . . . 73
6.2.3 Entwurf . . . 73
6.2.4 Realisierung . . . 74
6.2.5 Durchführung . . . 75
6.3 Umsetzung der Komponenten- und Integrationstests . . . 75
6.3.1 Definition von Variablen, Aliasen und Makros . . . 76
6.3.2 Menüeinträge . . . 76
6.3.3 Definition von Sequenzen . . . 77
6.3.4 Berechnung mathematischer Ausdrücke . . . 79
6.3.5 Kompilieren einer Sequenz . . . 80
6.4 Umsetzung Akzeptanztest . . . 81
6.5 Ergebnisse . . . 83

7 Fazit **85**
7.1 Zusammenfassung . . . 85
7.2 Evaluation . . . 86
7.3 Ausblick . . . 87

Literaturverzeichnis **XIII**

Anhang **XVII**

A Gewünschte Funktionalität anhand einer Beispielsequenz **XVIII**

B EBNF der bisherigen Syntax **XX**

C Grammatik der Sprache und das abgebildete Objektmodell **XXII**

C.1 Allgemeine Definitionen XXII
C.2 Mathematische Operationen XXII
C.3 Bereich *variables* XXIII
C.4 Bereich *settings* XXIII
C.5 Bereich *channels* XXIV
C.6 Bereich *menu* XXV
C.7 Events im Bereich der Sequenz- und Makrodefinition XXVI
C.8 Hauptanwendung Seco2015 XXVI

Abbildungsverzeichnis

1 PIV Versuchsaufbau 2
2 Timingdiagramm zur Koordinierung von PIV-Messtechnik 4
3 Frontansicht eines Hardsoft Sequenz-Generators v5.1 9
4 Zustandsübergangsdiagramm des Sequenz-Generators im *Restart*-Mode . . . 10
5 Zustandsübergangsdiagramm des Sequenz-Generators im *Alternative*-Mode . 10
6 GUI der Ablaufsteuerungssoftware Seco2004 zur Konfiguration von Variablen, Kanälen und Betriebsmodi im Normal-Mode 13
7 Proprietäre Hardware zur Generierung von Steuerungssequenzen 15
8 Beispiel für ein UML-Timingdiagramm 16
9 Phasen eines Übersetzungsvorgangs 24
10 Syntaxbaum einer Sequenz 26
11 Dekorierter Syntaxbaum als Ergebnis der semantischen Analyse eines Syntaxbaumes 26
12 UML-Klassendiagramm des Bereiches *variables* im Objektmodell 53
13 Übersicht der Teilsysteme des Compilers 56
14 UML-Klassendiagramm der Laufzeitumgebung 60
15 UML-Klassendiagramm der Ereignisse 62
16 UML-Klassendiagramm der mathematischen Ausdrücke im Objektmodell . . . 63
17 UML-Klassendiagramm des Optimierers 65
18 UML-Klassendiagramm der Hardwareansteuerung 67
19 Testaufbau für den Akzeptanztest 82
20 Vergleichsparameter des Akzeptanztests 82
21 UML-Klassendiagramm des Bereiches *settings* XXIII
22 UML-Klassendiagramm des Bereiches *channels* XXIV
23 UML-Klassendiagramm des Bereiches *menu* XXV
24 UML-Klassendiagramm der Hauptanwendung XXVI

Tabellenverzeichnis

1 Technische Spezifikation des Sequenz-Generators v5.1 9
2 Regeln zur Definition einer Grammatik in EBNF-Notation 23
3 Auswirkung der Parameter `wait` und `offset` in verschiedenen Anweisungen . 47
4 Kodierung eines Ereignisses für den Programmbibliothek-Aufruf des Sequenz-Generators 66
5 Umwandlung von Pascal-Datentypen in `ctypes` Instanzen 68
6 Testfälle zu Definitionen in der *Sequenzdatei* 76
7 Testfälle zur Menü-Definition in der *Sequenzdatei* 77
8 Testfälle zur Definition von Sequenzen in der *Sequenzdatei* 77
9 Testfälle zur Definition von Makros in der *Sequenzdatei* 78
10 Testfälle zur Definition von For-Schleifen in der *Sequenzdatei* 79
11 Testfälle zum Überprüfen von Ausnahmefehlern bei Verwendung von zuvor nicht definierten Bezeichnern 79
12 Testfälle zum Überprüfen der Berechnung mathematischer Ausdrücke 80
13 Testfälle zum Validieren des Compilers 81
14 Codeabdeckung bei Ausführung der Komponenten- und Integrationstests . . 84

Quellcodeverzeichnis

1 *Sequenzdatei* für die Koordinierung von Messtechnik 12

2 Tokenfolge als Ergebnis der lexikalischen Analyse einer Zeichenfolge 25
3 Beispielgrammatik für eine Sequenz 25
4 Zwischencode einer Eventsequenz 27
5 Optimierter Zwischencode einer Eventsequenz 27

6 Beispiel für den Bereich *variables* nach neuer und alter Sprachdefinition . . . 41
7 Beispiel für den Bereich *settings* nach neuer Sprachdefinition 43
8 Beispiel für den Bereich *channels* nach neuer Sprachdefinition 43
9 Beispiel für den Bereich *menu* nach neuer Sprachdefinition 44
10 Beispiel für den Bereich *sequence* nach neuer Sprachdefinition 48
11 PyParsing Syntax zur Definition der Grammatik zum Bereich *variables* 53

12 PyParsing Syntax zur Definition der Grammatik zum Bereich *variables* 58

13 *Sequenzdatei* zum Durchführen des Akzeptanztests 83

Abkürzungsverzeichnis

API	Application Programming Interface
AST	Abstract Syntax Tree
ANTLR	Another Tool for Language Recognition
AST	Abstract Syntax Tree
BOS	Background Oriented Schlieren Method
BNF	Backus-Naur-Form
EBNF	Erweiterte Backus-Naur-Form
DLL	Dynamic Link Library
DLR	Deutsches Zentrum für Luft- und Raumfahrt
DSL	Domain Specific Language
GPL	General Purpose Language
GUI	Graphical User Interface
LabView	Laboratory Virtual Instrumentation Engineering Workbench
LEX	Lexical Analyzer
NI	National Instruments
TTL	Transistor-Transistor-Logik
UML	Unified Modeling Language
PIV	Particle Image Velocimetry
PLY	Python LEX YACC
YACC	Yet Another Compiler Compiler

1 Einleitung

Im Rahmen dieser Arbeit wird ein Compiler zur Ansteuerung eines Sequenz-Generators entwickelt. Dieses Kapitel dient der Einleitung in die Arbeit und gibt zunächst in Abschnitt 1.1 einen thematischen Überblick über die Arbeit. In Abschnitt 1.2 wird das Arbeitsumfeld, indem die Arbeit entstanden ist, vorgestellt. Auf dieser Basis werden in Abschnitt 1.3 die Anforderungen definiert, die die Grundlage für die Bearbeitung der Aufgabenstellung bilden. Abschnitt 1.4 zeigt die Struktur dieser Arbeit auf.

1.1 Überblick

Das Deutsche Zentrum für Luft- und Raumfahrt (DLR) ist das nationale Forschungszentrum der Bundesrepublik Deutschland für Luft- und Raumfahrt. Es setzt Forschungs- und Entwicklungsarbeiten in den Bereichen Luftfahrt, Raumfahrt, Energie, Verkehr und Sicherheit um. Ferner ist das DLR als Raumfahrtagentur im Auftrag der Bundesregierung für die Planung und Umsetzung der deutschen Raumfahrtaktivitäten zuständig.

Am Standort Göttingen beschäftigt sich die Abteilung *Experimentelle Verfahren* des Instituts für Aerodynamik und Strömungstechnik mit der Entwicklung und dem Einsatz von akustischen und optischen Feldmessverfahren. Unter der Verwendung von Hochgeschwindigkeitskamerasystemen werden optische Messtechniken wie die *Particle Image Velocimetry (PIV)* (Siehe Abschnitt 1.2.1) zur Charakterisierung von Strömungen angewandt.

Zur Echtzeit-Koordinierung der Messausrüstung werden Sequenz-Generatoren von Hardsoft verwendet [1, Kapitel 2]. Für optische Messungen wie PIV ist die hoch präzise Koordination der Laser- und Kamerasysteme erforderlich. Die Schaltreihenfolge der Ausgänge, mit denen die Messgeräte getriggert werden, kann mit Hilfe einer Programmbibliothek an den Sequenz-Generator übertragen werden. Derzeit wird die hausintern entwickelte Software *Seco2004* für die Ablaufsteuerung solcher Messungen verwendet. Die Sequenz wird mit Hilfe eines Konfigurationsskripts vorgegeben, welches von einem handgeschriebenem Parser verarbeitet wird [2, Kapitel 6]. Da dieser diverse Mängel und Inkonsistenzen aufweist, soll er im Rahmen dieser Bachelorarbeit von einem auf Basis einer formalen Grammatik, automatisiert, erzeugten Parser abgelöst werden.

1.2 Arbeitsumfeld

Im Folgenden wird das Arbeitsumfeld und Einsatzgebiet der Anwendung vorgestellt und die besonderen Anforderungen erläutert.

1.2.1 Optische Messtechnik PIV

Bei PIV handelt es sich um eine optische Messtechnik zur Charakterisierung von Strömung in Windkanalmessungen. Die Messtechnik ermöglicht es, die momentane Strömung in einer Ebene als zwei- oder dreidimensionales[1] Geschwindigkeitsvektorfeld zu bestimmen. Um die Strömung optisch erfassbar zu machen, wird sie mit Streupartikeln[2] wie Öl oder Eis angereichert.

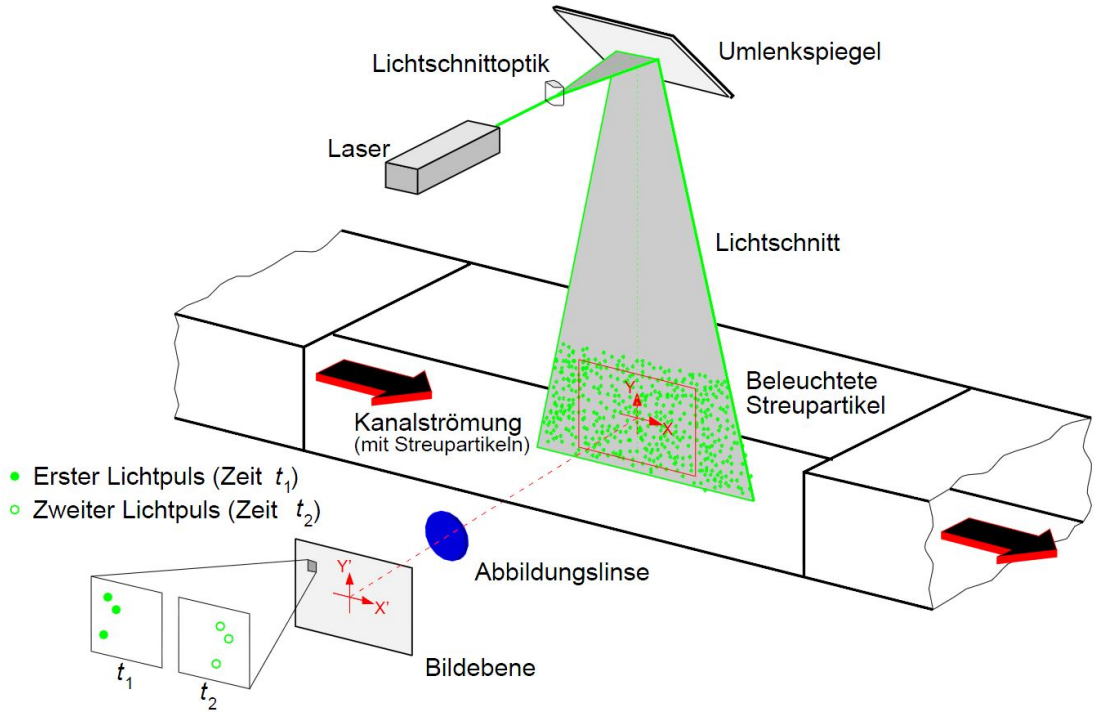


Abbildung 1: PIV Versuchsaufbau [3]

[1] Möglich durch die Verwendung von zwei Kameras (Stereo-PIV)
[2] Auch als Seeding bezeichnet

Abbildung 1 stellt die Funktionsweise sowie den Versuchsaufbau einer zweidimensionalen PIV-Messung dar. Mit Hilfe des Pulslasers wird eine Ebene der Strömung in einem geringem Zeitabstand Δt zweimal illuminiert und von einer speziellen CMOS-Kamera aufgenommen. Die vom Laser erreichte Beleuchtungsintensität ermöglicht eine sehr kurze Belichtungszeit, sodass die Partikel auch bei hohen Strömungsgeschwindigkeiten ohne Bewegungsunschärfe erfasst werden können. Mit Hilfe eines Güteschalters[3] wird die Intensität sowie die Pulsdauer des Lasers beeinflusst.

Sind die Partikel hinreichend klein, folgen sie der Strömung schlupffrei. Die Korrelation der beiden Bilder ermöglicht eine Berechnung der Versatzvektoren, deren Betrag Δs entspricht. Üblicherweise wird dieser Betrag für einen definierten Teilbereich gemittelt. Mit Hilfe der physikalischen Definition der Geschwindigkeit $\frac{\Delta s}{\Delta t}$ kann die lokale Strömungsgeschwindigkeit ausgewertet werden. [4, Kapitel 2]

1.2.2 Ablaufsteuerung

Zur Koordinierung von Lasern, Güteschaltern und Kameras kommt bei einer PIV Messung ein Sequenz-Generator zum Einsatz. Der Sequenz-Generator ist ein TTL[4] Impulsgenerator, für dessen Kanäle Impulsfrequenz, -anzahl und -breite programmierbar sind.

Unterschiedliche Strömungsgeschwindigkeiten, Seedinggrößen sowie Kameratypen erfordern bei optischen Messtechniken eine Anpassung der Ablaufsteuerung. Bei einer PIV Messung ist es beispielsweise erforderlich, dass mit der Programmierung des Sequenz-Generators folgende Parameter konfiguriert werden:

- Abstand der Laserimpulse sowie Auslöseverzögerung zwischen zwei Bildern
- Abstand zwischen der Ansteuerung von Laseranregung und Güteschalter zur Beeinflussung von Dauer und Intensität eines Laserpulses
- Belichtungszeit und Auslösefrequenz der Kamera(s)

Abbildung 2 zeigt das Ablaufdiagramm einer typischen PIV-Konfiguration mit einer Auflösung von 20 μs. Der *Input-Trigger* dient als Referenzzeitpunkt für die Generierung der Impulse.

[3] Auch als *Q-Switch* bezeichnet

[4] Bei der Transistor-Transistor-Logik (TTL) gilt $U_E < 0,8\ V$ als Low-Pegel, und $U_E > 2,0\ V$; $U_E \approx 5\ V$ als High-Pegel.

Da die *pco.edge*-Kamera eine Auslöseverzögerung hat, wird sie als erstes angesteuert. Die Güteschalter dienen der Bündelung des Lichtes. Sie werden jeweils nach den zugehörigen Blitzlichtern oder Lasern ausgelöst, um die beiden Aufnahmen zu beleuchten. Die Kamera unterstützt den sogenannten *Double-Shutter-Mode*, der dazu dient automatisiert zwei Bilder kurz hintereinander aufzunehmen.

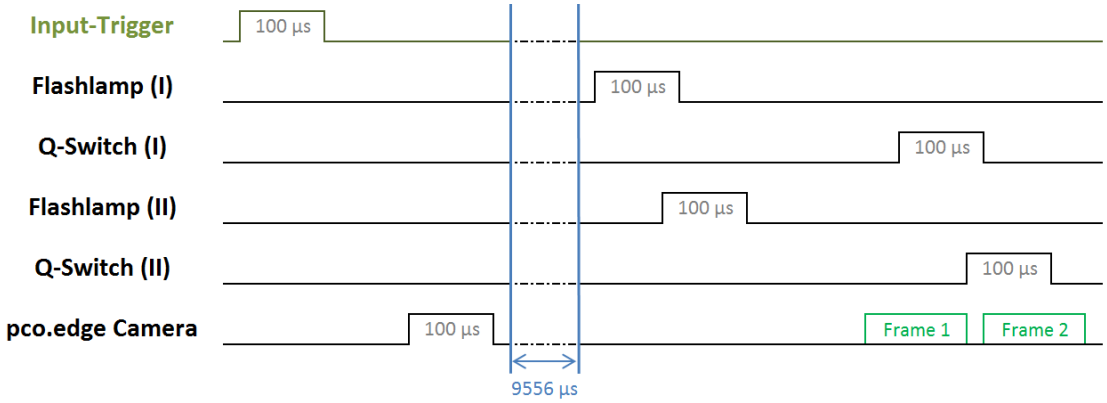


Abbildung 2: Timingdiagramm zur Koordinierung von PIV-Messtechnik

Im DLR wird die in Abschnitt 2.1.2 vorgestellte Software *Seco2004* genutzt, um eine solche Ablaufsteuerung zu programmieren und an einen Hardsoft Sequenz-Generator zu übertragen. Durch die Verwendung von Lasern der Klasse vier[5] kann die Ablaufsteuerung als sicherheitskritischer Prozess charakterisiert werden. Zum Einsatz kommende Software muss daher umfangreich getestet werden, damit Fehlansteuerungen ausgeschlossen werden können.

1.3 Anforderungen

Im Folgenden werden, basierend auf der Aufgabenstellung und der Motivation der Arbeit, konkrete Anforderungen sowie ein konkretes Ziel der Entwicklung festgelegt.

1.3.1 Motivation

Grundsätzlich ist *Seco2004* zur Konfiguration des Sequenz-Generators für optische Messungen etabliert. Die Ablaufsteuerung wird vom Nutzer in einer textbasierten Konfigurationsdatei[6]

[5] nach DIN EN 60825-1
[6] Im Folgenden als *Sequenzdatei* bezeichnet

implementiert und anschließend von der Software interpretiert.

Die Sprache ist jedoch an einigen Stellen nicht intuitiv und inkonsistent, sodass die Verwendung unkomfortabel und fehleranfällig ist. Ferner wird nur eingeschränkt validiert, ob eine definierte Sequenz ausführbar ist, sodass es teilweise zu unerwarteten Ausführungszuständen kommt. In Folge dessen, dass der eingesetzte Parser handgeschrieben ist, sind Wartung und Erweiterung von *Seco2004* aufwändig und aus Sicht der Softwarequalität nicht sinnvoll.

Aus diesem Grund ist eine Neuentwicklung notwendig, die an das bisher bestehende Programm angelehnt ist. Bei der Neuentwicklung wird nach einer fundierten Methodik vorgegangen.

1.3.2 Konkrete Anforderungen

Die Kernanforderung der Bachelorarbeit liegt darin, die Benutzung der *Sequenzdatei* zu vereinfachen, sowie die Qualität der generierten Sequenzen und der Hauptanwendung im Allgemeinen zu erhöhen.

Zur Erfüllung dieser Anforderungen wird unter dem Arbeitstitel *Seco2015* eine Konfigurationssprache und ein dazugehöriger Compiler entwickelt. An die neue Sprachdefinition sowie den Parser der *Sequenzdatei* werden folgende konkreten Anforderungen gestellt:

- Vereinfachung der Programmierung
- Erhöhen der Übersichtlichkeit und Lesbarkeit
- Verminderung der Fehleranfälligkeit generierter Sequenzen

An die Hauptanwendung bzw. an den Compiler werden folgende konkreten Anforderungen gestellt:

- Ermöglichen von einfacher Erweiterung und Wartung durch andere Entwickler
- Vorbereitung der Architektur für zukünftige Erweiterungen
- Sicherstellen von Softwarequalität um Sicherheitsrisiken auszuschließen

1.3.3 Rahmen der Arbeit

Im Rahmen dieser Bachelorarbeit soll ein Compiler entwickelt werden, mit dessen Hilfe sich die Schaltvorgänge des Sequenz-Generators programmieren lassen. *Seco2015* soll als Kommandozeilen-Applikation in Python entwickelt werden, mit der die Funktionsfähigkeit des Compilers demonstriert werden kann.

Aus der Aufgabenstellung lässt sich für die Entwicklung folgender Rahmen abgrenzen:

- Formale Definition der in *Seco2004* verwendeten Syntax in Backus-Naur-Form (BNF)[7]
- Umsetzung der Syntax mit einem automatisiert generiertem Parser
- Implementierung eines Compilers zur Erzeugung der Ablaufsteuerung
- Erweiterung der Syntax zum Erreichen zusätzlicher Funktionalität
- Software- und Systemtest

Die Sprachdefinition soll es ermöglichen, dass aus der *Sequenzdatei* eine Sequenz abgeleitet werden kann, die ohne weitere Konfigurationen an den Sequenz-Generator gesendet werden kann. Sind mehrere Sequenz-Generatoren an den Messrechner angeschlossen, soll der Compiler automatisch ein Gerät auswählen. Nach der Übertragung der Sequenz soll der Sequenz-Generator automatisch in den ausführungsbereiten Zustand versetzt werden, sodass die Ausführung vom Anwender auf Hardwareebene ausgelöst werden kann.

Es soll keine Möglichkeit geben, die eingelesenen Werte in einer Graphical User Interface (GUI) anzuzeigen oder anzupassen. Ferner soll kein Editor oder eine Entwicklungsumgebung für das Bearbeiten der *Sequenzdatei* entwickelt werden.

1.4 Aufbau der Arbeit

Diese Bachelorarbeit beschreibt die Bearbeitung der Aufgabenstellung, beginnend mit der Analyse des aktuellen Standes über den Entwurf eines Konzeptes bis hin zu dessen Umsetzung. Ferner wird das Ergebnis der Implementierung im Anwendungskontext getestet.

[7] Siehe Abschnitt 3.1.3

In Kapitel 2 wird ein Überblick über den aktuellen Stand der Technik gegeben. Dabei wird auf die vorhandene Software sowie vergleichbare Arbeiten eingegangen.

Die für die Bearbeitung der Schwerpunkte notwendigen theoretischen Grundlagen werden in Kapitel 3 eingeführt. In Kapitel 4 wird ein Konzept für die Erfüllung der Anforderungen erstellt.

Auf Basis des Konzeptes wird die Umsetzung der Problemlösung in Kapitel 5 beschrieben. Der praktische Teil der Arbeit wird mit einem Komponenten- sowie einem Akzeptanztest in Kapitel 6 abgeschlossen.

Kapitel 7 bildet den Schlussteil der Arbeit und bündelt gesammelte Ergebnisse. Es findet eine Evaluation der implementierten Lösung statt und es wird ein Ausblick auf Weiterentwicklungsmöglichkeiten gegeben.

2 Stand der Technik

Im folgenden Kapitel wird der Stand der Technik im Kontext der Abteilung sowie im Allgemeinen reflektiert. Dazu wird zunächst in Abschnitt 2.1 auf das im DLR etablierte System zur Sequenzgenerierung eingegangen. In Abschnitt 2.2 werden vergleichbare Arbeiten und Lösungsansätze gesammelt und hinsichtlich ihrer Praktikabilität evaluiert.

2.1 Etabliertes Messkonzept

Eine Aufgabe der Abteilung *Experimentelle Verfahren* ist die Anwendung sowie die Weiterentwicklung von Messtechniken. Die für die Ablaufsteuerung der Messgeräte zur Anwendung kommenden Hard- und Softwarekomponenten werden im Folgenden vorgestellt.

2.1.1 Hardsoft Sequenz-Generator v5.1

Das Messequipment wird an einen Sequenz-Generator von *Hardsoft* angeschlossen. Er ermöglicht eine programmierbare, hoch präzise Ablaufsteuerungen von echtzeitfähiger Hardware. Der Sequenz-Generator verfügt über 16 Anschlüsse, deren Ausgangssignale TTL-Logik entsprechen. Abbildung 3 zeigt die Bedienelemente des Sequenz-Generators.

Die Steuersequenz kann über die Bedienelemente auf dem Sequenz-Generator oder über eine PC-Software definiert werden. Ferner ermöglicht eine Softwarebibliothek das Ansprechen des Sequenz-Generators über die serielle Schnittstelle RS-232, welche auch von der im DLR eingesetzten Anwendung *Seco2004* (Siehe Abschnitt 2.1.2) genutzt wird. Die Schaltzeitpunkte der Ausgänge können im Rahmen der in Tabelle 1 gezeigten technischen Spezifikation beliebig definiert werden.

Ein Sequenz-Generator kann sich in folgenden Zuständen befinden:

- **STOP**: Starten der Sequenz nicht möglich
- **WAIT**: Bereitschaft zum Ausführen der Sequenz
- **RUN**: Ausführen der Sequenz

Der Sequenz-Generator unterstützt sechs verschiedene Betriebsmodi mit unterschiedlichem Transitionsverhalten der Zustände, ausgelöst vom *Trigger*- und *Arming*-Eingang sowie dem

Abbildung 3: Frontansicht eines Hardsoft Sequenz-Generators v5.1

Technische Eigentschaft	**Sequenz-Generator v5.1**
maximale Auflösung	50 ns
minimale Auflösung	3 ms
Jitter (Genauigkeitsschwankung der Impulsbreite)	12.5 ns
maximale Zeitdifferenz	100 s
Anzahl der Sequenzen	100
Anzahl an Impulsen	32000
Ausgangskanäle	16
Triggerquellen	4

Tabelle 1: Technische Spezifikation des Sequenz-Generators v5.1

Start- und *Stop*-Aufruf des Application Programming Interface (API).

Am häufigsten werden die Modi *Restart* und *Alternative* genutzt. Die Zustandsübergänge im *Restart*-Modus sind in Abbildung 4 dargestellt. Das *Start*-Signal der API versetzt den Sequenz-Generator in den *Wait*-Zustand. Sobald der Trigger ausgelöst wird, startet die Sequenz, welche nach Beendigung erneut ausgeführt wird, sobald ein Triggersignal anliegt.

Das *Stop*-Signal der API versetzt den Sequenz-Generator wieder in den *Stop*-Modus.

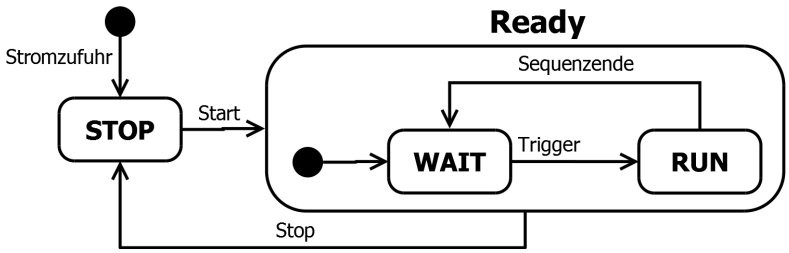


Abbildung 4: Zustandsübergangsdiagramm des Sequenz-Generators im *Restart*-Mode

Die Zustandsübergänge des *Alternative*-Mode sind in Abbildung 5 dargestellt. Dem Sequenz-Generator liegt eine Hautpsequenz (SEQ0) und eine Nebensequenz (SEQ1) vor. Befindet sich der Sequenz-Generator im initialen *Wait*-Zustand, kann mit einem Triggersignal die Hauptsequenz und mit einem Armingsignal die Nebenfrequenz beim nächsten Trigger gestartet werden. Wird während der Ausführung der Hauptsequenz ein *Arming*-Signal ausgelöst, kann mit einem Trigger nach Ende der Hauptsequenz die Nebensequenz gestartet werden.

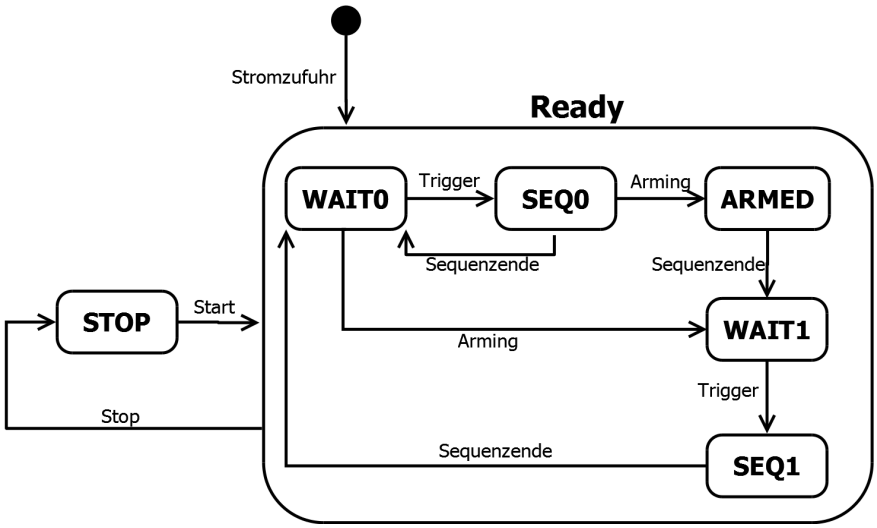


Abbildung 5: Zustandsübergangsdiagramm des Sequenz-Generators im *Alternative*-Mode

Neben dem für diese Arbeit verwendeten Hardsoft Sequenz-Generator v5.1 gibt es weitere Hardwarerevisionen, die sich in ihren technischen Eigenschaften sowie den verfügbaren Betriebsmodi unterscheiden. Bei Sequenz-Generatoren der älteren Hardwarerevision v3.1 können

beispielsweise nur für 8 Kanäle beliebige Impulsbreiten gewählt werden, während die verbleibenden 16 Kanäle nur 50 ns oder 100 ns breite Flanken ausgeben können. Außerdem wird der *Alternative*-Modus nicht unterstützt und Sequenzen können nur 8000 Impulse enthalten. Die zur Verfügung stehende API ist nur mit der Hardwarerevision v5.1 kompatibel. Ältere Sequenz-Generatoren müssen im Gegensatz dazu mit Low-Level Befehlen über die serielle Schnittstele RS-232 angesprochen werden. [5, Kapitel 2-6]

2.1.2 Vorhandene Software

Zur Ansteuerung des Sequenz-Generators steht die Herstellersoftware sowie die im DLR entwickelte Anwendung *Seco2004* zur Verfügung, welche im Folgenden vorgestellt werden.

Hardsoft Sequenz RS

Die Software des Herstellers bietet grundlegenden Möglichkeiten, eine Ablaufsteuerung mit Hilfe einer GUI zu programmieren. Sie ist nur für sehr einfache Sequenzen geeignet, da Zeitabstände nicht als Variablen definiert werden können. Infolgedessen kommt sie beim DLR nicht zum Einsatz und wird im Rahmen dieser Arbeit nicht weiter betrachtet.

Seco2004

Seco2004 ist eine DLR-interne Software, welche in C++ entwickelt wurde, um die mangelnde Funktionalität der Herstellersoftware auszugleichen und somit den Sequenz-Generator für Messungen komfortabel programmieren zu können. Sie stellt eine Neu- bzw. Weiterentwicklung der bisher im DLR eingesetzten Anwendungen *Seco95*, *Seco2000* und *Seco2002* dar, welche aus gleicher Motivation entwickelt wurden, aber dennoch diverse Einschränkungen, Mängel oder Inkompatibilitäten enthielten. [2, Abschnitt 2 und 3]

In *Seco2004* wird die Ablaufsteuerung vom Anwender in einem Konfigurationsskript (*Sequenzdatei*) programmiert. Dieses ist in die Bereiche *variables*, *channels*, *menus* und *sequence* untergliedert. Mit der Anwendung *Seco2004* kann das Skript interpretiert, angepasst und an den Sequenz-Generator übertragen werden. Anhand der *Sequenzdatei* in Quelltext 1 wird im Folgenden die Programmierung einer Sequenz erläutert.

```
variables {
  a, "Tau [us]",           300,     2, 2000;
  c, "FL-QS Delay (I)",    499,   270,  499;
  d, "FL-QS Delay (II)",   499,   270,  499;
  e, "PCO Delay",            0,  -100,  100;
  f, "Shift [us]",           0,     0, 9000;
  }
channels {
  1, "Flashlamp (I)";
  2, "Q-switch  (I)";
  3, "Flashlamp (II)";
  4, "Q-switch  (II)";
  5, PCO1;
  6, PCO2;
  7, "Set Box", X;
  8, "Reset Box", X;
  }
menus {
  "High Energy",      c=279, d=279;
  "Low Energy",       c=460, d=460;
  }
sequence {
  5, 300000 + f - 200 + e, 202; # PCO_1
  6, 300000 + f - 200 + e, 202; # PCO_2
  1, 300000 + f - c, 50;
  3, 300000 + f - d + 4.000 * a,50;
  2, 300000 + f, 50;
  4, 300000 + f + 4.000 * a,50;
  1, 400000 + f - c, 50;
  3, 400000 + f - d + 3.000 * a,50;
  1, 500000 + f - c, 50;
  3, 500000 + f - d + 2.000 * a,50;
  }
```

Quellcode 1: *Sequenzdatei* für die Koordinierung von Messtechnik

Die konkrete Ablaufsteuerung wird im Abschnitt *sequence* programmiert. Dabei repräsentiert jede Zeile ein Ereignis im Format:

```
<Kanalnummer>, <Schaltzeitpunkt>, <Schaltdauer>;
```

Die Ereignisse müssen nicht chronologisch sortiert werden und können sich überschneiden. Die verbleibenden Bereiche der *Sequenzdatei* dienen der Konfiguration und werden in die *Seco2004*-GUI, die in Abbildung 6 gezeigt ist, übernommen.

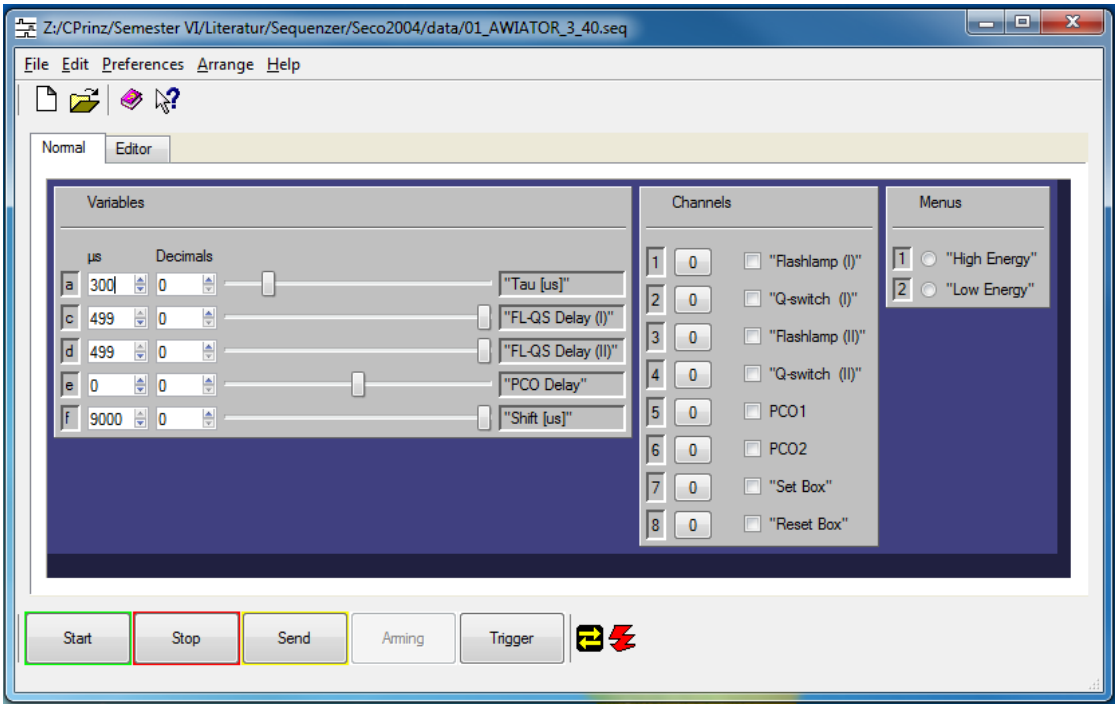


Abbildung 6: GUI der Ablaufsteuerungssoftware Seco2004 zur Konfiguration von Variablen, Kanälen und Betriebsmodi

Im Bereich *variables* werden Variablen mit einem Wertebereich sowie dem Momentanwert zur späteren Verwendung definiert. Im Abschnitt *channels* werden Kanäle benannt und temporär deaktiviert. Im Bereich *menu* werden durch Veränderung der Variablen verschiedene Messmodi definiert. In der GUI wird ausgewählt, welcher Messmodus aktiv sein soll. Ferner können Variablenwerte manuell angepasst werden, bevor die Sequenz generiert und an den Sequenz-Generator übertragen wird.

Seco2004 verfügt über einen *Normal*- und einen *Editor*-Modus. Ersterer erlaubt die beschriebene Modifikation der *variables*, *channels* und *menu* Bereiche auf GUI Ebene, während der Zweite einen Editor zur Modifikation der *Sequenzdatei* bereitstellt. Hier kann auch der *sequence* Bereich und damit das Grundgerüst der Ablaufsteuerung modifiziert werden.

Die Sprache der *Sequenzdatei* ist nicht formal definiert und weist Inkonsistenzen auf. Inkonsistent ist zum Beispiel, dass mathematische Ausdrücke und Variablen nur beim Schaltzeitpunkt

und nicht bei der Impulsdauer angegeben werden können. Außerdem fehlt die Unterstützung von Strukturierungswerkzeugen wie Schleifen oder Makros, sodass Nutzer bei der Programmierung des *sequence* Bereiches häufig Teilsequenzen kopieren und einfügen müssen. Dies erfordert die Anpassung der Schaltzeitpunkte, was eine häufige Fehlerquelle darstellt. Ferner ist nicht sichergestellt, dass eine vom Nutzer programmierte Sequenz hardwareseitigen Limitierungen entspricht. [2, Kapitel 6]

2.2 Vergleichbare Arbeiten

Neben der in der Abteilung bisher eingesetzten Strategie, einen Sequenz-Generator textbasiert zu programmieren, gibt es verschiedene Ansätze eine Ablaufsteuerung zu realisieren. Die folgenden Abschnitte spiegeln den Stand vergleichbarer Arbeiten wider und evaluieren sie im Kontext der Aufgabenstellung.

2.2.1 Proprietäre Komplettlösungen zur Sequenz-Generierung

Das Unternehmen TSI stellt eine Komplettlösung zur Durchführung optischer Messungen bereit. Die Ablaufsteuerung wird dabei von der zentralen Steuerungssoftware *Insight 4G* übernommen, die auch die Archivierung und Auswertung der Aufnahmeserien koordiniert. Die Software steuert den in Abbildung 7a dargestellten Sequenz-Generator[1] automatisch an. Sie kann über allgemeine Parameter angepasst werden und ermöglicht somit eine einfache Nutzung, während gleichzeitig Hardwarespezifikationen eingehalten werden. Problematisch ist jedoch, dass nur entsprechende Hardware des Herstellers verwendet werden kann und ein konkreter Eingriff in die Ablauffolge seitens des Nutzers nicht möglich ist. Dem Anspruch der Abteilung, Messtechniken weiterzuentwickeln, kann eine solche Komplettlösung nicht gerecht werden. [6] [7, Abschnitt II A]

Verbesserungen an den Messverfahren werden nur erreicht, wenn die Ablaufsteuerung von Forschern manuell programmiert wird. Die Programmiersprache *LabVIEW* von National Instruments (NI) hat sich im Allgemeinen bei der Koordination von Messgeräten etabliert und kommt auch bei anderen Softwareprojekten der Abteilung zum Einsatz [9] [10]. NI

[1] Hier als „Synchronizer“ bezeichnet

(a) LaserPulse Synchronizer 610036 [6] (b) NI Signalgenerator PXIe-5450 [8]

Abbildung 7: Proprietäre Hardware zur Generierung von Steuerungssequenzen

bietet passende Hardware zur Signalgenerierung (Siehe Abbildung 7b) sowie die LabVIEW-Softwarebibliothek „Modulation Toolkit“ an. Diese Umgebung ermöglicht ein komfortables Realisieren der Ablaufsteuerung auf Ebene des LabVIEW-Quellcodes [11] [12]. Jedoch besteht Abhängigkeit von NI Hardware, die nicht genug Kanäle unterstützt, um Messungen mit vier Kameras durchzuführen. Soll ferner am bisherigen Konzept mit einer Konfigurationsdatei festgehalten werden, muss der Parser in LabVIEW ebenfalls handgeschrieben oder durch Einbindung einer Bibliothek umgesetzt werden, da keine entsprechenden Parsergeneratoren für LabVIEW existieren.

Im Allgemeinen sind proprietäre Lösungen somit aufgrund der Hardwarerestriktionen nicht für die Koordinierung des Messequipments geeignet. Es wird eine Lösung gesucht, die mit dem bisherigen Sequenz-Generator zusammenarbeiten kann. Da dieser über eine Softwarebibliothek ansprechbar ist, besteht die Herausforderung darin, die Sequenz in einer geeigneten Form zu abstrahieren, und anschließend in Steuerungsbefehle der Softwarebibliothek umzuwandeln.

2.2.2 Objektmodelle zur Repräsentation von Sequenzen

Die Ablaufsteuerung basiert auf Ereignissen, die angeben, wann sich der Pegel eines Kanals ändert. Diese Parameter können auf ein Objektmodell abstrahiert und schließlich in einer

Modellierungssprache abgebildet werden.

In Abbildung 8 ist ein Beispiel für die Modellierung eines zeitlichen Ablaufs in Unified Modeling Language (UML) dargestellt. Auf einer Zeitachse (TimeLine) können Zustände (States) modelliert werden. Mit Hilfe von Nachrichten (Messages) können automatische Zustandsänderungen umgesetzt werden.

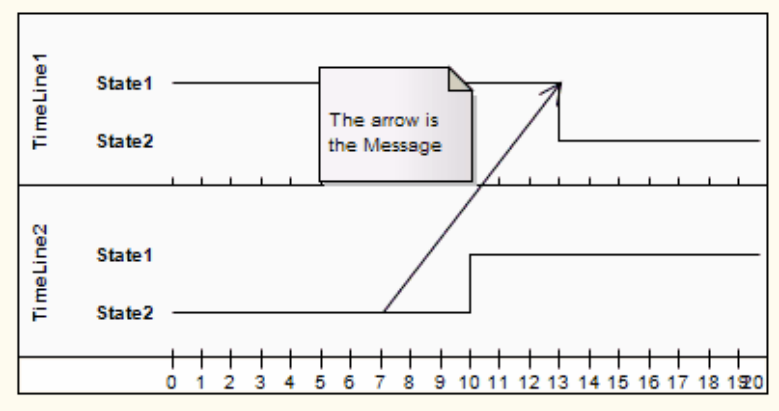


Abbildung 8: Beispiel für ein UML-Timingdiagramm [13]

In [14, Kapitel 3] wird beschrieben, wie auf Basis dieses UML-Profils ein *Event-Condition-Action* Ansatz umgesetzt werden kann. Die Zeitachsen treten als Objekte auf, zwischen denen Abhängigkeiten definiert wird. In einem solchen UML-Objekt wird definiert, bei welchem Ereignis unter bestimmten Bedingungen eine definierte Handlung ausgelöst wird. Die restliche Konfiguration kann mit Hilfe eines eigenen UML-Profils abgebildet werden. Für die Modellierung von UML existiert eine breite Werkzeugunterstützung, sodass der Nutzer auf dieser Basis komfortabel eine Ablaufsteuerung erstellen könnte. Neben dem UML-Profil für die Konfiguration müsste für dieses Konzept die Schnittstelle zwischen den UML-Profilen und der Sequenz-Generator Anwendung entwickelt werden.

Vergleichbar mit dieser Herangehensweise ist die in [15, Kapitel 2] vorgestellte XML-Notation zur Definition von *Event-Condition-Action* Regeln. XML kann ohne Werkzeugunterstützung textbasiert erstellt werden und zeichnet sich durch seine hierarchische Struktur aus. Bei steigender Komplexität wird eine solche Datei jedoch unübersichtlich, sodass ein grafisches

Frontend erforderlich wird. Ferner muss der Anwender, analog zur UML-Lösung, die Ereignisse als Regelsystem und nicht als konkrete Steuerungsbefehle angeben. Die Schnittstelle zur Sequenz-Generator-Anwendung müsste dieses in Steuerungsbefehle übersetzen, die an den Sequenz-Generator übertragen werden können.

In [16] basiert der Ansatz zur Synchronisierung des Messsystems dagegen nicht auf Ereignisregeln. Die Ereigniskette wird als gerichteter Graph definiert und kann mit entsprechender Werkzeugunterstützung generiert werden. Es können Abhängigkeiten zwischen Knoten erstellt werden, sodass zyklische Wiederholungen übersichtlich umsetzbar sind. Problematisch ist jedoch auch hier der Entwicklungsaufwand für die Übersetzung des Objektmodelles in konkrete Steuerbefehle.

Objektmodelle eigenen sich somit im Allgemeinen zur Abbildung von Ablaufsteuerungen. Sie strukturieren Ereignisse übersichtlich und können mit verfügbarer Werkzeugunterstützung einfach erstellt werden. Dennoch lässt sich die Problemdomäne mit bestehenden Objektmodellen nur allgemein abbilden. Ferner ist die Entwicklung eines Übersetzers notwendig, um aus einem Objektmodell die Ansteuerungsbefehle an einen Sequenz-Generator zu generieren.

2.2.3 Domänenspezifische Sprachen zur Definition von Sequenzen

Auf Basis eines Objektmodells kann eine domänenspezifische Sprache entwickelt werden, die spezifischere Operatoren sowie komplexere Hilfsstrukturen enthält. Eine Domain Specific Language (DSL) ist eine Programmiersprache, deren Operatoren und Werkzeuge für eine bestimmte Problemdomäne ausgelegt und damit sehr anwendungsspezifisch sind.

Im Allgemeinen wäre es auch denkbar, die Sequenzsteuerung in Form von Steuerbefehlen an die Softwarebibliothek in einer General Purpose Language (GPL) wie Python, Java, C# oder C++ zu programmieren. Der Entwicklungsaufwand hierfür wäre minimal, da kein Übersetzungsvorgang stattfinden muss. Eine solche Lösung ist jedoch nicht nutzerfreundlich, da der Nutzer über Kenntnisse von der Programmiersprache verfügen und ferner den Aufbau des Quelltextes kennen muss, um Änderungen an der Sequenz durchführen zu können. Wenn

ein solch anwendungsspezifisches Problem mit Hilfe einer DSL umgesetzt wird, ergeben sich nach [17, Abschnitt 1.1] folgende Vorteile:

- Erhöhung der Produktivität durch anwendungsspezifische Operatoren
- Verbesserung von Übersichtlichkeit, Codeverständnis und Reduzierung der Codekomplexität durch die Definition domänenspezifischer Konstrukte entsprechend einem geforderten Modell
- Erhöhung der Qualität durch Optimierungs- und Fehlerbehebungsmaßnahmen während der Übersetzung
- Gewährleistung von Portabilität

Um aus einem Skript auf Basis der entsprechenden DSL-Syntax die eigentliche Anwendung zur Lösung der Problemstellung zu generieren, wird ein zugehörigen Compiler benötigt, dessen Entwicklung neben der formalen Definition der Grammatik zur Umsetzung eines solchen Ansatzes gehört.

In [18, Kapitel 3] wird eine DSL erfolgreich genutzt, um Messequipment zu konfigurieren, Aufgaben zu synchronisieren und Tests durchzuführen. Hierbei wird ein modellgetriebener Ansatz verfolgt. Das bedeutet, dass mit Hilfe der DSL ein Modell definiert wird, aus dem anschließend ausführbarer Code generiert wird. Hier werden Vorteile bei der Fehlerkorrektur sowie eine effiziente Abstraktion der Funktionalität deutlich. Eine solche Lösung ist zur Ablaufsteuerung denkbar und würde die Softwarequalität verbessern. Die zusätzliche Abstraktion auf ein Modell erhöht jedoch den Entwicklungsaufwand, ohne die Nutzerfreundlichkeit signifikant zu steigern.

Um das Entwickeln einer DSL zu beschleunigen, können Frameworks wie *XText* verwendet werden, welche aus der Modelldefinition neben dem Parser auch eine Entwicklungsumgebung generieren können. In [19, Kapitel 4] wird eine DSL geschaffen, mit der die Hardware von Satelliten konfiguriert werden kann. Durch die Syntaxhervorhebung in der generierten Entwicklungsumgebung ist die Verwendung der DSL komfortabel und effizient umgesetzt. Da *XText* nur für Java zur Verfügung steht und die Vorgabe für die Compilerentwicklung die Verwendung von Python ist, kann ein solcher Ansatz im Rahmen dieser Arbeit nicht verfolgt werden. Ferner existieren keine vergleichbaren Frameworks zur Entwicklung einer Sprache in Python.

Das Potential der domänenspezifischen Vorabanalyse, welche in den Compiler einer DSL integriert werden kann, wird in [20, Kapitel 5] deutlich. Dort wird eine Sprache entwickelt, mit deren Hilfe Ereignisse auf Embedded-Hardware unter Echtzeit-Anforderungen ausgelöst werden können. Besonderes Augenmerk liegt während der Übersetzungsvorgangs auf der Einhaltung technischer Spezifikationen sowie der Vermeidung von Race-Conditions[2] oder Deadlocks[3]. Die Sprache ist jedoch nicht geeignet, um im Rahmen dieser Bachelorarbeit genutzt zu werden, da sie nur die Abstraktion der Ablaufsteuerung bereitstellt und der zugehörige Parser nur in Java verfügbar ist. Die Konfiguration des Sequenzers sowie das konkrete Generieren der Ablaufsteuerung müsste separat entwickelt, und der Java Parser eingebunden werden.

Ein weiteres Beispiel für eine umfangreiche Fehlererkennung ist die Entwicklung einer Sprache zur Konfiguration von Testequipment in [21, Kapitel 3]. Hierbei werden Programmierfehler inhaltlicher und logischer Art erkannt, sodass es für Testingenieure einfacher wird, valide Testabläufe zu definieren. Die Grammatik der Sprache ist mit Hilfe von regulären Ausdrücken definiert, auf deren Basis der Parser generiert werden kann. Ein generierter Parser zeichnet sich durch eine hohe Zuverlässigkeit sowie eine gute Anpassbarkeit aus. Im Rahmen der Bachelorarbeit sollte der Parser daher auch automatisiert auf Basis einer Grammatik generiert werden.

Die aufgezählten Arbeiten machen deutlich, dass eine DSL sehr gut zur Abbildung einer Ablaufsteuerung geeignet ist. Domänennahe Operatoren erhöhen die Übersichtlichkeit des Quelltextes und damit die Effizienz des Anwenders. In Anlehnung an bekannte Skriptsprachen ist sind sie ferner für den Anwender leicht zu erlernen. Die Grammatik der Sprache sollte formal definiert werden und mit einem generiertem Parser interpretierbar sein. Optimierungs- und Fehlererkennungsroutinen während des Übersetzungsvorgangs steigern die Qualität einer Problemlösung.

Das in Abschnitt 2.1.2 vorgestellte *Seco2004* enthält eine Konfigurationssprache die ebenfalls als DSL kategorisiert werden kann. Sie ermöglicht die Definition einer Ablaufsteuerung für Sequenz-Generatoren. Mit ihr können Schaltzeitpunkte und die entsprechende Pulsdauer für

[2] Konkurrierende Zugriffe, die zu Inkonsistenzen führen können
[3] Zirkuläre Abhängigkeiten, die den Programmfluss stoppen können

die Kanäle eines Sequenz-Generators angegeben werden. Der handgeschriebene Parser sowie das Fehlen einer Fehlererkennung schöpfen jedoch das Potential, welches die Entwicklung einer DSL bietet, nicht aus.

2.2.4 Fazit

In vergleichbaren Arbeiten werden Ablaufsteuerungen in proprietäre Software integriert, mit Hilfe von Objektmodellen abgebildet oder auf Basis domänenspezifischer Sprachen definiert.

Die Verwendung von proprietärer Software macht nur Sinn, wenn die gesamte Prozesskette eines einzigen Herstellers genutzt wird, was in der Abteilung *Exerimentelle Verfahren* jedoch nicht realisierbar ist. Ist der Nutzer in der Lage das Werkzeug zum Erstellen eines Objektmodells zu verwenden, kann er diese übersichtlich und effizient modellieren. Aus Sicht des Entwicklers können Konstrukte wie Schleifen nur schwierig umgesetzt werden. Ferner ist der Aufwand zur Implementierung des Übersetzers im Vergleich zum Parser einer DSL höher.

Als Fazit wird daher geschlossen, dass bestehende Arbeiten, in denen eine DSL entwickelt wird, die meisten Vorteile für die Umsetzung der Aufgabenstellung aufgezeigt haben. In Anlehnung an die bisherige *Sequenzdatei* ist eine einfache Anwendbarkeit sichergestellt. Im Übersetzungsvorgang können ferner Fehler erkannt und Optimierungen vorgenommen werden.

3 Theoretische Grundlagen

Im folgenden Kapitel werden die Grundlagen für die Umsetzung der Aufgabenstellungen zusammengetragen. In Abschnitt 3.1 werden formale Sprachen, Grammatiken und die Typisierung von Sprachen eingeführt. Zur Übersetzung einer formalen Sprache wird ein Compiler verwendet. Die Grundlagen des Compilerbaus sind in abschnitt 3.2 zusammengefasst. In Abschnitt 3.3 werden DSLs charakterisiert und ein Vorgehensmodell zur Entwicklung vorgestellt.

3.1 Formale Sprachen

Die Theorie der formalen Sprachen befasst sich mit der Syntax von Sprachen, die in Computersystemen genutzt werden, um Anwendungen zu beschreiben. Eine formale Sprache besteht aus einem Alphabet und Wörtern. Das Alphabet repräsentiert dabei eine endliche Menge an Zeichen. Die Syntax einer Sprache legt fest, welche Kombinationen der Zeichen, valide Ausdrücke der Sprache sind. Unter der Semantik einer Sprache versteht man die Bedeutung der einzelnen Ausdrücke.

3.1.1 Grammatiken

Die Syntax einer formalen Sprache kann mit Hilfe einer Grammatik definiert werden. Sie besteht aus einer Menge von nicht-terminalen Symbolen, terminalen Symbolen, Regeln und einem Startsymbol. Im folgenden werden diese Begriffe im Kontext der Sprachtheorie definiert:

- **Terminal:** Zeichen, welches Bestandteil des Alphabetes ist
- **Nicht-Terminal:** Zusammensetzung aus Terminalen und Nicht-Terminalen zur Repräsentation von Zwischenstrukturen
- **Regeln:** Beschreibung der Struktur eines validen Wortes
- **Startsymbol:** Ein Nicht-Terminal, welches den Beginn eines validen Ausdrucks bildet

Mit Hilfe der Grammatik kann die syntaktische Korrektheit einer beliebigen Eingabefolge validiert oder ein gültiger Satz der Sprache generiert werden [22, Kapitel 1]. Dies soll an folgendem Beispiel deutlich gemacht werden. Gegeben ist eine Grammatik zur Konfiguration der Interpretation von Trigger- und Arming-Signalflanken an den Eingängen eines Sequenz-Generators.

Terminale	=	{;, =, *Arming-Slope*, *Trigger-Slope*, *Up*, *Down* }
Nicht-Terminale	=	{Settings, Setting, Options, Parameter}
Regeln	=	{Options: *Arming-Slope* | *Trigger-Slope*
		Parameter: *Up* | *Down*
		Setting : Options = Parameter *;*
		Settings: Setting | Setting Settings}
Startsymbol	=	{ Settings }

Mit Hilfe der Grammatik können Einstellungen wie „Arming-Slope = Up; Trigger-Slope = Down;“ konstruiert werden.

3.1.2 Sprachhierachie

Grammatiken lassen sich in verschiedene Typen klassifizieren, zwischen denen eine hierarchische Beziehung existiert. Diese wird auch als Chomsky-Hierachie bezeichnet und unterscheidet vier Typen von Grammatiken. Aufbauend auf Typ 0, der **unbeschränkten Sprache**, für dessen Grammatik keine Einschränkungen gelten, basiert die Abstufung auf den Eigenschaften der Ersetzungsregeln:

- Typ 1 - **kontextsensitive Sprache:** Ein Nicht-Terminal umgeben von Terminalen und Nicht-Terminalen wird durch eine beliebige Kombination von Terminalen und Nicht-Terminalen ersetzt
- Typ 2 - **kontextfreie Sprache:** Genau ein Nicht-Terminal wird durch eine Folge von Terminalen und Nicht-Terminalen ersetzt
- Typ 3- **reguläre Sprache:** Genau ein Nicht-Terminal wird durch ein Nicht-Terminal gefolgt von einem Terminal (linksregulär) oder einem Terminal gefolgt von einem Nicht-Terminal (rechtsregulär) ersetzt

3.1.3 Darstellung einer Sprache in EBNF

Um das Regelsystem kontextfreier Grammatiken für Maschinen besser lesbar darzustellen, existiert die BNF. In ihr werden Nicht-Terminale in eckigen Klammern und Terminale in Anführungszeichen angegeben. Mit dem „|“-Operator lassen sich Alternativen angeben. Um Wiederholungen und Optionalität nicht durch Rekursion oder Alternativen ausdrücken zu müssen, existiert die Erweiterte Backus-Naur-Form (EBNF), die in der Praxis häufig zur

Definition von kontextfreien Grammatiken verwendet wird. Die entsprechende Notation ist in Tabelle 2 angegeben.

Notation	Bedeutung
<Zeichenkette>	Nicht-Terminal
"*valide(s) Zeichen*"	Terminal
+	Und Verknüpfung
|	Oder Verknüpfung
(*Ausdruck*)?	Optionalität des Ausdrucks
(*Ausdruck*)+	Einmaliges oder beliebig häufiges Vorkommen
(*Ausdruck*)*	Optionales oder beliebig häufiges Vorkommen

Tabelle 2: Regeln zur Definition einer Grammatik in EBNF-Notation

3.2 Compilerbau

Ein Compiler (dt. Übersetzer) ist ein Programm, dass eine Anwendung von einer Quellsprache in eine identische Anwendung einer Zielsprache übersetzt [23, Kapitel 1, S. 1]. Die Syntax und die Semantik einer Sprache kann in Form einer formalen Grammatik definiert werden. In den Übersetzungsprozess können ferner Optimierungen und kontextabhängige Fehlerbehebungen integriert werden. [24, Kapitel 1, S. 1]

3.2.1 Übersetzungsschritte

Der Übersetzungsvorgang findet in mehreren Phasen statt, die als Ergebnis unterschiedliche Repräsentationen des Quellprogramms haben. Man spricht von einem klassischen Compiler, wenn Quelltext einer GPL in Maschinencode übersetzt wird. In Abbildung 9 sind die Phasen sowie die jeweilige Repräsentation des Quellprogrammes während des Übersetzungsvorgangs dargestellt. Die verschiedenen Phasen werden in den folgenden Abschnitten charakterisiert.

Lexikalische Analyse

Bei der lexikalischen Analyse wird die Eingangszeichenfolge auf Schlüsselworte durchsucht. Zusammenhängende Zeichenfolgen werden in sogenannten Tokens zusammengefasst. Schlüssel-

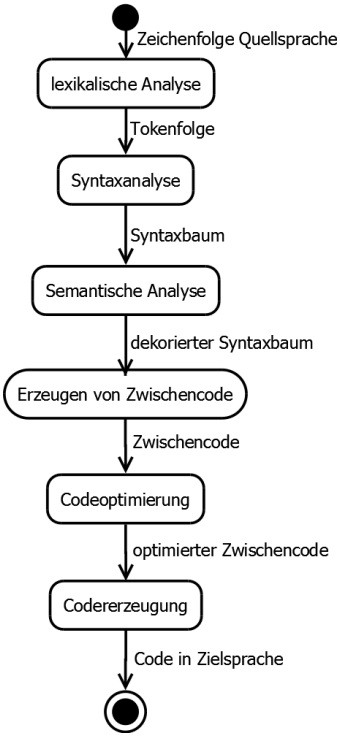


Abbildung 9: Phasen eines Übersetzungsvorgangs [24, Abschnitt 1.2, S. 3]

worte und andere Bestandteile der Sprache sind als reguläre Ausdrücke[1] oder als Grammatik des Chomsky-Typ 3 definiert und können auch als endlicher Automat aufgefasst werden. Jedes eingelesene Zeichen erwirkt einen Zustandsübergang. Das Erreichen eines bestimmten Zustandes bedeutet, dass ein Schlüsselwort erkannt ist. Invalide Zustandsübergänge zeigen Syntaxfehler auf. [24, Kapitel 2, S.19-40]

Ein auf diese Weise erkannter Token besteht immer aus dem Bezeichner der erkannten Zeichenfolge sowie der erkannten Zeichenfolge. Aus folgender Zeichenfolge wird die in Quelltext 2 gezeigte Tokenfolge erzeugt:

```
sequence{17, 100, f;}
```

[1] Notation zur Definition von Wörtern

```
1 [(<sequenceKeyword>, "sequence"),
2 (<Literal>, "{"),
3 (<canalnumber>, "17"),
4 (<Literal>, ","),
5 (<time>, "100"),
6 (<Literal>, ","),
7 (<duration>, "f")
8 (<Literal>, ";"),
9 (<Literal>, "}")]
```

Quellcode 2: Tokenfolge als Ergebnis der lexikalischen Analyse einer Zeichenfolge

Syntaxanalyse

Während der Syntaxanalyse werden die Tokens auf Basis einer kontextfreien Grammatik (Chomsky-Typ 2) hierarchisch systematisiert. In Quelltext 3 ist eine Grammatik definiert, auf deren Grundlage die Syntaxanalyse des obigen Beispiels vorgenommen wird.

```
1 Quellcode = Sequenz
2 sequenceKeyword = "sequence"
3 Sequenz = sequenceKeyword + "{" + (Event)+ + "}"
4 Event = canalnumber  + "," + time + "," + duration + ";"
```

Quellcode 3: Beispielgrammatik für eine Sequenz

Eine übliche Darstellungsweise des Ergebnisses ist der Abstract Syntax Tree (AST)[2], dessen Knoten einen Operator oder eine Zusammenfassung repräsentieren. Beim AST handelt es sich um einen homogenen Baum, dessen Blätter jeweils nur ein Element enthalten. Der Syntaxbaum zur Tokenfolge aus Quelltext 2, erweitert um ein weiteres Ereignis, ist in Abbildung 10 dargestellt. [23, Abschnitt 4.1, S. 192 - 196]

Semantische Analyse

Bei der semantischen Analyse wird geprüft, ob der Programmablauf gültig ist und es werden Optimierungen auf semantischer Ebene vorgenommen. Es werden zum Beispiel Konstanten ersetzt und überprüft, ob verwendete Variablen definiert worden sind. Optimierungen können das Zusammenfassen von Operationen oder das Entfernen von „totem Code“ sein.

[2] Abstrakter Syntaxbaum

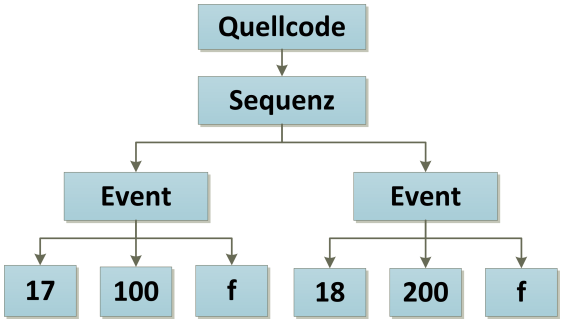


Abbildung 10: Syntaxbaum einer Sequenz

Der dekorierte Syntaxbaum ist ein inhomogener Baum, dessen Blätter neben dem eigentlichen Element weitere Informationen enthalten. Unter der Annahme, dass im Quelltext die Konstante f definiert ist und die beiden Ereignisse zum Zeitpunkt 100 stattfinden, ist in Abbildung 11 der dekorierte Syntaxbaum zum obigen Beispiel dargestellt. Die Blätter werden bei der semantischen Analyse dieses Beispiels um die entsprechenden Datentypen erweitert und es findet eine Zusammenlegung gleicher Events statt.

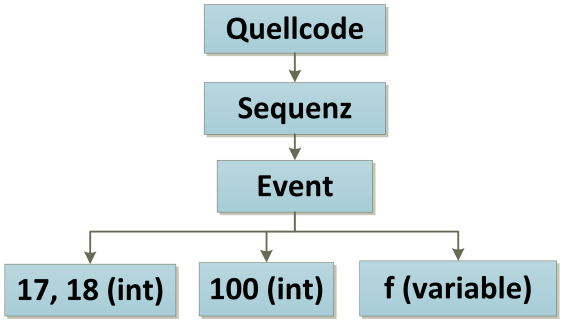


Abbildung 11: Dekorierter Syntaxbaum als Ergebnis der semantischen Analyse eines Syntaxbaumes

Erzeugung von Zwischencode

Auf Basis des dekorierten Syntaxbaumes wird der Zwischencode erzeugt. Er ist eine abstrakte Repräsentation des Quellprogrammes in einer Form, die einfach in die Zielsprache übertragen werden kann. Dies ist zum Beispiel notwendig, wenn mehrere verwandte Zielplattformen unterstützt werden sollen. Der Zwischencode liegt anschließend in einer generischen Form,

welche maschinennah aber allgemeingültig ist, vor.

Beim verwendeten Beispiel wird der dekorierten Syntaxbaum auf eine Form abstrahiert, welche die Ereignisse hardwareunabhängig repräsentiert. Quellcode 4 zeigt einen möglichen Zwischencode, bei dem die Ereignisse chronologisch sortiert mit dem Status aller Kanäle für f = 250 angegeben sind.

```
1  time  canal-high
2  100: [17 = 1, 18 = 0]
3  200: [17 = 1, 18 = 1]
4  250: [17 = 0, 18 = 1]
5  350: [17 = 0, 18 = 0]
```

Quellcode 4: Zwischencode einer Eventsequenz

Codeoptimierung

Die Optimierung des generierten Zwischencodes ist domänenspezifisch und kann sehr komplex ausfallen. Neben der allgemeinen Optimierung im Rahmen der semantischen Analyse werden im Übersetzungsschritt der Codeoptimierung plattformspezische Optimierungen vorgenommen. Grundlegende Motivation ist es, den Zwischencode für die Architektur der Zielsprache zu optimieren, um eine Effizienzsteigerung des Programms zu bewirken.

Bei der Ablaufsteuerung werden zum Beispiel die Zeitangaben auf einen gemeinsamen Takt umgerechnet, was auf dem Sequenz-Generator längere Sequenzen ermöglicht. Der auf diese Weise optimierte Zwischencode ist in Quelltext 5 abgebildet.

```
1  clock     canal-high
2  2:        [17 = 1, 18 = 0]
3  4:        [17 = 1, 18 = 1]
4  5:        [17 = 0, 18 = 1]
5  7:        [17 = 0, 18 = 0]
```

Quellcode 5: Optimierter Zwischencode einer Eventsequenz

Codeerzeugung

Der letzte Arbeitsschritt eines Compilers ist das Erzeugen des Programmes in der Zielsprache. Der Zwischencode wird hierzu auf die Operatoren der Zielsprache abgebildet. Ferner müssen sprachspezifische Besonderheiten, wie zum Beispiel notwendige Initialisierungen, Adressumwandlungen oder die korrekte Befehlsreihenfolge beachtet werden.

Das gewählte Beispiel wird in diesem Schritt in Steuerungsbefehle übersetzt, welche an den Sequenz-Generator übertragen werden. Da der Programmablauf abstrakt in Zwischencode vorliegt, kann die Codeerzeugung einfach für verschiedene Zielsprachen umgesetzt werden.

3.2.2 Werkzeugunterstützung

Für die Entwicklung von Compilern existieren viele Werkzeuge, die die Entwicklungszeit verkürzen und die Qualität des entwickelten Compilers steigern. Die meisten Werkzeuge sind auf die Implementierung eines Übersetzerschrittes spezialisiert. Es gibt jedoch auch Werkzeuge, die die Umsetzung aller Entwicklungsschritte unterstützen.

Das Vorgehen bei den Schritten lexikalische Analyse, Syntaxanalyse und semantische Analyse ist für viele Problemstellungen gleich, sodass sich für diese Schritte viele Werkzeuge existieren, die im Allgemeinen für den Bau eines beliebigen Compilers genutzt werden können. Die darauf folgenden Schritte der Zwischencodeerzeugung, Optimierung und der Transformation in die Zielsprache dagegen sind abhängig von der konkreten Problemstellung und erfordern individuelle Lösungen, die durch allgemeine Werkzeuge nicht optimal abgedeckt sind.

Für die Entwicklung von Compilern bzw. den Teilsystemen eines Compilers, existieren für viele GPLs Werkzeuge. Im Kontext der DSL Entwicklung können folgende Werkzeuge als etabliert bezeichnet werden. [25]

Java:

- *Another Tool for Language Recognition (ANTLR)*: Zur Erstellung von Parser und Lexer; unterstützt auch Python und C# als Zielsprache [26]

- *XTEXT*: Framework für Parser und Lexer welches auch Werkzeuge zur Interpreter-, Linker- und Compiler-Entwicklung enthält [27]
- *JavaCC*: Zur Erstellung von Parser und Lexer; unterstützt auch C/C++ als Zielsprache [28]

C / C++:

- *Lexical Analyzer (LEX) / Yet Another Compiler Compiler (YACC)*: Zur Generierung von Parser und Lexer in C (ursprünglich kommerziell) [29] [30]
- *FLEX/Bison*: Open-Source Implementierung von LEX/YACC zur Generierung von Parser und Lexer [31] [32]

Python:

- *PLY*: LEX / YACC Implementation in Python zur Generierung von Parser und Lexer [33]
- *PyParsing*: Bibliothek zur Definition und Ausführung von Parser und Lexer [34]

In Abschnitt 4.4.1 wird die Entscheidung getroffen, welche Werkzeuge im Rahmen dieser Bachelorarbeit verwendet werden.

3.3 Domänenspezifische Sprachen

Eine domänenspezifische Sprache ist eine Programmiersprache, die auf einen bestimmten Problembereich zugeschnitten ist. Ihre Operatoren zeichnen sich durch die Nähe zur Problemdomäne aus und ermöglichen deshalb eine effiziente Problemlösung.

Sie differenzieren sich damit zu den GPLs, welche mit allgemeinen Operatoren für eine Vielzahl von Problemlösungen verwendet werden können. GPLs sind ferner Turing-vollständig und können damit zur universellen Programmierung genutzt werden. Eine Programmiersprache wird als Turing-vollständig bezeichnet, wenn sie zur Abbildung aller mathematischen Probleme geeignet ist, die auch mit einer Turingmaschine[3] abgebildet werden können. Zu den GPLs gehören unter anderem Programmiersprachen wie Java, C oder Python.

[3] 1936 von Alan Touring entwickeltes, mathematisch analysierbares Rechnermodell zur Lösung universeller mathematischer Probleme

In der Praxis haben sich folgende DSLs etabliert:

- **SQL**: Abfragesprache für Datenbanken
- **LaTeX**: Textsatzsystem; vorrangig zur Erstellung von wissenschaftlichen Arbeiten
- **Matlab**: Skriptsprache für technische Berechnungen (an der Grenze zur GPL)
- **HTML**: Sprache zur Strukturierung von Webdokumenten
- **make**: Programmierwerkzeug zur Automatisierung von Build-Prozessen

In Absatz 2.2.3 wird deutlich, dass DSLs auch für kleinere Problemdomänen, wie die Konfiguration von Embedded-Hardware, entwickelt und genutzt werden können.

3.3.1 Einsatzgebiete

Die Vorstufe einer DSL ist eine API, die eine Problemdomäne abbildet und über einer GPL angesprochen werden kann. Die Entwicklung ist einfach und die erhöhte Abstraktion steigert die Übersichtlichkeit bei der Programmierung. Die Entwicklung einer DSL ist jedoch, in Hinblick auf das Optimierungspotential sowie dem Abstraktionsniveau, der konsequentere Weg eine Problemdomäne abzubilden.

Nach [17, Abschnitt 2.2] werden DSLs im Allgemeinem zur Umsetzung der folgenden Funktionalität genutzt:

- Analyse, Verifikation, Parallelisierung und Transformierung von GPL Programmen
- Abstraktion von Anwendungsbibliotheken
- Automatisierung von Aufgaben
- Codegenerierung
- Repräsentation von Datenstrukturen
- Konfiguration und Anpassung von Hardware- oder Softwaresystemen
- Konstruktion von GUIs
- Vornehmen von Programmeingaben

Die Entwicklung einer Ablaufsteuerung mit einem Sequenz-Generator kann als *Abstraktion einer Anwendungsbibliothek* oder *Konfiguration und Anpassung eines Hardwaresystems* kategorisiert werden.

3.3.2 Entwicklungsprozess

Nach [17, Abschnitt 2.1] wird der Entwurfsprozess in die Phasen Entscheidung, Analyse, Design, Implementation und Bereitstellung unterteilt. Diese Aufteilung orientiert sich am etablierte Wasserfallmodell des Software-Engineering und kann somit als Vorgehensmodell für die Entwicklung einer DSL verwendet werden. Die Phasen Design und Implementation können iterativ durchlaufen werden.

Entscheidung

Grundsätzlich sollte die Entscheidung für eine DSL aus den zu erwartenden Vorteilen hinsichtlich der Nutzerergonomie und der damit verbunden Produktivitätssteigerung getroffen werden. In Abschnitt 3.3.1 sind verschiedene Problemdomänen benannt, in denen dies im Allgemeinen erreicht wird. Fällt eine Problemstellung in diese Domänen, sollte die Entscheidung für die Entwicklung einer DSL getroffen werden. [17, Abschnitt 2.2]

Analyse

In der Analysephase wird die Problemdomäne abgegrenzt und auf fachspezifische Operatoren abstrahiert. Außerdem werden Anforderungen an die Funktionalität gestellt und Anwendungsfälle, in denen die DSL eingesetzt werden soll, definiert. Teil der Analyse ist ebenfalls der Kentnissstand der Nutzer im Bereich der Problemdomäne sowie der Programmierung mit GPLs im Allgemeinen. Ergebnis der Analyse sollte ein grobes Konzept über den Sprachumfang und konkrete Schlüsselworte sein. [17, Abschnitt 2.3]

Design

In der Designphase wird die Syntax und die Semantik der Sprache, auf Basis der getätigten Analyse, definiert. Eine einfache Form der Umsetzung einer DSL ist die Erweiterung oder die Erneuerung einer existierenden Sprache oder die Anlehnung an eine Notation [17, Abschnitt 2.4]. Nach [35, Lesson 3] sollte beim konkreten Design der Sprache beachtet werden, dass man Operatoren und Funktionen nicht zu allgemein definiert, aber gleichzeitig vermeidet zu viele Spezialfälle zu modellieren.

Implementation

Die DSL dient zur Beschreibung einer Problemdomäne. Auf Basis des Designs muss ein Übersetzer implementiert werden, der aus der abstrakten Beschreibung konkrete Operationen generiert. Im Allgemeinen verwendet man hierzu einen Compiler dessen Aufbau in Abschnitt 3.2 im Detail beschrieben ist. Um Fehlerquellen zu minimieren und eine Erweiterung und Wartbarkeit der DSL zur ermöglichen, ist die Verwendung von Werkzeugen die empfohlene Strategie, einen Compiler zu entwickeln. Zur Implementation gehört ferner das Testen des entwickelten Compilers.

4 Konzept

Auf Basis der bisherigen Erkenntnisse wird im Rahmen dieser Bachelorarbeit eine domänenspezifische Sprache entwickelt, die an die Konfigurationssprache von *Seco2004* angelehnt ist. Der Parser wird mit Hilfe eines Parsergenerators in Python umgesetzt. Die Zielplattform des Compilers ist ein Hardsoft Sequenz-Generator v5.1 für dessen Ansteuerung eine Softwarebibliothek zu Verfügung steht.

Im folgenden Kapitel werden verschiedene Konzepte zur Umsetzung der Aufgabenstellungen gegenübergestellt. Auf dieser Basis wird entschieden, welche konkreten Konzepte für die Umsetzung von *Seco2015* zur Anwendung kommen.

Hierzu wird in Abschnitt 4.1 ein Vorgehensmodell definiert, welches die Zerlegung der Problemlösung in Teilsysteme vorsieht. Diese wird in Abschnitt 4.2 vorgenommen und bildet die Grundlage des Konzeptes.

In Abschnitt 4.3 wird der Sprachumfang der zu entwickelnden Sprache abgegrenzt und ein konkretes Konzept für die neue Syntax aufgestellt. In Abschnitt 4.4 wird der Parser konzeptioniert, der die *Sequenzdatei* einliest. In Abschnitt 4.5 wird ein Konzept für eine Laufzeitumgebung erstellt, welche die eingelesenen Daten verwaltet und auswertet.

In Abschnitt 4.6 wird ferner die Optimierung der generierten Ereignisfolge und die Transformation in die Zielsprache konzeptioniert. Die Hardwareansteuerung wird in Abschnitt 4.7 entworfen.

Eine Zusammenfassung des entwickelten Konzeptes wird in Abschnitt 4.8 gegeben.

4.1 Vorgehensmodell

Der Prozess des Software-Engineering ist die „[...] zielorientierte Bereitstellung und systematische Verwendung von Prinzipien, Methoden und Werkzeugen für die arbeitsteilige, ingenieurmäßige Entwicklung und Anwendung von umfangreichen Software-Systemen.“ [36] In der Praxis sollte dieser Prozess das iterative Durchlaufen folgender Arbeitsschritte umfassen:

1. Analysieren der Problemdomäne
2. Entwerfen von Lösungsstrategien
3. Entwickeln der konkreten Problemlösung
4. Bereitstellen der fertigen Lösung
5. Betreiben und Warten der Anwendung

Die in Abschnitt 1.3 definierten Anforderungen basieren auf der Analyse der Problemdomäne, für die im Folgenden eine geeignete Lösungsstrategie entworfen werden muss. Nach [36, Kapitel I] bedeutet dies, eine softwaretechnische Lösung in Form einer Architektur zu entwickeln.

Ein geeigneter Ansatz hierfür ist es, zunächst einen Grobentwurf aufzustellen, der die Problemlösung in Teilsysteme aufteilt. Kern dieses Entwurfes ist eine funktionale Beschreibung der Teilsysteme, sowie die Definition von Schnittstellen.

Anschließend werden für die Teilsysteme unterschiedliche Feinentwürfe entwickelt, welche im Kontext der Anforderungen evaluiert werden. Auf dieser Basis entsteht schließlich eine Gesamtarchitektur, die die Grundlage für den nächsten Arbeitsschritt, das Entwickeln der konkreten Problemlösung, bildet.

Der folgende Arbeitsschritt der Bereitstellung einer fertigen Lösung wird in Kapitel 5 beschrieben.

4.2 Aufteilung der Problemlösung in Teilsysteme

Der Abgrenzung der Teilsysteme liegt die in Abschnitt 1.3 durchgeführte Anforderungsanalyse und der in Abschnitt 3.2 vorgestellte Ablauf des Übersetzungsvorgangs zu Grunde. Im Folgenden werden die Teilsysteme von *Seco2015*, sowie deren Schnittstellen untereinander definiert.

Die **Grammatik der Sprache** wird als erstes Teilproblem definiert, welches den Entwurf der DSL hinsichtlich der Funktionalität sowie die konkrete Syntax umfasst. Die Grammatik repräsentiert eine abstrakte Darstellung der neuen Sprache, welche als Grundlage für den Parser genutzt wird.

Der **Parser** setzt die lexikalische und syntaktische Analyse aus dem Compilerbau um. Er stellt der Laufzeitumgebung den Syntaxbaum in einer zu entwerfenden Form zur Verfügung.

Die **Laufzeitumgebung** verwaltet die geparsten Informationen und stellt Schnittstellen zum Aktivieren von Laufzeitkonfigurationen bereit. Sie ist als Teilsystem für das Generieren der Ereignisfolge in einer generischen Form verantwortlich. Im Compilerbau entspricht dies dem Erzeugen von Zwischencode.

Der **Optimierer** führt den Schritt der Codeoptimierung aus dem Compilerbau aus. Er nimmt die Rohdaten der Ereignisfolge entgegen und verbessert sie im Rahmen der technischen Eigenschaften der Zielplattform[1].

Das Teilsystem **Übersetzer** transformiert die optimierte, generische Ereignisfolge in das Format, welches die entsprechende Zielplattform erwartet. Dies entspricht im Compilerbau dem Schritt der Codeerzeugung.

Die **Hardwareansteuerung** umfasst die Schnittstelle zwischen Compiler und Sequenz-Generator. Sie initialisiert die Hardware und überträgt den generierten Code der Ereignisfolge.

Die Koordination der Teilsysteme wird in der **Hauptanwendung** umgesetzt. Sie initiiert die einzelnen Übersetzungsschritte und baut die Verbindung zu einem angeschlossenen Sequenz-Generator auf. Ferner bietet sie die Möglichkeit, eine übertragene Sequenz zu starten sowie die Ausführung zu unterbrechen.

In den folgenden Abschnitten werden Konzepte für jedes Teilsystem entwickelt und die notwendigen Schnittstellen definiert.

4.3 Grammatik der Sprache

In Abschnitt 2.2.3 wird, basierend auf vergleichbaren Arbeiten, das Fazit geschlossen, dass eine DSL optimal für die Abbildung einer Ablaufsteuerung geeignet ist. Nach dem in Abschnitt

[1] entspricht hier der Hardwarerevision des Sequenz-Generators

3.3.2 vorgestellten Vorgehensmodell wird im Folgenden die Analyse der mit der Sprache abzudeckenden Problemdomäne durchgeführt.

4.3.1 Analyse der Problemdomäne

Die Analyse der Problemdomäne beinhaltet sowohl die Analyse der abzudeckenden Funktionalität, als auch die Analyse der Nutzerbasis.

Wie in Abschnitt 1.2.2 beschrieben, umfasst die konkrete Problemdomäne der zu entwickelnden Sprache die zeitkritische Synchronisierung von Messequipment zur Durchführung von Messkampagnen. Das Timing der Komponenten muss von Wissenschaftlern entsprechend der technischen Rahmenbedingungen, sowie den konkreten Messanforderungen definiert werden. Die Kernaufgabe der folgenden Analyse ist es, einen geeigneten Abstraktionsgrad zur Abbildung der Ablaufsteuerung zu finden, die als Grundlage für das Sprachdesign verwendet werden kann.

Aus technischer Sicht wird die Synchronisierung durch Schaltzeitpunkt und Impulsdauer der Impulse auf den einzelnen Kanälen repräsentiert. Beim Hardsoft Sequenz-Generator ist es möglich, die Synchronisierung auf Basis dieser technischen Sicht über die Hardware-Bedienelemente oder mit der zugehörigen Softwarebibliothek zu modellieren. Dieser Ansatz ist generisch und nicht an die konkrete Problemdomäne der Abteilung angepasst. Anhand der optischen Messtechnik PIV werden im Folgenden Anforderungen an den Funktionsumfang der neuen DSL gestellt. Hierzu werden zunächst Anwendungsfälle aus Sicht der Experimentatoren dargestellt.

Beim Erstellen des Timings werden die einzelnen Messkomponenten von den Experimentatoren nie autark, sondern immer in Abhängigkeit voneinander betrachtet. Um zum Beispiel die korrekte Lichtintensität zu erreichen, muss der Abstand und die Impulsdauer von Lasern und Güteschaltern berücksichtigt werden.

Im Rahmen von Messkampagnen werden verschiedene Aufnahmeserien mit unterschiedlichen Messparametern durchgeführt. Das Grundgerüst des Timings bleibt hierbei gleich, jedoch erfordert die Anpassung von Messparametern eine Anpassung der einzelnen Timings. Beispielsweise erfordert eine Erhöhung der Strömungsgeschwindigkeit eine Verringerung des Abstandes

zwischen den Laserimpulsen. Damit wird sichergestellt, dass die Korrelation der Streupartikel möglich bleibt.

Ein weiterer Anwendungsfall ist das Durchführen von Kalibrierungen. Hier wird aus Sicherheitsgründen nur eine geringe Laserintensität verwenden. Die Anpassung der Laserintensität ist über das Timing der Güteschalter möglich. Ferner werden bei der Kalibrierung beispielsweise die Kanäle für die Kameras deaktiviert. Damit für derartige Messmodi nicht manuell in die Sequenzfolge eingegriffen werden muss, sollten Voreinstellungen möglich sein.

In der Praxis kommt es vor, dass eine Messung nicht vom Entwickler der *Sequenzdatei* durchführt wird. Damit für die Ausführung der *Sequenzdatei* keine Fachkenntnisse erforderlich sind, ist die Abstraktion auf verschiedener Messmodi ebenfalls sinnvoll. Die *Sequenzdatei* kann auf Basis der Voreinstellungen auch ohne Fachkenntnisse ausgeführt werden.

Zusammengefasst können auf Basis der bisherigen Analyse folgende Anforderungen an eine abstrakte Modellierung der Ablaufsteuerung gestellt werden:

- Definition von Abhängigkeiten zwischen Komponenten
- Modellierung von Abhängigkeiten zwischen Messparametern und Komponenten
- Festlegen und Speichern von unterschiedlichen Messkonfigurationen
- Temporäre Deaktivierung von Kanälen

Das Sequenz-Generierungswerkzeug *Seco2004* abstrahiert die Ablaufsteuerung auf eine der Problemdomäne angepasste Ebene. Die Sprachsyntax zur Definition der Echtzeitsynchronisierung unterstützt Mechanismen, um die einzelnen Anforderungen zu erfüllen. In Anhang B ist die Sprachsyntax von *Seco2004* in EBNF-Notation definiert.

Eine *Sequenzdatei* wird hierzu, wie in Abschnitt 2.1.2 erläutert, in verschiedene Bereiche untergliedert. Die konkrete Sequenzdefinition findet im Abschnitt *sequences* statt. Unter Angabe der Kanalnummer, des absoluten Startzeitpunktes und der Impulsdauer kann ein Ereignis definiert werden. Abhängigkeiten, wie ein Phasenversatz zu anderen Komponenten oder ein linearer Zusammenhang zu Messparametern, können in mathematischen Formeln und global definierten Variablen abgebildet werden. Die Variablen werden im Bereich *variables*

definiert und können zur Laufzeit angepasst werden.

Um die an die Kanäle angeschlossenen Geräte identifizieren zu können, wird im Bereich *channels* eine entsprechende Zuordnung hergestellt. Diese können über die Voreinstellungen temporär deaktiviert werden.

Die Anforderung, konkrete Messprofile definieren zu können, wird durch den Bereich *menu* erfüllt. Hier werden verschiedene Voreinstellungen definiert, in denen die Standardwerte zuvor definierter Variablen überschrieben werden können. Werden die Variablen im Bereich der Sequenzdefinition genutzt, um Abhängigkeiten zu parametrisieren, ist es möglich, verschiedene Messprofile in den Menüeinträgen abzubilden.

Die Sprachsyntax von *Seco2004* bildet die Problemdomäne entsprechend der Grundanforderungen der Experimentatoren ab und wird daher als Grundlage für die Sprachsyntax von *Seco2015* verwendet. Die weitere Analyse der Problemdomäne offenbart jedoch weiteres Potential, die Ablaufsteuerung zu abstrahieren.

Zwischen einzelnen Messsequenzen müssen die Laser auf Betriebstemperatur gehalten werden. Hierzu müssen sie in regelmäßigen Abständen ausgelöst werden. Aktuell müssen für den Messbetrieb und derartige Aufgaben jeweils ein Sequenz-Generator verwendet werden. In einer Sequenz ist die Ablaufsteuerung für das Erstellen einer Aufnahme definiert. Dieses wird in einer bestimmten Frequenz wiederholt. Über einen Multiplexer wird bei der Pausierung einer Messung sichergestellt, dass die Laser über einen anderen Sequenz-Generator angesteuert werden. Dieser führt die Sequenz zum Halten der Betriebstemperatur aus.

Diese Aufgabe könnte bei der Unterstützung von Schleifen auch mit einem Sequenz-Generator im Alternative-Modus (siehe Abbildung 5 auf Seite 10) umgesetzt werden. Die Hauptsequenz enthält die Erwärmung des Lasers. In der Nebensequenz kann in einer Schleife die Sequenzfolge zur Bilderfassung definiert werden. Die Anzahl der Iterationen gibt die Anzahl der aufzunehmenden Bilder an. Wenn keine Messung stattfindet, führt der Sequenz-Generator automatisch die Hauptsequenz aus. Sobald ein Arming-Signal ausgelöst wird, wird die Nebensequenz ausgeführt, sodass die gesamte Messausrüstung für die Anzahl der Aufnahmen getriggert wird.

Bisher können nicht alle Einstellungen des Sequenz-Generators auf Ebene der *Sequenzdatei*, sondern nur in der Benutzeroberfläche von *Seco2004* vorgenommen werden. Die Einführung eines Bereiches *settings* mit der entsprechenden Möglichkeit, den Sequenz-Generator in Abhängigkeit zur aktuellen Messkampagne zu konfigurieren, würde die gesamte Konfiguration in der Sequenzdatei möglich machen. Die Anpassung der Konfiguration zur Laufzeit wäre nicht mehr erforderlich.

Um die Analyse der Problemdomäne abzuschließen, muss die Anwenderbasis von *Seco2015*, bestehend aus Wissenschaftlern und Experimentatoren, näher betrachtet werden. Grundlegend sind bei den Anwendern Erfahrungen mit Skriptsprachen wie *Matlab* oder *Python* vorhanden. Außerdem sind sie vertraut mit der bisherigen Konfigurationssprache sowie deren Einschränkungen. Um sicherzustellen, dass die Nutzung der neuen Sprache für die Anwender intuitiv ist, wird die Syntax der alten Sprache übernommen und im Detail verbessert. Die geplanten Erweiterungen des Funktionsumfangs werden syntaktisch an bekannte Skriptsprachen angelehnt.

Zusammen mit den Anwendern wird auf Basis dieser Anforderungen das in Anhang A dokumentierte Beispiel für eine Sequenzdatei von *Seco2015* entworfen. Auf dieser Grundlage wird die konkrete Sprachsyntax entwickelt.

4.3.2 Design der Sprachsyntax

Die *Sequenzdatei* besteht aus den in Abschnitt 2.1.2 beschriebenen Bereichen, die unterschiedliche Funktionen erfüllen und dem Skript eine Grundstruktur geben. Zur Definition von Parametern werden die Abschnitte *variables*, *channels* und *menu* genutzt, welche um die Abschnitte *settings* und *macros* erweitert werden. Die konkrete Ablaufsteuerung wird weiterhin im Abschnitt *sequences* vorgenommen.

Um eine konsistente Syntax innerhalb der *Sequenzdatei* sicherzustellen, werden folgende allgemeingültige Regeln aufgestellt:

- Zahlen können immer als mathematischer Ausdruck angegeben werden

- Mathematische Ausdrücke können Variablen, Gleitkommazahlen, Additionen, Subtraktion, Multiplikation, Division, Potenzierung, sowie die Schachtelung beliebiger Ausdrücke enthalten
- Parameter von Anweisungen werden, wie zum Beispiel in Java und C, mit Kommata getrennt
- Kommentare werden mit dem aus Python bekannten Symbol # eingeleitet
- Anweisungen werden, wie zum Beispiel in Java und C, mit einem Semikolon abgeschlossen
- Bereiche und Definitionen werden, mit aus Java und C bekannten, geschweiften Klammern begonnen und beendet
- Freizeichen zwischen einzelnen Terminalen sowie Einrückungen werden ignoriert

Einige Nicht-Terminale kommen darüber hinaus ebenfalls in mehreren Bereichen der *Sequenzdatei* vor. In Anhang C.1 ist die Grammatik der im folgenden beschriebenen globalen Definitionen dokumentiert:

- Identifier, als kurze Beschreibung von Variablen, Kanälen, Einstellungen oder Menüeinträgen; Kann zur Anzeige in einer GUI verwendet werden
- Variablenname, als Bezeichnung von Variablen, Aliasen sowie Markos
- Die Datentypen Float und String

Die globale Definition wird um die formale Definition von mathematischen Ausdrücken ergänzt. Hierfür ist in Anhang C.2 eine rekursive Grammatik definiert, welche beliebig tief geschachtelte, mathematische Ausdrücke ermöglicht.

In den folgenden Abschnitten werden konkrete Entwürfe für die Sprachsyntax der verschiedenen Bereiche entwickelt. Sie verwenden die, in diesem Abschnitt vorgestellten, globalen Definitionen.

4.3.3 Globale Definitionen

Im Folgenden wird die konkrete Sprachsyntax für die globalen Definitionen innerhalb einer *Sequenzdatei* entworfen. Dies umfasst die Bereiche *variables*, *settings*, *channels* und *menu*.

Bereich variables

Der Bereich *variables* dient zur Definition von Variablen und erhält im Vergleich zu *Seco2004* keine neue Funktionalität, sodass er aus der alten Sprachdefinition übernommen wird. Eine Anweisung besteht aus folgenden Operatoren:

```
Name, Beschreibung, Gleitkommazahl, Gleitkommazahl, Gleitkommazahl;
```

Die drei Zahlenwerte stehen für den Wertebereich sowie den Standardwert der Variable. Da der Momentanwert immer der Median der angegebenen Werte ist, spielt die Reihenfolge der Zahlen für die semantische Analyse keine Rolle. Die Übernahme des Wertebereiches ist zur Erfüllung der Aufgabe nicht notwendig. In Zukunft könnte er jedoch genutzt werden, um in einer GUI die Skalierung von Schiebereglern zu ermöglichen bzw. die Beschädigung von Hardware durch Programmierfehler zu verhindern.

Die Grammatik des Bereiches *variables* ist im Anhang C.3 dokumentiert. In Quelltext 6 ist auf deren Basis ein Beispiel für eine valide Definition angegeben. Die Anweisung in Zeile 2 gibt an, dass eine Variable a definiert wird, der der aktuelle Wert 80 zugewiesen ist. Sie hat die Bedeutung „Tau“ und muss im Wertebereich zwischen 4 und 1000 liegen.

```
1 variables {
2 a,   "Tau,[us]",          4,     80,    1000;
3 qs1, "FL-QS delay, 1",  220,    360,     499;
4 qs2, "FL-QS delay, 2",  220,    360,     499;
5 cd,  "Camera delay",   9937,   9956,    9977;
6 f,   "Laser Period",  99000, 100000,  105000;
7 h,   "High-time",       100,    100,     100;
8 n,   "Frames",            1,    200,    2000;
9 }
```

Quellcode 6: Beispiel für den Bereich *variables* nach neuer und alter Sprachdefinition

Bereich settings

Eine gestellte Anforderung ist es, die Konfiguration des Sequenz-Generators in der *Sequenzdatei* zu ermöglichen. Bei *Seco2004* existiert hierzu der Bereich *settings*. Da die Implementierung fehlerhaft ist, können die Einstellungen nicht angewendet werden. Dennoch kann die Grundidee

der Sprachsyntax übernommen werden.

Die Einstellungen können als Wertzuweisung aufgefasst werden. Deshalb wird die Syntax von Wertzuweisungen aus GPLs übernommen. Eine gültige Anweisung besteht aus folgenden Operatoren:

```
Einstellungsparameter = Wunschwert;
```

Ein Wunschwert kann dabei entweder eine Zahl oder ein definierter Bezeichner sein. Der Compiler gibt verfügbare Optionen an, wenn der Nutzer an dieser Stelle Syntaxfehler verursacht.

Die Einstellungsmöglichkeiten definieren sich durch den Funktionsumfang der, für den Sequenz-Generator v5.1 verwendeten, Programmbibliothek . Die Einstellungsparameter werden nicht explizit als Schlüsselwörter definiert, da sie sich in den verschiedenen Hardwarerevisionen unterscheiden. Aus diesem Grund wird die Typkorrektheit, sowie die Umsetzbarkeit der Zuweisung, erst im Rahmen der semantischen Analyse validiert. Um dem Nutzer die Verwendung der Einstellungen zu erleichtern und Fehler zu vermeiden, spielt die Groß- und Kleinschreibung bei Einstellungen und deren Zuweisungen keine Rolle.

Die Grammatik des Bereiches *settings* ist im Anhang C.4 dokumentiert. In Quelltext 7 ist ferner eine lexikalisch korrekte Definition des Bereiches angegeben. In Zeile 2 wird beispielsweise die Einstellung *ArmingSlope*, welche die Interpretation des *Arming*-Signals beeinflusst, angepasst. Es wird konfiguriert, dass es bei *rising*, also einer steigenden Flanke, ausgelöst wird.

Bereich channels

Der Bereich *channels* dient zur Aufzählung der anzusteuernden Kanäle. Der alte Sprachumfang sieht ebenfalls die Benennung der Kanäle vor, um sie in der GUI beschreiben zu können. Aufbauend auf dieser Funktionalität wird die Möglichkeit eingeführt, einen Alias für einen Kanal zu definieren. Dies ist optional und erhöht die Lesbarkeit einer programmierten Sequenz. Anstatt der Kanalnummer können so Aliase verwendet werden. Die Trennung der Namensräume für Variablen, Makros und Aliase verhindert Namenskonflikte zwischen den Defi-

```
1 settings {
2   ArmingSlope    = "rising";
3   Resolution     = 5;
4   Terminator     = "true";
5   TriggerSlope   = "falling";
6   Active         = "high";
7   TriggerSource  = "Generator";
8   WorkingMode    = "Alternative";
9   Generator      = 1e6/f;
10 }
```

Quellcode 7: Beispiel für den Bereich *settings* nach neuer Sprachdefinition

nitionen und steigert somit die Qualität der Entwicklung. Die Syntax-Definition ist semantisch eindeutig, sodass die entsprechenden Variablen, Makros und Aliase korrekt zugeordnet werden.

Eine Anweisung im Bereich *channels* besteht aus folgenden Operatoren:

```
Kanalnummer, Alias, Beschreibung;
```

Die zugehörige Grammatik ist im Anhang C.5 definiert. In Quelltext 8 ist ein Beispiel für eine valide Definition von Kanälen dargestellt. In Zeile 3 wird beispielsweise definiert, dass der Kanal 18 mit dem *Q-Switch 1* verbunden ist und an seiner Stelle auch der Alias `qs1` verwendet werden kann.

```
1 channels {
2   17,        "Flashlamps 1";
3   18, qs1,   "Q-switch 1";
4   19,        "Flashlamps 2";
5   20, qs2,   "Q-switch 2";
6   21,        "Camera";
7   22, start, "Started";
8   23, stop,  "Stopped";
9 }
```

Quellcode 8: Beispiel für den Bereich *channels* nach neuer Sprachdefinition

Bereich menu

Der Bereich *menu* stellt die Möglichkeit bereit, bestimmte Voreinstellungen festzulegen, in denen Variablenwerte angepasst und Kanäle deaktiviert werden können. Der Umfang der Sprachdefinition von *Seco2004* wird hierfür beibehalten werden, profitiert jedoch von der Möglichkeit, Formeln und Aliase zu verwenden. In der alten Anwendung können Menüeinstellungen ferner über die GUI aktiviert werden.

Da im Rahmen von *Seco2015* zunächst keine GUI geplant ist, muss die Vorauswahl einer Menüeinstellung bereits auf Ebene der Skripts möglich sein. Aus Sicht der Sicherheit ist es jedoch auch bei Vorhandensein einer Benutzeroberfläche sinnvoll, einen ungefährlichen, initialen Zustand angeben zu können. Dies wird durch Einführung des Schlüsselworts `default` möglich gemacht.

Eine Menüeinstellung besteht aus folgenden Operatoren:

```
Beschreibung, Wertzuweisungen, inaktive Kanäle ;
```

Die entsprechende Grammatik ist in Anhang C.6 definiert. Ein valides Beispiel für eine Menü-Definition ist in Quelltext 9 gegeben. Das für die Ausführung der Sequenz aktive Menü ist bei diesem Beispiel in Zeile 4 definiert. Es handelt sich um die Einstellung „Flash-lamps only“, welche die Variablen qs1 und qs2 anpasst und Schaltvorgänge der Kanäle mit den Aliasen qs1 und qs2 deaktiviert. Dieses Beispiel demonstriert die Unterstützung verschiedener Namensräume für Variablen und Kanal-Aliase. Es wird deutlich, dass die Verwendung von Variablen und Kanälen in einem Menüeintrag trotz identischer Bezeichner syntaktisch eindeutig ist.

```
1 menus {
2   "High Energy",              qs1=220, qs2=220;
3   "Low Energy",               qs1=360, qs2=qs1;
4   default "Flash-lamps only",   qs1=a, qs2=500, qs1, qs2;
5   "Laser off",                qs1=500, qs2=500, qs1, qs2, 17, 19;
6 }
```

Quellcode 9: Beispiel für den Bereich *menu* nach neuer Sprachdefinition

4.3.4 Definition der Sequenzen

Im Bereich *sequence* wird die konkrete Ablaufsteuerung implementiert. Im Gegensatz zu den anderen Bereichen ist es möglich, mehrere *sequence* Bereiche anzugeben, da der Sequenz-Generator mehrere Sequenzen verwalten kann. In Abschnitt 2.1.1 wird beschrieben, wie der Sequenz-Generator mit mehreren Sequenzen umgeht.

Eine Kernanforderung an *Seco2015* ist, die Lesbarkeit der *Sequenzdatei* zu erhöhen. In *Seco2004* muss jeder einzelne Schaltvorgang in der Sequenz-Definition angeben werden. Dies bedeutet für den Anwender, dass er wiederholte oder mehrfach vorkommende Schaltfolgen explizit definieren muss. Die Analyse der Problemdomäne ergibt, dass dies in der Praxis häufig vorkommt und bisher mittels „Kopieren und Einfügen“ gelöst wird.

Um die damit verbundenen Probleme hinsichtlich Arbeitsaufwand, Übersichtlichkeit und Fehleranfälligkeit zu beheben, wird die bestehende Sprachsyntax um Makros und For-Schleifen erweitert. Makro-Definitionen erfolgen in der *Sequenzdatei* über den Sequenz-Definitionen und können zu Definition von Ereignisfolgen genutzt werden, die an verschiedenen Stellen verwendet werden sollen. Über einen eindeutigen Bezeichner werden sie in die entsprechende Sequenz eingefügt. For-Schleifen können für die wiederholte Ausführung einer Ereignisfolge genutzt werden. Sie können sowohl in Makro-Definitionen als auch in Sequenz-Definitionen verwendet werden.

Zu beachten ist, dass der Schaltzeitpunkt einer Transition bisher immer absolut zum Startzeitpunkt der Sequenz gesehen wird, sodass die Reihenfolge der Anweisungen nicht chronologisch sein muss. Damit das Einfügen von Makros und Schleifen Sinn macht, muss für sie ebenfalls ein Schaltzeitpunkt definiert werden können, der als Referenzstartpunkt für die jeweilig enthaltende Ereignisfolge dient. Für Schleifen muss ferner definierbar sein, wie sich dieser Referenzstartpunkt mit jedem Iterationsschritt verändert. Im Praxisgebrauch stellen Schleifen nur einen Mehrwert dar, wenn die Iterationsdauer nicht von der letzten Transition abhängt, sondern konkret definiert werden kann.

Die Syntax für die Definition einfacher Transitionen wird aus *Seco2004* übernommen. Die Konsistenz wird jedoch verbessert, indem, anders als bisher, die Impulsdauer als mathema-

tischer Ausdruck angegeben werden kann. Im Folgenden sind zwei valide Definition einer Transition angegeben. Im ersten Beispiel wird programmiert, dass der Kanal 18 nach 800 μs für 50 μs aktiviert wird.

```
18, 800, 50;
qs1, 1000-delay, f;
```

Die Syntax für Makros orientiert sich, sowohl bei der Definition als auch bei der Verwendung von Makros, an Funktionsaufrufen aus GPLs. Eine Makrodefinition wird mit dem Schlüsselwort `macro` und der Angabe eines beliebigen Namens eingeleitet. Die Namen für Makros werden in einem separaten Namensraum verwaltet, um Namenskonflikte mit Variablen und Aliasen zu vermeiden. Ein Makro wird in *Seco2015* wie folgt definiert:

```
macro LASER_TRIGGER {
  18, 800, 50;
  qs1, 1000-delay, f;}
```

Ein Makro wird über seinen Namen in eine Sequenz eingefügt werden. Um den Referenzstartpunkt beim Einfügen eines Makros beeinflussen zu können, wird der `offset`-Parameter eingeführt. Er wird wie optionale Funktionsparameter aus Python in runden Klammern angegeben. Eine Unterstützung von Funktionsparametern im Allgemeinen ist jedoch nicht vorgesehen. Die Transitionen des Makros `LASER_TRIGGER` werden mit folgender Syntax 100 μs nach dem übergeordneten Referenzzeitpunkt ausgeführt:

```
LASER_TRIGGER( offset=100 );
```

Die Syntax für For-Schleifen orientiert sich ebenfalls an GPLs. Sie werden mit dem Schlüsselwort `for` gefolgt von der Anzahl der Iterationen eingeleitet. Die zu wiederholende Ereignisfolge muss von geschweiften Klammern umgeben werden. In *Seco2015* können For-Schleifen mit folgender Syntax in Makro- und Sequenz-Definitionen verwendet werden:

```
for 5 {
  LASER_TRIGGER;}
```

Um den Referenzstartpunkt bei Schleifen angeben zu können wird ebenfalls der `offset` Parameter unterstützt. Er wird, analog zum Makroaufruf, in runden Klammern nach der

Anzahl der Iterationen angegeben. Da in der Praxis eine neue Iteration meist nicht unmittelbar nach Ausführung des letzten Schaltvorgangs beginnen soll, muss die Mindestdauer einer Iteration anpassbar sein. Hierfür wird der `wait` Parameter eingefügt. Er wird wie der `offset` im Schleifenkopf definiert. Die Syntax für eine For-Schleife, bei der jede Iteration mindestens 500 μs dauert, sieht wie folgt aus:

```
for 5 (wait = 500) {
  LASER_TRIGGER;}
```

Um die Konsistenz der Syntax sicherzustellen, werden die `offset` und `wait`-Parameter in weiteren Zusammenhängen unterstützt. Dies umfasst die Definition von Sequenzen und Makros sowie das Einfügen von For-Schleifen und Makros. Bei den Definitionen ist die Syntax ebenfalls an optionale Parameter in Python angelegt. Folgende Syntax zeigt die Verwendung der Parameter bei der Definition eines Makros:

```
macro LASER_TRIGGER (offset = 500, wait = 1000) {
  18, 800, 50;
  qs1, 1000-delay, f;}
```

Obwohl nicht für alle Kombinationen Anwendungsfälle existieren, wird in Tabelle 3 die Wirkung der Parameter in den verschiedenen Kontexten deutlich gemacht.

	offset	**wait**
Sequenz-Definition	globaler Referenzstartpunkt	globale Mindestlänge
Makro-Definition	globaler Referenzstartpunkt	globale Mindestlänge
Makro-Aufruf	überschreibe Referenzstartpunkt	überschreibe Mindestlänge
Schleifen-Definition	Referenzstartpunkt	minimale Iterationsdauer

Tabelle 3: Auswirkung der Parameter `wait` und `offset` in verschiedenen Anweisungen

Die Grammatik der Sprachsyntax zur Definition von Sequenzen und Makros ist in Anhang C.7 dokumentiert. In Quellcode 10 ist ein Beispiel für die Definition einer Sequenz unter Verwendung eines zuvor definierten Makros und einer For-Schleife dargestellt.

```
1  macro LASER_TRIGGER(offset=50, wait=500) {
2    17,  cd - qs1,     h;
3    qs1, cd,           h;
4    19,  cd - qs2 + a, h;
5    qs2, cd + a,       h;
6  }
7
8  sequence 0 (wait=f/2) {
9    LASER_TRIGGER;
10 }
11
12 sequence 1 {
13   start, 0, h;
14   for n (offset=0, wait=f) {
15     21, 0, h;
16     LASER_TRIGGER(offset = 100);
17   }
18   stop, n*f, h;
19 }
```

Quellcode 10: Beispiel für den Bereich *sequence* nach neuer Sprachdefinition

4.3.5 Aufbau der Sequenzdatei

Die vorgestellten Bereiche müssen in der *Sequenzdatei* in folgender Reihenfolge angeben werden:

1. Variablen-Definitionen (*variables*)
2. optional: Einstellungen (*settings*)
3. Kanal-Definitionen (*channels*)
4. optional: Menü-Definitionen (*menu*)
5. optional: eine oder mehrere Makro-Defintionen (*macro*)
6. eine oder mehrere Sequenz-Definitionen (*sequence*)

Dies entspricht der Grammatik des Startsymbols, welche in Anhang C.8 angegeben ist.

4.4 Parser

Der Parser ist als Teilsystem für die lexikalische und syntaktische Analyse einer eingelesenen *Sequenzdatei* verantwortlich. Aus den globalen Anforderungen (vgl. Abschnitt 1.3) kann abgeleitet werden, dass der Parser eine Steigerung der Qualität sowie eine Optimie-

rung des zukünftigen Entwicklungsprozesses bewirken soll. Im Detail müssen bei dessen Konzeptionierung folgende Anforderungen beachtet werden:

- Qualitätsteigerung beim Entwickeln von *Sequenzdateien*:
  - Sichergestellte Konsistenz der Sprache
  - Garantierte Zuverlässigkeit des Parsers
  - Fehlermeldungen bei Fehleingaben
- Erleichterungen zukünftiger Entwicklungen am Parser:
  - Erweiterbarkeit
  - Schnelle Fehlerbehebung und Anpassbarkeit
  - Niedrige Einarbeitungszeit

Seco2004 verwendet einen handgeschriebenen Parser, der mit geschachtelten `If`- und `Case`-Anweisungen in C++ umgesetzt ist, und nicht auf einer formal definierten Grammatik basiert. Dies hat den Nachteil, dass die Sprache und der Parser Inkonsistenzen enthalten. Ein weiteres Problem ist der monolithische Aufbau des Codes, der Fehlerbehebungen und Erweiterungen erschwert. Aus diesen Gründen wird im Rahmen von *Seco2015* von der Weiterverfolgung dieses Ansatzes abgesehen.

Eine Alternative zu einem handgeschriebenem Parser ist die Generierung eines Parsers mit einem Parsergenerator. Der Generator erzeugt im Allgemeinen auf Basis einer Grammatik einen ausführbaren Parser, der die Syntax einer Eingabe validieren und in ein abstraktes Ergebnis konvertiert.

Die meisten Parsergeneratoren erzeugen einen Scanner und einen Parser. Dem Scanner liegt die Definition regulärer Ausdrücke zu Grunde. Auf dieser Basis generiert er eine Tokenfolge, die der Parser anschließend weiterverarbeitet. Er überprüft die semantische Korrektheit und systematisiert die Tokens gemäß der Grammatik.

Da der Parser in der Regel auf Basis einer EBNF ähnlichen Definition generiert wird, ist es einfach, Änderungen an der Sprache vorzunehmen. Ferner ist die Laufzeit des Parsers geringer als bei der handgeschriebenen Variante. Außerdem bieten Parsergeneratoren oftmals zusätzliche Funktionen zur Erstellung detaillierter Fehlermeldungen, sodass die Entwicklungszeit verkürzt

wird. Aus diesen Gründen sowie dem in Abschnitt 2.2.4 gezogenem Fazit, wird das Teilsystem Parser mit Hilfe eines generierten Parsers umgesetzt.

4.4.1 Auswahl eines Parsergenerators

Im Folgenden werden gängige Parsergeneratoren im Python Ökosystem verglichen. Auf dieser Basis wird die Entscheidung getroffen, welcher Parsergenerator im Rahmen von *Seco2015* genutzt wird. Im Python Ökosystem existieren verschiedene Parsergeneratoren, von denen *PyParsing*, *Python LEX YACC (PLY)* und *ANTLR* die Meistgenutzten sind.

Bei der Auswahl eines Parsergenerators für *Seco2015* spielt insbesondere der Entwicklungsprozess zur Erstellung und Anpassung des Parsers eine Rolle. Die Definition der zugrundeliegenden Grammatik sollte übersichtlich und nah an der EBNF-Definition sein, damit eine spätere Erweiterung und Wartung des Parsergenerators erleichtert wird. Ferner sollten ein automatisch generierter Parser detaillierte Informationen zu Syntaxfehlern liefern, um dem Nutzer die Fehlerbehebung zu erleichtern.

Eine geringe Relevanz dagegen hat die Geschwindigkeit des Parsers. Die Komplexität der *Sequenzdatei* umfasst zwischen 50 und 500 Codezeilen, sodass die zu erwartende Laufzeit, der mit den oben genannten Werkzeugen generierten Parser, in einem nicht-signifikanten Bereich liegt.

PLY ist eine Python Implementierung der UNIX-Parserwerkzeuge LEX und YACC. Es ist komplett in Python implementiert und bildet die Funktionen der UNIX-Vorbilder ab. Die Regeln des Scanners werden mit Hilfe von regulären Ausdrücken definiert. Die Grammatik des Scanners wird dagegen mit Docstrings abgebildet, deren Struktur vergleichbar mit EBNF ist. Der generierte Parser arbeitet Bottom-Up, sodass es sich bei der Grammatik um eine LR-Grammatik[2] handeln muss.

PLY zeichnet sich durch eine hohe Performance aus und sollte verwendet werden, wenn der Entwickler bereits Erfahrung mit LEX und YACC hat. Da der Parser vor der Programmlaufzeit generiert wird, muss er entsprechend in die Hauptanwendung eingebunden werden. Bei Änderungen an der Grammatik muss der Parser neu generiert werden. Die Definition

[2] Rechtsrekursive, kontextfreie Grammatik

der Regeln mit Hilfe von regulären Ausdrücken ist, ausgehend von einer Grammatik, relativ anspruchsvoll.

ANTLR ist ein in Java implementierter Parsergenerator, der auf Basis einer Grammatik einen ausführbaren Parser in verschiedenen Zielsprachen, zu denen auch Python gehört, generiert. Es existiert ferner das Werkzeug ANTLRWorks, welches das Entwickeln und Debuggen des Parser erleichtert. Der generierte Parser arbeitet Top-Down, sodass es sich bei der zugrundeliegenden Grammatik um eine LL-Grammatik[3] handeln muss.

ANTLR zeichnet sich ebenfalls durch eine hohe Geschwindigkeit aus und sollte verwendet werden, wenn der Entwickler bisher keine Erfahrung mit Parsergeneratoren hat. Der Parser wird auch hier vorab generiert und muss entsprechend in die Hauptanwendung integriert werden. Demzufolge ist ebenfalls eine Neugenerierung erforderlich, wenn die Grammatik verändert wird. Aufgrund des durch ANTLRWorks vereinfachten Entwicklungsprozesses ist die Verwendung von ANTLR im Rahmen von *Seco2015* gegenüber der Verwendung von PLY vorzuziehen.

Das dritte Werkzeug **PyParsing** verfolgt dagegen einen grundsätzlich anderen Ansatz als PLY und ANTLR. Es ist ausschließlich für Python entwickelt und verfolgt den Ansatz, die Definition und Generierung des Parsers in die Hauptanwendung zu integrieren. Die Grammatik wird in einer an die EBNF-Notation angelehnten Syntax direkt im Python-Code definiert. PyParsing stellt verschiedene High-Level Funktionen bereit, mit denen Optionalität, Wiederholungen und andere Eigenschaften umgesetzt werden können. Jedes Nicht-Terminal tritt als Klasse auf, welche Methoden zum Parsen einer Eingabe bereitstellt.

PyParsing ist vorrangig auf die Verarbeitung kleiner Grammatiken ausgelegt und arbeitet deutlich langsamer als ANTLR und PLY. Dafür bietet es eine intuitive Möglichkeit, die Grammatik direkt in Python-Syntax auszudrücken. Aufgrund dessen, dass der Parser erst zur Laufzeit generiert wird, muss er nicht separat erstellt werden. Er wird aus diesem Grund in die Architektur der Hauptanwendung integriert werden.

[3] Linksrekursive, kontextfreie Grammatik

Da die langsameree Ausführungsgeschwindigkeit von PyParsing im Kontext von *Seco2015* nicht spürbar ist und bei der Entwicklung konsequent Python-Konzepte sowie die Nähe zur EBNF-Notation verfolgt werden, wird der Parser von *Seco2015* mit Hilfe von PyParsing generiert.

4.4.2 Generieren des Objektmodells

Der generierte Parser erzeugt aus einer *Sequenzdatei* ein geschachteltes Python-Dictionary, welches dem Syntaxbaum entspricht. Auf Basis dieser Repräsentation kann nicht direkt mit den eingelesenen Daten gearbeitet werden. Der Baum müsste erneut analysiert werden, um gewünschte Informationen zu erhalten. PyParsing verfügt jedoch über Werkzeuge, mit denen die Parserergebnisse unmittelbar verarbeitet werden können.

Ein Konzept zur Lösung dieses Problems ist es, einen Großteil der Analyse bereits während des Parsens durchzuführen. Dies umfasst das Erstellen von Symboltabellen in den Bereichen *variables* und *channels* sowie die Überschreibung der Variablenwerte anhand des standardmäßig ausgewählten Menüeintrags.

Beim Scannen von Sequenzen können Variablen und Makros direkt in der Tokenfolge ersetzt werden, sodass das Ergebnis des Parsens bereits die generische Ereignisfolge enthält. Eine Sprache, welche einen zu *Seco2004* äquivalenten Funktionsumfang abdeckt, kann auf diese Weise geparst werden. Bei umfangreichen Sprachen, wie *Seco2015*, kommt es jedoch zu folgenden Problemen:

- Änderung eines Menüeintrags oder einer Variable erfordern ein erneutes Parsen
- die Bereiche der *Sequenzdatei* müssen in der korrekten Reihenfolge gescannt werden
- Parsen und Übersetzen finden parallel statt und produzieren somit Abhängigkeiten
- Integration neuer Funktionen wird erschwert

Um diese Probleme zu vermeiden, wird ein anderes Konzept zur Verwaltung der Daten entwickelt, auf dessen Basis die Weiterverarbeitung stattfinden kann. Die Grundidee dieses Konzeptes ist es, die Tokenfolge auf ein Objektmodell zu abstrahieren. Dieses entspricht einem aggregativen Objektmodell und ersetzt somit die geschachtelte Ergebnisliste mit einer Hierachie, die dem dekorierten Syntaxbaum entspricht. Die Klassenstruktur wird parallel zur Grammatik definiert und bildet die gleiche Hierachie ab. Klassen von Nicht-Terminalen enthal-

ten Instanzen anderer Nicht-Terminale oder die konkreten Informationen aus den Terminalen. Zur Umsetzung der Weiterverarbeitung können die einzelnen Klassen ferner um konkrete Funktionalität erweitert werden.

Anhand von folgendem Beispiel soll das übernommene Konzept deutlich gemacht werden. Die Grammatik für den Variablenbereich entspricht der in Quelltext 11 angegeben PyParsing Syntax.

```
1  variable_definition = variable_name + "," + identifier + "," +
2    floatnumber("v1") + "," + floatnumber("v2") + "," +
3    floatnumber("v3") + ";"
4  variable_scope = "variables"  + "{" + OneOrMore(variable_definitions) +
       "}"
```

Quellcode 11: PyParsing Syntax zur Definition der Grammatik zum Bereich *variables*

Mit Hilfe von PyParsing wird der Bereich *variables* in der Instanz `VariableScope` zusammengefasst, welches wiederum die einzelnen `variable_definitions` enthält. Das entsprechende UML-Diagramm ist in Abbildung 12 dargestellt. In `IValue` wird der aktuelle Wert der Variable gespeichert.

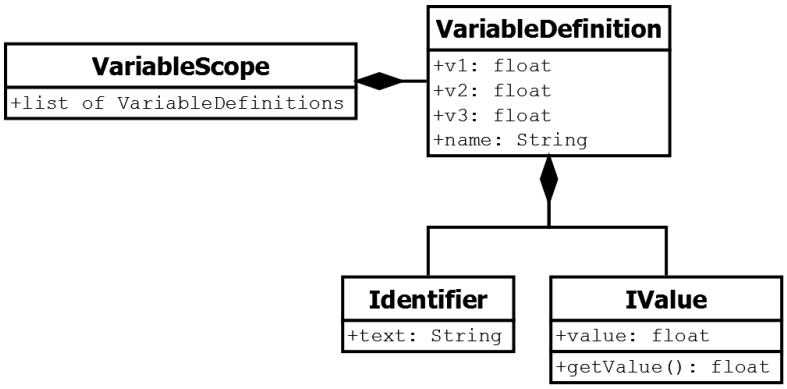


Abbildung 12: UML-Klassendiagramm des Bereiches *variables* im Objektmodell

Im Anhang C werden den anderen Bereichen der Grammatik ebenfalls UML-Klassendiagramme Diagramme zugeordnet.

4.5 Laufzeitumgebung

Hauptaufgabe der Laufzeitumgebung ist das Transformieren der eingelesenen Daten in eine generische Form. Im Compilerbau wird diese Form als Zwischencode beschrieben, auf dessen Basis der Zielcode generiert wird. An die generische Ereignisfolge wird die Anforderung gestellt, dass die Ereignisse in einer Form repräsentiert werden, die unabhängig von künftigen Zielplattformen möglichst allgemeingültig ist. Zusammen mit dem Parser wird so das Einlesen der *Sequenzdatei* vom Übersetzungsvorgang in die Zielsprache gekapselt.

Die generische Ereignisfolge ist eine hardwareunabhängige Repräsentation der Sequenz. Mit ihr ist die Schnittstelle zwischen der allgemeinen Laufzeitumgebung und dem plattformabhängigen Übersetzer spezifiziert. Um die generische Ereignisfolge zu erzeugen, müssen Makros, Variablen und Aliase aufgelöst und For-Schleifen ausgerollt werden. Ferner müssen mathematische Ausdrücke ausgerechnet und die `wait`- und `offset`-Parameter angewendet werden. Die generische Ereignisfolge soll ausschließlich aus Transitionen mit absoluten Zahlenwerten bestehen.

Die Laufzeitumgebung muss hierfür Symboltabellen anlegen, mit deren Hilfe Makros, Variablen und Aliase aufgelöst werden können. Ferner muss die Laufzeitumgebung in der Lage sein Menüeinträge zu aktivieren.

Um die Funktionalität der Laufzeitumgebung umzusetzen, muss das Objektmodell um Schnittstellen erweitert werden, die eine Verarbeitung der geparsten Daten ermöglicht. Dies betrifft insbesondere die Klassen, in denen mathematische Ausdrücke und die einzelnen Ereignistypen verwaltet werden.

4.6 Optimierer und Übersetzter

Der Optimierer verbessert die generische Ereignisfolge und bereitet sie für den Übersetzer vor. Eine Anforderung an dieses Teilsystem ist es, dass ein Großteil der Verbesserungen möglichst unabhängig von der Zielplattform vorgenommen werden. Aus diesem Grund wird das Teilsystem in einen allgemeinen und einen spezifischen Optimierer unterteilt. Im Rahmen der spezifischen Optimierung wird ferner eine Validierung der technischen Ausführbarkeit der

Sequenz vorgenommen.

Der Übersetzer muss abhängig von der Zielplattform umgesetzt werden. Im Rahmen von *Seco2015* wird der Übersetzer für einen Hardsoft Sequenz-Generator v5.1 entwickelt. Die Zielsprache wird durch die Programmbibliothek zur Übertragung einer Sequenz vorgegeben. Für die notwendige Transformation werden die Python-Binäroperatoren sowie die Listenoperationen des Moduls `numpy` genutzt.

4.7 Hardwareansteuerung

Die Hardwareansteuerung umfasst das Verbinden und Konfigurieren eines Sequenz-Generators sowie die Übertragung der in Zielsprache vorliegenden Sequenz. Eine Anforderung hierbei ist es, eine Architektur zu konzeptionieren, welche eine einfache Integration weiterer Hardwarerevisionen ermöglicht.

Die Instanziierung der Hardwareansteuerung ist abhängig davon, welche Sequenz-Generatoren an den Computer angeschlossen sind. Aus diesem Grund wird sie mit den Entwurfsmuster Fabrik [37, Kapitel 3] realisiert. Die Fabrik nutzt verschiedene Programmbibliotheken, um herauszufinden, welcher Sequenz-Generator für einen Verbindungsaufbau zur Verfügung steht. Auf dieser Basis generiert sie eine entsprechende Instanz, über die die Hardwareansteuerung möglich ist. Im Rahmen der Hardwareansteuerung wird ferner eine allgemeine Schnittstelle entwickelt, über die verschiedene Zielplattformen angesprochen werden können.

Im Rahmen von *Seco2015* wird die Hardwareansteuerung für einen Hardsoft Sequenz-Generator v5.1 umgesetzt. Hierfür steht eine Programmbibliothek bereit, die über das Modul `ctypes` angesprochen werden kann.

4.8 Zusammenfassung

Im folgenden Abschnitt wird das Konzept für *Seco2015* zusammengefasst. Abbildung 13 zeigt, in welcher Reihenfolge die Teilsysteme des *Seco2015* Compilers innerhalb der **Haupt-**

anwendung ausgeführt werden.

Abbildung 13: Übersicht der Teilsysteme des Compilers

Die **Grammatik** basiert auf der alten Konfigurationssprache und wird als Ergebnis der Problemanalyse um Makros, For-Schleifen und Konfigurationsmöglichkeiten erweitert und hinsichtlich Ihrer Konsistenz verbessert. Sie ist formal definiert und orientiert sich bei der Syntax zusätzlich an Skriptsprachen wie Python.

Als **Parser** wird ein mit dem Python-Modul PyParsing generierter Parser verwendet. Der Syntaxbaum wird in Form eines Objektmodells generiert, welches sich an der Grammatik orientiert.

Die **Laufzeitumgebung** verwaltet das Objektmodell und erzeugt eine generische Ereignisfolge. Sie ermöglicht die Anpassung von Variablen- und Kanal-Definitionen zur Laufzeit. Um dies möglich zu machen, muss das Objektmodell um Funktionalität erweitert werden.

Die generische Ereignisfolge wird vom **Optimierer** weiterverarbeitet. Er wird in einen allgemeinen und ein spezifischen Optimierer unterteilt, die neben der Optimierung auch die Validierung der programmierten Sequenz vornehmen. Der **Übersetzer** transformiert die optimierte Ereignisfolge in die vom Sequenz-Generator erwartete Zielsprache. Für die Transformationen werden Python-Binäroperatoren und die Listenoperationen des Moduls `numpy` verwendet werden.

Für die **Hardwareansteuerung** wird zunächst eine Schnittstelle zum Konfigurieren, Initialisieren und der Übertragung einer Sequenz geschaffen. Das Verbinden eines konkreten Sequenz-Generators wird in einer Fabrik umgesetzt. Die für die Hardwareansteuerung erforderlichen Programmbibliotheken werden über `ctypes` eingebunden.

5 Umsetzung

Im folgenden Kapitel wird die Umsetzung des Compilers auf Basis des Konzeptes erläutert. Abschnitt 5.1 beschreibt die Umsetzung des Parsers. Dies umfasst die Definition der Grammatik mit Hilfe von PyParsing sowie das Generieren des Objektmodells. Die Implementation der Laufzeitumgebung wird in Abschnitt 5.2 beschrieben. Hierfür werden die Vorbereitungen zum Erzeugen der generischen Ereignisfolge sowie der Erzeugungsvorgang thematisiert.

In Abschnitt 5.3 werden Strategien zur Verbesserung der Rohdaten entwickelt, welche schließlich in Abschnitt 5.4 in die Zielsprache transformiert werden. In Abschnitt 5.5 wird die Umsetzung der Hardwareeinbindung sowie deren Integration in *Seco2015* beschrieben.

Die vorgestellten Implementierungen werden in die in Abschnitt 5.5 beschriebenen Hauptanwendung integriert. Inhalt des Abschnitts ist ebenfalls eine Übersicht der Abläufe bei Ausführung von *Seco2015*.

5.1 Parser

Der Parser ist im Rahmen von *Seco2015* für das Einlesen einer *Sequenzdatei* und das Erzeugen des Objektmodells verantwortlich. Er wird mit Hilfe von PyParsing umgesetzt. Im Folgenden wird beschrieben, wie die Grammatik in PyParsing definiert wird und mit welchen Werkzeugen das Objektmodell aufgebaut wird.

5.1.1 Definition der Grammatik mit PyParsing

PyParsing ermöglicht es, eine Grammatik direkt im Python-Code zu definieren. Die Definition ist an die EBNF-Notation angelehnt und kann, ausgehend von einer formalen Grammatik, wie folgt angeben werden. Das Modul PyParsing wird in Codebeispielen mit `pp` abgekürzt:

- **Terminale**: String oder `pp.Literal(String)`
- **Nicht-Terminale**: Variablen-Definition oder Wertzuweisung
- **Und-Verknüpfung**: Operator + zwischen PyParsing-Variablen
- **Oder-Verknüpfung**: Operator | zwischen PyParsing-Variablen
- **Optionalität**: `pp.Optional(Terminal/Nicht-Terminal)`

- **Wiederholung**: `pp.ZeroOrMore(Terminal/Nicht-Terminal)` oder `pp.OneOrMore`

Außerdem ignoriert PyParsing standardmäßig Freizeichen und Zeilenumbrüche. Mit der Klasse `pp.Words` können die Zeichen eines Alphabetes mit Hilfe der Übergabeparameter `pp.nums`, `pp.alphas` und `pp.alphanums` festgelegt werden. Die PyParsing Grammatik in Quelltext 12 zeigt die formale Definition des Bereiches *variables* einer Sequenzdatei in PyParsing (vgl. Grammatiken in Anhang C.1 und C.3).

```
1 variablen_name = Word(alphas + "_", alphanums)
2 float = Optional("+" | "-") + Word(nums) + Optional("." + Word(nums))
3 identifier = Literal('"') + Word(alphanums + "-,+[]._ ") + Literal('"')
4
5 variable_definition = variable_name + "," + identifier + "," + float +
      "," + float + "," + float + ";"
6 variable_scope = "variables"  + "{" + OneOrMore(variable_definition) +
      "}"
```

Quellcode 12: PyParsing Syntax zur Definition der Grammatik zum Bereich *variables*

Rekursive Definitionen sind mit der Klasse `pp.Forward` möglich. Sie wird einem Nicht-Terminal zugewiesen und gibt an, dass es später im Quelltext konkret definiert wird. Es kann jedoch vorher in andere Reduktionsregeln integriert werden.

Jedes Nicht-Terminal einer PyParsing-Grammatik tritt als Klasse auf, welche verschiedene Methoden zum Parsen eines Eingabe-Strings bereitstellt. Im Rahmen von *Seco2015* sind dabei folgende Methoden relevant:

- `pp.ParseFile(Filename)`, zum Einlesen einer *Sequenzdatei* von der Festplatte
- `pp.ParseString(String)`, zum Einlesen von in Testfällen vordefinierten *Sequenzdateien*

Die Methoden werden auf das Startsymbol angewendet und liefern standardmäßig den dekorierten Syntaxbaum in Form eines geschachtelten Python-Dictionaries zurück.

5.1.2 Aufbau des Objektmodells

PyParsing macht die Elemente des geschachtelten Dictionaries standardmäßig über den Namen der Regel, auf deren Basis sie erstellt werden, zugänglich. In den einzelnen Regeln können

diese Namen mit Aufruf der Funktion `pp.setResultsName(Name)` manipuliert werden, um zum Beispiel verschiedene Zahlenwerte, bei denen die Regel für eine Gleitkommazahl erfüllt ist, unterscheiden zu können.

Darüber hinaus bietet PyParsing die Möglichkeit, die Elemente des Syntaxbaums vor dem Hinzufügen in den Baum zu verändern. Erreicht wird dies, indem einer Regel mit der Funktion `pp.setParseAction(Functionname)` eine beliebige Funktion zugewiesen wird. Diese Funktion erhält als Eingabewert die Tokenfolge als Liste aus Strings. Der Rückgabewert der Funktion wird anstatt des String-Tokens in den Syntaxbaum eingefügt.

In Abschnitt 4.4 wird spezifiziert, dass der Syntaxbaum der Sequenzdatei während des Parsevorgangs in ein Objektmodell abstrahiert werden soll. Jede Klasse entspricht einem Nicht-Terminal, welches als Attribute weitere Nicht-Terminale oder Terminale enthält. Die Konstruktoren der Klassen erwarten als Übergabeparameter die Parameter, welche die Funktion `pp.setParseAction(Konstruktorname)` übergibt.

5.2 Laufzeitumgebung

Die Laufzeitumgebung verwaltet die geparsten Daten und stellt Schnittstellen zur Aktivierung von Menüeinträgen sowie zur Erzeugung der generischen Ereignisfolge bereit. Die Architektur der Laufzeitumgebung ist in Abbildung 14 in Form eines UML-Diagramms dargestellt. Hierbei wird deutlich dass in der Laufzeitumgebung die übergeordneten Nicht-Terminale des Objektmodells aggregiert werden.

Die generische Ereignisfolge besteht aus einzelnen Ereignissen, deren Form der Sprachdefinition einer einfachen Transition ohne Variablen und mathematische Ausdrücke entspricht. Für jede Sequenz-Definition muss eine Liste mit Einträgen folgender Form generiert werden:

```
(Kanalnummer, Schaltzeitpunkt, Impulsdauer)
```

Im Folgenden wird beschrieben, welche Teilaufgaben umgesetzt werden müssen, um die generische Ereignisfolge zu erzeugen.

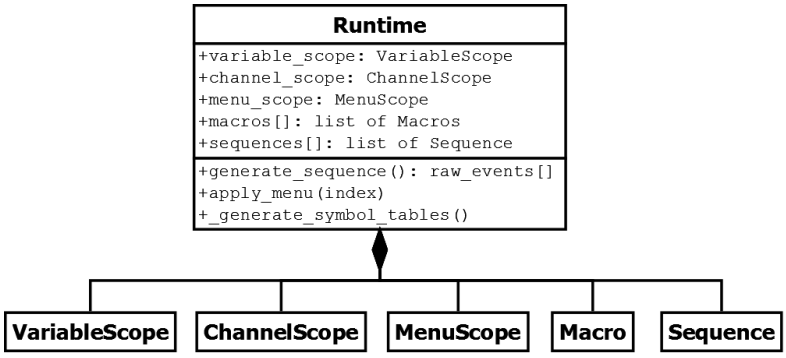


Abbildung 14: UML-Klassendiagramm der Laufzeitumgebung

5.2.1 Generieren der Symboltabellen

Um die Aliase, Variablen und Makros auflösen zu können, werden zunächst die Symboltabellen erstellt. Die Klassen der Bereiche *macro*, *variables* und *channels* verfügen über Methoden, die es der Laufzeitumgebung ermöglichen, die Symboltabellen abzurufen. Am Beispiel der Variablen soll im folgenden deutlich gemacht werden, wie die Symboltabellen generiert werden.

Die Klasse `VariableScope` enthält die einzelnen Instanzen der Klasse `VariableDefinition`. Bei der Erzeugung der Symboltabelle wird ein Python-Dictionary angelegt, welches den entsprechenden Namen der Referenz auf die zugehörige Instanz zuordnet. Die Symboltabelle wird der Laufzeitumgebung übergeben und wird zum Beispiel beim Auflösen eines mathematischen Ausdrucks der Klasse `Variables` genutzt, um den konkreten Wert einer Variable zu ermitteln.

5.2.2 Aktivieren von Menüeinträgen

Die Laufzeitumgebung verfügt über die Schnittstelle, einen bestimmten Menüeintrag zu aktivieren. Nach Aufbau des Objektmodells wird automatisch der Standard-Menüeintrag (`default`) aktiviert. In Menüeinträgen können die Werte von Variablen verändert und Kanäle deaktiviert werden.

Zum Anpassen einer Variable wird über die Symboltabelle auf die *set_value()*-Methode der jeweiligen Variablen-Definition im Objektmodell zugegriffen. Analog hierzu werden Kanäle

über die `set_active()`-Methode der jeweiligen Kanal-Definition im Objektmodell deaktiviert.

Beim Erzeugen der generischen Ereignisfolge werden die entsprechenden Definitionen aufgelöst, sodass die aktualisierten Werte übernommen werden. Ferner wird bei der Verwendung von Kanälen geprüft, ob diese aktiv sind. Transitionen inaktiver Kanäle werden nicht in die generische Ereignisfolge übernommen.

5.2.3 Auflösen von Ereignissen

Um das Erzeugen der generischen Ereignisfolge möglich zu machen, muss die Funktionalität des Objektmodells der *Sequenzdatei* erweitert werden. Dies umfasst die Klassen, in denen Ereignisse und mathematische Ausdrücke abgebildet werden. Diese sind rekursiv definiert und können mit den Entwurfsmustern Besucher oder Interpreter aufgelöst werden [38, Kapitel 2] [37, Kapitel 5]. Beim Besucher wird die Logik extern zur bestehenden Objektstruktur definiert. Die Objektstruktur enthält lediglich Methoden, die einen konkreten Besucher akzeptieren und ihm die eigene Instanz übergeben, sodass eine konkrete Aktion ausgeführt werden kann. Dieses Vorgehen ist insbesondere bei einer komplexen Grammatik, auf der eine Vielzahl unterschiedlicher Operationen ausgeführt werden soll, empfehlenswert. Das Datenmodell wird von den Aktionen getrennt, was die Wartung der Anwendung erleichtert.

Beim Interpreter wird die Logik direkt in die Objektstruktur integriert. Über Schnittstellen können Aktionen kategorisiert werden, die in den jeweiligen Klassen eine konkrete Implementation enthalten. Für kleinere Objektmodelle, auf denen nur eine geringe Anzahl an Operationen durchgeführt werden, ist die Verwendung dieses Entwurfsmusters empfohlen, da es einfacher zu warten und zu entwickeln ist.

Beide Ansätze können in *Seco2015* ohne gravierende Nachteile umgesetzt werden. Aufgrund der geringen Komplexität des Objektmodells werden die mathematischen Ausdrücke sowie die einzelnen Ereignisse mit Hilfe eines Interpreters aufgelöst.

Das Klassendiagramm der Ereignisse innerhalb des Objektmodells ist in Abbildung 15 dargestellt. Die verschiedenen Ereignisse erben von der abstrakten Klasse `IEvent` und werden vom Parser entsprechend der Grammatik instanziiert. Alle Events außer `Macro` und `Transition`

enthalten eine Liste aus `IEvents`, sodass die Anforderung, beliebige Schachtelungen zu ermöglichen, erfüllt wird.

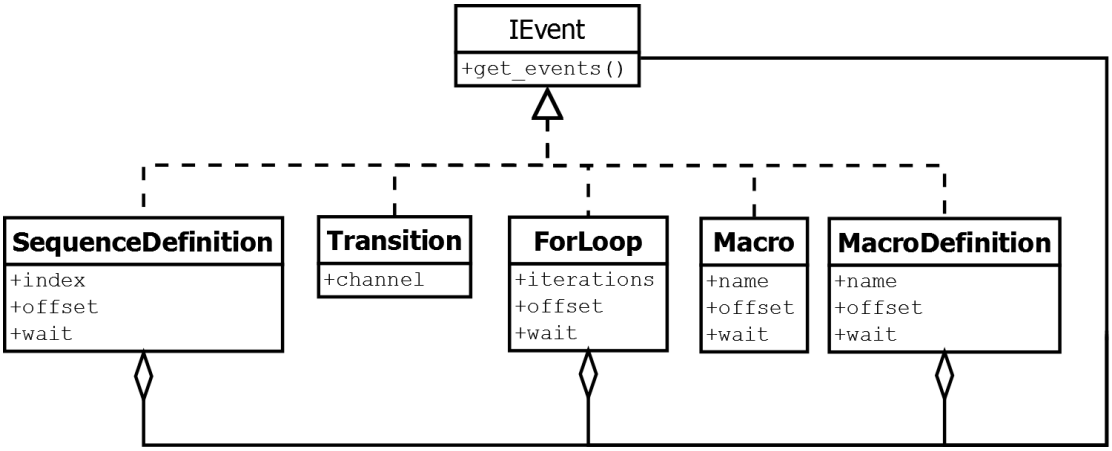


Abbildung 15: UML-Klassendiagramm der Ereignisse

Ferner enthält jede Instanz die Funktion `get_events()`, mit deren Hilfe die entsprechenden Teilfolgen erzeugt werden. Durch den Interpreter verhält sich die Funktion für die verschiedenen Ereignistypen unterschiedlich. Das entsprechende Verhalten wird im Folgenden vorgestellt:

- `SequenceDefinition`: ruft `get_events()` für alle Einträge der Ereignisliste auf und erzeugt somit die generische Ereignisfolge
- `Transition`: liefert ein generisches Ereignis[1] zurück
- `ForLoop`: ruft `get_events()` für alle Einträge der Ergebnisliste auf, kopiert die generischen Ereignisse entsprechend der Anzahl der Iterationen und liefert das Ergebnis zurück
- `Macro`: Löst das Macro mit Hilfe der Symboltabelle auf, ruft get_events() der entsprechenden `MacroDefinition` auf und liefert das Ergebnis zurück
- `MacroDefinition`: siehe `SequenceDefinition`

Die Laufzeitumgebung ruft `get_events()` für alle `SequenceDefinitions` auf und erzeugt somit die generische Ereignisfolge.

[1] Ereignis der Form (Kanalnummer, Schaltzeitpunkt, Impulsdauer)

5.2.4 Auflösen von mathematischen Ausdrücken

Die mathematischen Ausdrücke und Variablen in den einzelnen Ereignissen werden analog dazu nach dem gleichen Konzept aufgelöst. Sie erben von der abstrakten Klasse `IValue` und werden vom Parser entsprechend der Grammatik instanziiert. Das Klassendiagramm ist in Abbildung 16 dargestellt.

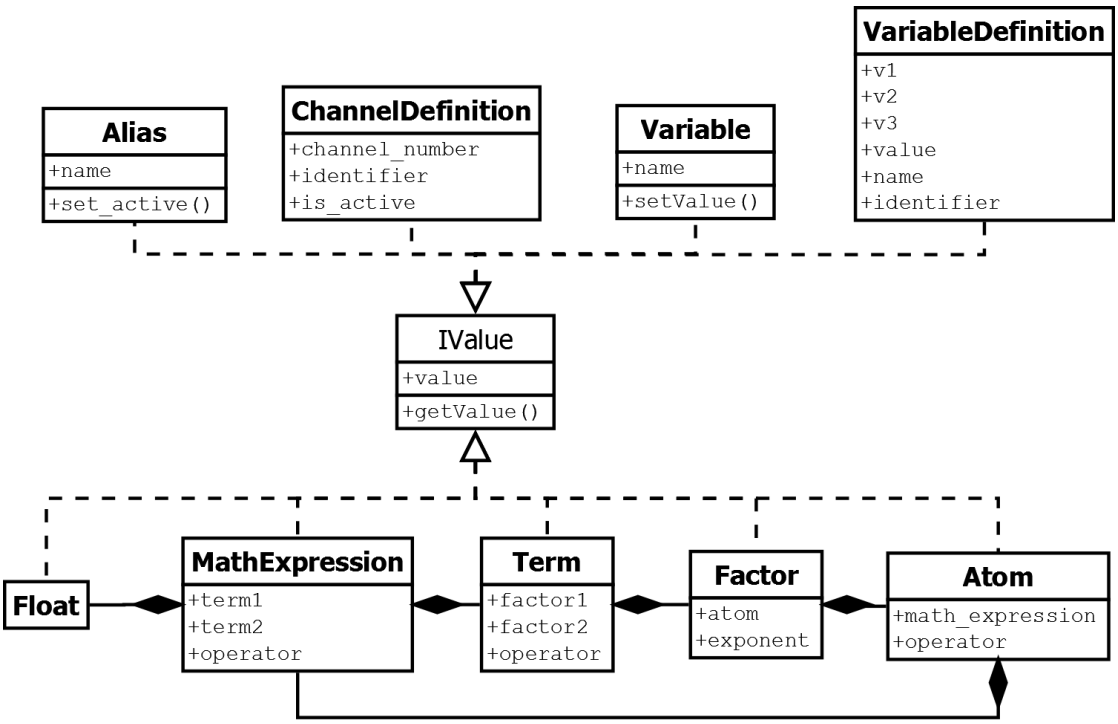


Abbildung 16: UML-Klassendiagramm der mathematischen Ausdrücke im Objektmodell

Alle Instanzen verfügen über die Funktion `get_value`, die einen konkreten Wert zurückliefert. Entsprechend des Interpreter-Musters werden die unterschiedlichen Implementierungen dieser Funktion ausgeführt. Folgende Aufzählung macht dies deutlich:

- `Alias`: ruft mit Hilfe der Symboltabelle die Funktion `get_value()` der zugehörigen `ChannelDefinition` auf und liefert den Wert zurück
- `ChannelDefinition`: liefert die Kanalnummer zurück
- `Variable`: ruft mit Hilfe der Symboltabelle die Funktion `get_value()` der zugehörigen `VariableDefinition` auf und liefert den Wert zurück

- `VariableDefinition`: liefert den Median der 3 angegebenen Werte zurück
- `Float`: liefert eine konkrete Zahl zurück
- `MathExpression`: ruft `get_value()` aller Werte auf und liefert die Summe oder die Differenz davon zurück
- `Term`: ruft `get_value()` aller Werte auf und liefert das Produkt oder den Quotient davon zurück
- `Factor`: ruft `get_value()` aller Werte auf und liefert das Ergebnis der Potenz zurück
- `Atom`: ruft `get_value()` des Werts auf und liefert ihn zurück

5.2.5 Anwenden der wait- und offset-Parameter

Die wait- und offset-Parameter stehen für die Definition von Makros, Sequenzen, For-Schleifen und den Aufruf von Makros zu Verfügung. Die unterschiedliche Bedeutung ist in Tabelle 3 aus Abschnitt 4.3.2 dargestellt.

Grundsätzlich enthält jede `IEvent` Instanz den geparsten Wert für offset und wait, oder 0, insofern diese nicht vom Nutzer definiert worden sind. Die Anwendung des offset- und wait-Parameters wird in der jeweiligen `get_events()` Methode umgesetzt.

Der wait-Parameter wird vor Erzeugung der generischen Ereignisfolge angewendet. Hierzu wird der letzte Schaltvorgang aus der Ereignisliste ermittelt und mit dem definierten wait-Wert verglichen. Sollte dieser überschritten sein, wird keine Anpassung vorgenommen. Anderenfalls wird ein Dummy-Event zum Zeitpunkt der letzten Transition mit der Impulsdauer bis zum Erreichen des Mindestdauer eingefügt.

Der offset-Parameter wird im Gegensatz dazu nach dem Erzeugen der generischen Ereignisfolge angewendet. Er wird entsprechend zu allen Ereignissen der generischen Teilfolge addiert.

5.3 Optimierer

Der Optimierer bereitet die generische Ereignisfolge auf den Übersetzer vor. Unabhängig von der Hardwarerevision müssen die generischen Ereignisse in absolute Zustände zerlegt werden.

Ein absoluter Zustand besteht aus einem Zeitpunkt und der Information, ob auf einem Kanal einen HIGH oder LOW Pegel anliegt.

```
Kanal, absoluter Zeitpunkt, Pegel(HIGH, LOW);
```

Die Folge der absoluten Zustände wird zunächst chronologisch sortiert. Anschließend werden folgende allgemeinen Optimierungen vorgenommen:

- Anwenden eines globalen Offsets, falls Ereignisse einen negativen Startzeitpunkt haben
- Zusammenlegen von Transitionen, die zum gleichen Zeitpunkt stattfinden

Die optimierte Transitionsfolge wird nun an den spezifischen Optimierer übergeben, der abhängig vom zum Einsatz kommenden Sequenz-Generator ist. Er setzt folgende Optimierungen umsetzen:

- Ermitteln und Anwenden der Frequenz als größter gemeinsamer Teiler aller Schaltzeitpunkte
- Einfügen von Dummy-Events falls der Abstand zwischen zwei Transitionen zu groß ist
- Validierung der technischen Ausführbarkeit der Sequenz

Die allgemeinen und spezifischen Optimierungen erfolgen mit Hilfe von Python-Listenoperationen und dem Modul `numpy`. Die optimierte Transitionsfolge wird im Anschluss an den Übersetzer übergeben. Abbildung 17 zeigt das Klassendiagramm des Optimierers. Der allgemeine Optimierer ist Bestandteil der abstrakten Sequenz-Generator-Klasse und wird von der jeweiligen konkreten Sequenz-Generator-Klasse aufgerufen. Die spezielle Optimierung findet schließlich in der Sequenz-Generator-Klasse statt.

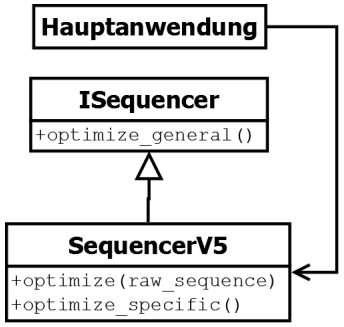


Abbildung 17: UML-Klassendiagramm des Optimierers

5.4 Übersetzer

Der Übersetzer wandelt die vom Optimierer generierte Transitionsfolge in die Zielsprache um. Das Format der Zielsprache ist durch den Aufruf der Programmbibliothek zum Übertragen einer Ereignisfolge definiert. Eine Ereignis, welches an die API übergeben werden kann, besteht aus den in Tabelle 4 spezifizierten sechs Bytes.

Byte	Bedeutung
[1][2][3]	Auflösungsschritte bis zum nächsten Ereignis
[4]	Pegel (HIGH = 1, LOW = 0) der Kanäle 1-8 bitweise
[5]	Pegel (HIGH = 1, LOW = 0) der Kanäle 9-16 bitweise
[6]	Status des Ereignis (Normal = 0, Sequenzende = 2, Programmende = 3)

Tabelle 4: Kodierung eines Ereignisses für den Programmbibliothek-Aufruf des Sequenz-Generators

Mit Hilfe der in Python verfügbaren Binäroperatoren sowie den Listenoperationen wird die generische Ereignisfolge in die Zielsprache transformiert. Sobald das Ende einer Sequenz erreicht ist, wird die entsprechende Eigenschaft hinzugefügt, sodass der Sequenz-Generator die verschiedenen Sequenzen voneinander unterscheiden kann.

Der Übersetzer wird in der konkreten Sequenz-Generator-Klasse implementiert und übergibt die generierte Ereignisfolge an die Hardwareansteuerung.

5.5 Hardwareansteuerung

Für den Sequenz-Generator der Hardwarerevision v5.1 liegt eine in Turbo-Pascal entwickelte Programmbibliothek in Form einer Dynamic Link Library (DLL) vor. Diese wird über die Aufrufkonvention `stdcall` angesprochen und stellt folgende Funktionalität bereit:

- Auflisten und Verbinden von angeschlossenen Sequenz-Generatoren
- Definition von Einstellungen wie *WorkingMode*, *ArmingSlope*, *Resolution*, *TriggerSource*
- Auslösen von Arming und Trigger Events
- Ausführen der *Start* und *Stopp*
- Übertragen von Transitionen

- Manipulation der aktuellen Ausführungsadresse
- Deaktivieren der Bedienelemente

Die Hardwareansteuerung wird ebenfalls in der konkreten Sequenz-Generator-Klasse (hier `SequencerV5`) implementiert. Um die Erweiterbarkeit der Anwendung sicherzustellen, wird die programmierbare Funktionalität anderer Hardwarerevisionen des Sequenz-Generators analysiert. Die Schnittmenge der gemeinsamen Funktionen wird auf das Interface *ISequencer* abstrahiert, über das die Hauptanwendung auf die konkrete Implementation zugreift. Das Klassendiagramm der hardwarespezifischen Implementation ist in Abbildung 18 dargestellt. Auf die Funktionen wird in den folgenden Abschnitten näher eingegangen.

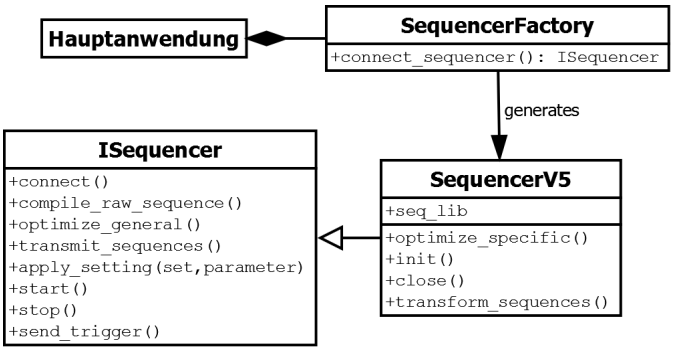


Abbildung 18: UML-Klassendiagramm der Hardwareansteuerung

5.5.1 Einbindung der Hardwarefunktionen

Die API für den zur Verfügung stehenden Sequenz-Generator v5.1 wird mit Hilfe des Python Moduls `ctypes` in die Klasse `SequencerV5` integriert. Die Klasse wird als Adapter für die Programmbibliothek umgesetzt, indem sie die einzelnen API-Funktionen in entsprechende Methoden abstrahiert.

Eine Adapterfunktion enthält die Definition und Verarbeitung der Übergabeparameter sowie den mit `ctypes` realisierten Bibliotheksaufruf. Da `ctypes` für den Aufruf von C-Bibliotheken entwickelt wurde, muss bei der Definition der Übergabeparameter die in Tabelle 5 aufgezeigte Transformation der Datentypen beachtet werden.

Pascal-Datentyp	C-Types Entsprechung	Beschreibung, Größe
Byte	c_ubyte()	8 Bit
Word	c_ushort()	16 Bit
LongWord	c_ulong()	32 Bit
Var	Reference()	Zeiger
WideString	comtypes.BSTR()	COM-spezifischer String Datentyp (16-bit pro Zeichen)
OleVariant	comtypes.VARIANT()	COM-spezifischer Datentyp, der erst zur Laufzeit definiert wird

Tabelle 5: Umwandlung von Pascal-Datentypen in `ctypes` Instanzen

Alle Funktionen der API werden über die Aufrufkonvention `stdcall` aufgerufen. Die Bibliothek ist in der `SequencerV5`-Klasse instanziiert und kann über die entsprechenden Adapterfunktionen aufgerufen werden.

5.5.2 Verbinden eines Sequenz-Generators

Die Hauptanwendung instanziiert die Klasse `SequencerFactory`, welche für die Verbindung eines Sequenz-Generators verantwortlich ist. Da die Kommandozeilenapplikation *Seco2015* ohne Konfiguration lauffähig sein soll, ist die Fabrik darauf ausgelegt, eine Verbindung mit dem ersten verfügbaren Sequenz-Generator aufzubauen. Da aktuell nur Sequenz-Generatoren von Hardsoft in der Hardwarerevision 5.1 unterstützt werden, wird die entsprechende Klasse instanziiert und mit der `connect` Funktion der erste verfügbare Sequenz-Generator verbunden. Bevor die verbundene Instanz von der Fabrik zurückgegeben wird, werden folgende Hardwareinitialisierungen ausgeführt:

1. Deaktivieren der Hardware-Bedienfelder am Sequenz-Generator
2. Senden des *Stop* Befehls
3. Zurücksetzen der Ausgänge und der aktuellen Ausführungsadresse
4. Aktivieren initialer Einstellungen für *Arming-Slope*, *Trigger-Input*, *Working-Mode*, *Trigger-Source*, *Terminator* und *Trigger-Divider*

Der Sequenz-Generator ist nun in einer `SequencerV5` Instanz verbunden und stellt die beschriebenen Funktionen über die `ISequencer` Schnittstelle bereit.

5.6 Hauptanwendung

Die Hauptanwendung ermöglicht es dem Nutzer, den Compiler *Seco2015* auszuführen und eine *Sequenzdatei* zu übersetzen. Sie dient zur Demonstration der Funktionsfähigkeit des Compilers und kann die übersetzte Sequenz an einen verbundenen Sequenz-Generator übertragen, sowie die Ausführung starten und beenden.

Der Pfad der einzulesenden *Sequenzdatei* kann dem Compiler als Kommandozeilenparameter übergeben werden. Diese wird über die Eigenschaft `argv` des Moduls `sys` ausgelesen und an den Parser übergeben. Wird die *Sequenzdatei* unter dem angegebenen Pfad gefunden, werden folgende Schritte ausführt.

1. Generieren und Ausführen des Parsers zum Erhalten der Laufzeitumgebung
2. Warten auf Nutzerbestätigung zum Starten des Übersetzungsvorgangs
3. Verbinden eines Sequenz-Generators
4. Anwenden der Laufzeitkonfiguration
5. Übergeben der generischen Ereignisfolge an den Übersetzer
6. Warten auf Nutzerbestätigung zum Starten der Übertragung
7. Übertragen der übersetzten Sequenz
8. Senden eines Start-Befehls
9. Warten auf Nutzereingabe zur Unterbrechung der Sequenz
10. Senden eines Stop-Befehls

Für die Instanziierung des Sequenz-Generators wird ein Kontextmanager genutzt. Er stellt sicher, dass die allokierten Ressourcen nach Programmablauf freigegeben werden und ermöglicht es darüber hinaus, Aufgaben zu definieren die beim Trennen des Sequenz-Generators ausgeführt werden. Dies wird mit Hilfe des Python-Moduls `contextlib` umgesetzt.

Hierzu wird die Fabrik zur Verbindung eines Sequenz-Generators in der `closing`-Umgebung des Kontextmanagers aufgerufen. Sobald der Programmablauf innerhalb dieser Umgebung abgeschlossen ist, wird automatisch die Methode `close()` der zugehörigen `Sequencer` Klasse ausgeführt. Diese führt bei Unterbrechung oder Beendigung einer Sequenz folgende Schritte aus:

- Unterbrechen der aktuellen Ausführung

- Zurücksetzen der Ausgänge
- Aktivieren der Hardwareeingabe am Sequenz-Generator

Zusammengefasst kann der Nutzer die Hauptanwendung zum Kompilieren einer, auf Basis der neuen Sprachsyntax erstellten, *Sequenzdatei* nutzen. *Seco2015* informiert den Nutzer über Syntax- und Semantikfehler und gibt Rückmeldung ob eine programmierte Sequenz technisch valide ist. Auf Bestätigung des Nutzers überträgt es die Sequenz an einen angeschlossenen Sequenz-Generator und versetzt diesen anschließend in einen ausführungsbereiten Zustand. Die Hauptanwendung wird auch zur Ausführung des Akzeptanztests verwendet.

6 Test

Die Entwicklung der alten Software *Seco2004* umfasst kein Qualitätskonzept in Form von Softwaretests. Probleme beim Auflösen der mathematischen Ausdrücke sowie bei der Ausführung bestimmter Sequenzfolgen sind Fehler, die erst bei der Benutzung der Software aufgefallen sind. Um das Risiko von auftretenden Fehlern zu minimieren, ist es Bestandteil der Aufgabenstellung, den Compiler *Seco2015* in geeignetem Maße zu testen.

In Abschnitt 6.1 werden die Grundlagen des Testens vorgestellt. Dies umfasst den Testprozess im Allgemeinen sowie die Abgrenzung verschiedener Testarten. Anschließend wird in Abschnitt 6.2 ein Testkonzept für *Seco2015* aufgestellt.

Auf Basis des Konzeptes werden in Abschnitt 6.3 konkrete Komponenten- und Integrationstests entwickelt und ausgeführt. Die praktische Umsetzung der Tests endet mit dem in Abschnitt 6.4 vorgestellten Akzeptanztest.

Die Ergebnisse der Tests werden ferner in Abschnitt 6.5 zusammengefasst.

6.1 Grundlagen

Das Durchführen von Software-Tests hilft, die Qualität von Software zu steigern. Jedoch ist es nicht möglich ein Softwareprodukt vollständig zu testen. Ebenso beweist das fehlerfreie Ausführen von Tests nicht, dass die Software fehlerfrei ist. Mit Tests lässt sich lediglich die Anwesenheit von Fehlern und nicht deren Abwesenheit belegen.

6.1.1 Fundamentaler Testprozess

Der fundamentale Testprozess im Bereich der Softwareentwicklung wird nach [39] in folgende Phasen unterteilt:

- Planung: Definieren von Testzielen und einem Testkonzept
- Analyse: Überprüfen von Testbedingungen und Wahl von Werkzeugen
- Entwurf: Spezifizieren konkreter Testfälle
- Realisierung: Implementation der Tests

- Durchführung: Ausführen der Tests und Dokumentation der Resultate

In Abschnitt 6.2 wird das Testkonzept von *Seco2015* auf Basis dieser Phasen erstellt.

6.1.2 Arten von Tests

In der Praxis unterscheidet man anhand des Testumfangs bzw. der einbezogenen Teilsysteme folgende Testarten:

- Komponententest: Isolierte Tests an einzelnen Programmabschnitten
- Integrationstest: Testen des Zusammenspiels einzelner Komponenten
- Systemtest: Testen des Gesamtsystems durch den Entwickler
- Akzeptanztest: Testen des Gesamtsystems durch den Abnehmer

Beim Entwerfen konkreter Testfälle kann nach dem Verfahren „Black-Box“ oder „White-Box“ vorgegangen werden. „Black-Box“ Tests werden ohne Berücksichtigung des Quellcodes auf die reine Funktionalität ausgelegt. Sie helfen sicherzustellen, dass konkrete Anforderungen oder Funktionalitäten erfüllt sind. Bei „White-Box“ Tests werden konkrete Quellcode Konstrukte analysiert oder berücksichtigt. Sie helfen sicherzustellen, dass die Implementierung der Entwickler die erwarteten Resultate erzielt und liefern Aussagen über die Testabdeckung.

Die Testfälle für Komponten-, Integrations-, und Systemtests können sowohl mit dem „Black-Box“- als auch „White-Box“-Verfahren entwickelt werden. Ein Akzeptanztest dagegen sollte immer ein „Black-Box“ Test sein.

Test fälle lassen sich ferner in Negativ- und Positivtests unterteilen. Der Positivtest zielt auf die fehlerfreie Ausführbarkeit einer Funktionalität ab. Beim Negativtest werden dagegen bewusst Fehleingaben vorgenommen um zu prüfen, das diese korrekt abgefangen werden.

6.2 Testkonzept

Im Folgenden wird ein Konzept zur Sicherstellung der Softwarequalität entsprechend der Aufgabenstellung aufgestellt. Hierzu wird sich beim Vorgehen am fundamentalen Testprozess orientiert.

6.2.1 Planung

Das allgemeine Testziel ist es, die Zuverlässigkeit von *Seco2015* im Messbetrieb sicherzustellen. Konkret müssen hierzu folgende Teilsysteme validiert werden:

- Parser: Korrektes Einlesen der *Sequenzdatei* sowie Ausgabe von Syntaxfehlern
- Laufzeitumgebung: Generieren der Ereignisfolge durch Auflösen von Variablen, Aliasen und Makros sowie die Ausgabe von Fehlermeldungen
- Übersetzer: Korrekte Transformation der generischen Ereignisfolge in die Zielsprache
- Hardwarekommunikation: Fehlerfreie Übertragung und Ausführen einer Sequenz sowie Ausgabe von Fehlermeldungen

Innerhalb der Laufzeitumgebung müssen ferner die Komponenten zum von mathematischen Funktionen getestet werden. Die Hardwareansteuerung kann über die Anzeige des Sequenz-Generators oder durch Messung der TTL-Signale validiert werden.

6.2.2 Analyse

Die einzelnen Teilsysteme des Compilers stehen als Python-Klassen zur Verfügung. Zum Durchführen von Komponenten- und Integrationstests kann das Python-Modul `unittest` genutzt werden. Mit ihm lassen sich Testfälle definieren, bei denen ein erwarteter Ausgabewert einer Funktion mit ihrem tatsächlichen Ausgabewert verglichen wird.

Um das Gesamtsystem zu validieren wird ein Systemtest durchgeführt. Dieser muss sicherstellen, dass der Sequenz-Generator die erwarteten Schaltvorgänge durchführt. Hierzu wird die Ausgabe der LEDs genutzt, welche leuchten sobald ein Kanal angesteuert wird. Um das genaue Timing zu verifizieren, wird ferner ein Oszilloskop mit den Ausgängen des Sequenz-Generators verbunden. In beiden Fällen muss händisch überprüft werden, ob die mit der *Sequenzdatei* programmierte Synchronisierung der Ausgangsbeschaltung entspricht.

6.2.3 Entwurf

Die Spezifikation konkreter Testfälle sollte zielgerichtet sein und die verschiedenen Funktionen testen. Die Funktionalität der einzelnen Sprachbereiche kann separat und im Zusammenspiel

getestet werden. Aufgrund ihrer Relevanz im Messbetrieb liegt der besondere Augenmerk auf folgenden Funktionen:

- Definition von Variablen und Kanälen sowie das Anwenden von Menüeinträgen
- Offset- und wait-Parameter in Schleifen und Makros
- Mathematische Ausdrücke
- Schachtelung von Schleifen und Makros

Die Eingabedaten des Teilsytems Übersetzer sind im Vergleich zur *Sequenzdatei* sehr statisch und haben immer den gleichen Aufbau. Nachdem die korrekte Übersetzung einer generischen Ereignisfolgen validiert ist, werden durch das gezielte Platzieren technisch invalider Ereignisse oder Ereignisfolgen die Fehlererkennungsroutinen überprüft.

Die Hardwareansteuerung basiert ebenfalls auf einer statischen Form von Eingabedaten. Neben der zeitlich korrekten Ausführung einer Beispielsequenz wird mit Hilfe eines Oszilloskops das korrekte Einfügen von Dummy-Events sowie die Trennung von Teilsequenzen überprüft.

6.2.4 Realisierung

Die einzelnen Komponenten- und Integrationstests werden von der Hauptanwendung gekapselt in einem externem Modul implementiert, dass manuell ausgeführt werden muss. Es importiert alle Teilsysteme von *Seco2015* und hat somit Zugriff auf alle Teilfunktionen.

Das Framework `unittest` stellt Werkzeuge zur Durchführung der Tests bereit. Die einzelnen Tests werden jeweils in Klassen implementiert, die von `unittest.TestCase` erben. Das Framework führt automatisch alle Methoden eines solchen Testfalls aus, denen das Präfix `test_` vorangestellt ist. In diesen Methoden werden Varianten des `assert`-Statements genutzt um den Vergleich zwischen Soll- und Ist-Ausgabe einer bestimmten Komponente zu testen. Im Rahmen von *Seco2015* werden folgende dieser Statements genutzt:

- `assertEqual`: Überprüfung ob zwei Variablen den gleichen Inhalt aufweisen
- `assertIn`: Überprüfung ob eine Wert in einer Liste enthalten ist
- `assertRaises:` Überprüfung ob ein konkreter Ausnahmefehler ausgelöst wird

Um die konkreten Tests ausführen zu können, kann es vorher erforderlich sein, verschiedene Aufgaben auszuführen. Dies kann zum Beispiel das Generieren von Instanzen oder die Ausführung von Initialisierungsroutinen sein. Die entsprechenden Aufgaben können in der Klasse des Testfalls in der Methode `setUp()` implementiert werden. Das Framework führt diese automatisch aus, bevor es die einzelnen Tests abarbeitet.

6.2.5 Durchführung

Die entsprechenden Integrationstests werden parallel zu neuer Funktionalität entwickelt und können somit zum Erkennen von Regressionen genutzt werden. Die kontinuierliche Entwicklung von Tests stellt sicher, dass viele Ausführungspfade der Software validiert werden. Durch die Erstellung von Metriken, wie der „Code-Coverage“ können Aussagen über die Testabdeckung getroffen werden. Hierfür steht in Python das Skript `coverage.py` zur Verfügung. Es erstellt eine Statistik über die Anweisungsabdeckung von Tests innerhalb eines Projektes.

Das Ausführen der mit Hilfe von `unittest` entwickelten Komponententests findet manuell statt. Der Entwickler ist damit selbst für die regelmäßige Durchführung von Tests verantwortlich. Bei der Entwicklungsumgebung muss hierzu bei den Build-Einstellungen des Testmoduls festgelegt werden, dass es sich um ein solches handelt. Das Framework stellt zu jedem Test eine kurze Ausgabe bereit, die den Status und die Laufzeit der entsprechenden Tests umfasst. Schlägt ein Test fehl, wird dies zusätzlich in der Zusammenfassung am Ende der Testdurchläufe hervorgehoben.

Der Akzeptanztest wird am Ende der Entwicklung zur Validierung des Gesamtsystems durchgeführt. Mit ihm kann insbesondere die Korrektheit der Steuerbefehle verifiziert werden.

6.3 Umsetzung der Komponenten- und Integrationstests

In den folgenden Abschnitten ist die konkrete Umsetzung der einzelnen Komponenten- und Integrationstests beschrieben. Außerdem werden alle Tests zum Ende der Entwicklungen an *Seco2015* durchgeführt und das entsprechende Ergebnis dokumentiert.

6.3.1 Definition von Variablen, Aliasen und Makros

Die Definition von Variablen, Aliase und Makros werden durch das Vergleichen der entsprechenden Symboltabellen überprüft. Da die Redundanz zwischen den Testfällen hoch ist, wird eine abstrakte Klasse `AbstractDefinitionTest` geschaffen. Sie enthält die Instanz des Parsers sowie eine valide *Sequenzdatei*, in der verschiedene Definitionen vorgenommen werden. Die Testfälle für Variablen, Makros und Aliase implementieren die abstrakte Klasse und erzeugen über die Laufzeitumgebung die entsprechenden Symboltabellen. Testkriterium des Positivtests ist das Vergleichen der Symboltabelle mit den erwarteten Definitionen.

In Tabelle 6 ist die Durchführung der zugehörigen Testfälle protokolliert.

Testfall	**überprüfte Funktionalität**	**bestanden**
D1	Definition von Variablen	ja
D2	Definition von Kanälen	ja
D3	Definition von Makros	ja

Tabelle 6: Testfälle zu Definitionen in der *Sequenzdatei*

6.3.2 Menüeinträge

Das Aktivieren von Menüeinträgen hat das Deaktivieren von Kanälen und das Verändern von Variablenwerten zur Folge. Die Testfälle werden ebenfalls auf Basis der abstrakten Klasse `AbstractDefinitionTest` implementiert. Die Laufzeitumgebung aktiviert den mit `default` gekennzeichneten Menüeintrag automatisch. Ist kein `default` angegeben, wird eine Warnung erzeugt und die Standardeinstellungen beibehalten. Die Anzeige der Warnung wird mit dem Negativtest M1 geprüft. Testkriterium ist das Auslösen eines Ausnahmefehlers. Die Laufzeitumgebung erstellt eine Liste mit deaktivierten Kanälen, die bei der Erstellung der Ereignisfolge berücksichtigt wird. Das Zusammenspiel der Komponenten wird mit einem Positivtest validiert, bei dem die erzeugte generische Ereignisliste mit dem erwarteten Ergebnis verglichen wird. Für die Überprüfung der Anpassung der Variablen wird eine konkrete Variable vor und nach der Aktivierung des Eintrages ausgelesen. In Tabelle 7 ist die Durchführung der entsprechenden Tests protokolliert.

Testfall	überprüfte Funktionalität	bestanden
M1	Warnung bei Menü-Definition ohne `default`	ja
M2	Verändern von Variablenwerten	ja
M3	Deaktivieren von Kanälen	ja

Tabelle 7: Testfälle zur Menü-Definition in der *Sequenzdatei*

6.3.3 Definition von Sequenzen

Da die Testfälle zur Sequenz-Generierung ebenfalls Redundanzen aufweisen, wird eine abstrakten Klasse `AbstractEventTest` geschaffen. Diese instanziiert den Parser und enthält eine valide *Sequenzdatei*, in der die Bereiche *variables*, *channels* und *menu* mit validen Definitionen ausgefüllt sind. Die konkreten Testfälle erben von dieser Klasse und beschreiben die Bereiche der Makro- und Sequenz-Definitionen. Testkriterium für alle `AbstractEventTests` ist die generische Ereignisfolge. Eingabewert für die Durchführung dieser Positivtests ist eine konkrete Sequenz- oder Makrodefinition, in der die zu testende Funktionalität modelliert ist.

In Tabelle 8 ist die Durchführung der Test zur Definition von Sequenzen protokolliert.

Testfall	überprüfte Funktionalität	bestanden
E1	Definition einer Ereignisfolge	ja
E2	Definition mehrerer Sequenzen	ja
E3	Verwendung des wait-Parameters	ja
E4	Verwendung des offset-Parameters	ja
E5	Verwendung von wait- und offset-Parameter	ja

Tabelle 8: Testfälle zur Definition von Sequenzen in der *Sequenzdatei*

Eine Spracherweiterung von *Seco2015* sind die Makros. Bei den Tests muss insbesondere auf das Einfügen von Makros sowie auf ein korrektes Verhalten bei Schachtelungen geachtet werden. Die verschiedenen Möglichkeiten, die wait- und offset-Parameter zu verwenden, müssen ferner stets das erwartete Ergebnis liefern. Im Testfall E9 wird mit einem Negativtest validiert, dass zirkuläre Makroaufrufe eine Fehlermeldung auslösen. Testkriterium hierfür ist das Auslösen eines Ausnahmefehlers. Bei den anderen Testfällen handelt es sich um Positivtests, die ebenfalls auf den `AbstractDefinitionTests` aufbauen. Die Ergebnisse

sind in Tabelle 9 protokolliert.

Testfall	überprüfte Funktionalität	bestanden
E6	Definition eines Makro	ja
E7	Definition mehrere Markos	ja
E8	Schachtelung von Makros	ja
E9	Abfangen zirkulärer Makroaufrufe	nein
E10	Verwendung des offset-Parameters	ja
E11	Verwendung des wait-Parameters	ja
E12	Verwendung von wait- und offset-Parameter	ja
E13	Verwendung von offset beim Einfügen eines Makros	ja
E14	Verwendung von wait beim Einfügen eines Makros	ja
E15	Verwendung von offset und wait beim Aufruf eines Makros	ja

Tabelle 9: Testfälle zur Definition von Makros in der *Sequenzdatei*

Die Ausführung der Tests ergibt, dass bei einem zirkulärer Makroaufruf (Testfall E9) keine Fehlermeldung vom Compiler ausgegeben wird. Bei der Erzeugung der generischen Ereignisfolge entsteht eine wechselseitige Rekursion, die schließlich zum Absturz des Compilers führt. Dieser Fehler ist jedoch nicht sicherheitskritisch und kann vom Nutzer einfach erkannt und behoben werden.

Eine weitere Neuerung in *Seco2015* ist die Möglichkeit, bei der Makro- und Sequenz-Definition Schleifen nutzen zu können. Auch hier muss insbesondere ein korrektes Verhalten bei beliebiger Schachtelung und dem Einsatz der wait- und offset-Parameter validiert werden. Die Positivtests basieren ebenfalls auf den `AbstractDefinitionTests`. Die Durchführung der Tests zur Definition von For-Schleifen sind in Tabelle 10 protokolliert.

Um dem Nutzer beim Beheben von Fehlern zu Unterstützen, existieren eine Reihe von Ausnahmefehlern, deren korrektes Auslösen Bestandteil der Komponententests sein soll. Hierfür werden Negativtests definiert, deren Testkriterium das Auslösen von detaillierten Ausnahmefehlern sind. Die Durchführung der Tests zum Auslösen der korrekten Fehlermeldung sind in Tabelle 11 protokolliert.

Testfall	überprüfte Funktionalität	bestanden
E16	Definition einer For-Schleife	ja
E17	Definition einer For-Schleife mit wait-Parameter	ja
E18	Definition einer For-Schleife mit offset-Parameter	ja
E19	Definition einer For-Schleife mit wait- und offset-Parameter	ja
E20	Schachtelung von For-Schleifen	ja
E21	Schachtelung von Makros und For-Schleifen	ja

Tabelle 10: Testfälle zur Definition von For-Schleifen in der *Sequenzdatei*

Testfall	überprüfte Funktionaliät	bestanden
E22	Fehlermeldung bei Verwendung nicht definierter Variable	ja
E23	Fehlermeldung bei Verwendung nicht definierter Alias	ja
E24	Fehlermeldung bei Verwendung nicht definiertes Makro	ja

Tabelle 11: Testfälle zum Überprüfen von Ausnahmefehlern bei Verwendung von zuvor nicht definierten Bezeichnern

6.3.4 Berechnung mathematischer Ausdrücke

Da für die Angabe von Schaltzeitpunkten, Impulsdauern, Menüeinträgen, Einstellungen sowie der wait- und offset-Parameter mathematische Ausdrücke verwendet werden können, ist die Korrektheit einer generierten Sequenz von deren korrekter Berechnung abhängig. Ein Integrationstest von Parser und den `IValue`-Klassen des Objektmodells validiert das korrekte Einlesen und Berechnen von mathematischen Ausdrücken.

Bei diesem Test wird ein „White-Box“ Ansatz verfolgt. Aus diesem Grund ist bekannt, dass mathematische Ausdrücke in der Grammatik allgemein definiert sind. Daher ist es für den Integrationstest nicht relevant, an welcher Stelle der *Sequenzdatei* die Module getestet werden. Es wird eine abstrakte Klasse `AbstractMathTest` definiert, welche eine Beispiel-*Sequenzdatei* enthält. Sie enthält eine Sequenz mit einem Ereignis, dessen Ausführungszeitpunkt von den einzelnen Testfällen über den Konstruktor der abstrakten Klasse angepasst werden kann.

In der Praxis werden mathematische Ausdrücke zur Abbildung von linearen Zusammenhängen oder Phasenversätzen verwendet. Aus diesem Grund liegt der Fokus der Tests zunächst auf den Grundrechenoperationen sowie deren Schachtelung. Testkriterium ist der Wert, der bei Auflösung eines mathematischen Ausdrucks von der Funktion `get_value() zurückgegeben`

wird. In Tabelle 12 ist die Durchführung der Integrationstests der Mathematik-Module und des Parsers protokolliert. Es handelt sich dabei um Positivtests.

Testfall	überprüfte Funktionalität	bestanden
F1	Grundrechenarten	ja
F2	Verwendung von negativen Zahlen	ja
F3	Verwendung von Gleitkommazahlen	ja
F4	Potenzierung	ja
F5	Mehrfachausführung von Potenzierung	nein
F6	Verwendung von E für Zehnerpotenzen	ja
F7	Schachtelung von Termen mit Klammern	ja
F8	Schachtelung von Termen ohne Klammern	nein
F9	Verwendung von Variablen	ja

Tabelle 12: Testfälle zum Überprüfen der Berechnung mathematischer Ausdrücke

Bei der Ausführung der Tests sind Fehler bei der Mehrfachausführung einer Potenzierung (Testfall F5) sowie bei geschachtelten Ausdrücken ohne Klammern (Testfall F8) aufgetreten. Die Ursache hierfür liegt bei einer inkorrekten Umsetzung der Potenzregeln. Der Fehler kann jedoch leicht durch die Anpassung der Klasse `Factor` behoben werden.

6.3.5 Kompilieren einer Sequenz

Im Rahmen der Tests werden die Teilsysteme Optimierer und Übersetzer unter dem Begriff Compiler zusammengefasst. Der Compiler erhält als Eingabe die generische Ereignisfolge und generiert daraus eine Datenstruktur, die der Bibliothek übergeben wird. Um die Redundanzen der Testfälle zu minimieren, wird eine abstrakte Klasse `AbstractCompilerTest` geschaffen, die den Compiler instanziiert. Sie enthält ferner einen Testprototypen, welcher eine generische Ereignisfolge kompiliert und mit dem erwarteten Ergebnis vergleicht. Die von der abstrakten Klasse erbenden Testfälle führen den Testprototyp aus. Die Eingabedaten können dem Konstruktor übergeben werden. Testkriterium der Positivtests C1, C2, C3, C5, C6 und C7 ist der Code in Zielsprache. Für den Testfall C4 wird eine Sequenz mit einer zu niedrigen Auflösung als Eingabe verwendet. Testkriterium für diesen Negativtest ist das Auslösen eines detaillierten Ausnahmefehlers.

Das Interpretieren und Umsetzen von Einstellungen wird ebenfalls vom Teilsystem Compiler vorgenommen. Die Tests hierfür sind von den `AbstractDefinitionTests` abgeleitet. Aus diesem Grund werden Komponententests geschaffen, die das korrekte Interpretieren der Einstellungen in den Positivtests C6 und C7 validieren. Testkriterium der Negativtests C8 und C9 ist das Auslösen des korrekten Ausnahmefehlers. Als Eingabedaten werden invaliden Einstellungen gewählt.

In Tabelle 13 ist die Durchführung der Tests zum Kompilieren einer Sequenz aufgeführt.

Testfall	**überprüfte Funktionalität**	**bestanden**
C1	Kompilieren einer einfachen Ereignisfolge	ja
C2	Zusammenfassen von Ereignissen	ja
C3	Berechnung der Auflösung	ja
C4	Ausnahmefehler bei zu geringer Auflösung	ja
C5	Kompilieren mehreren Sequenzen	ja
C6	Definition von Einstellungen auf Basis von Zahlenwerten	ja
C7	Definition von Einstellungen auf Basis von Schlüsselworten	ja
C8	Ausnahmefehler bei der Anwendung unbekannter Einstellungen	ja
C9	Ausnahmefehler bei der Anwendung invalider Einstellungsparameter	ja

Tabelle 13: Testfälle zum Validieren des Compilers

6.4 Umsetzung Akzeptanztest

Der Akzeptanztest soll abschließend die Funktionsfähigkeit der Gesamtanwendung validieren. Die generierte Sequenz wird hierzu auf Hardwareebene nachvollzogen werden. Um die generierten Schaltfunktionen erfassen zu können, werden zwei Kanäle des Sequenz-Generators mit einem digitalen Oszilloskop verbunden. Der Testaufbau ist in Abbildung 19 dargestellt.

Die auszuführende *Sequenzdatei* ist in Quellcode 13 angeben. In ihr sind zwei Makros definiert, die jeweils einen der beiden Kanäle auslösen. Bei der Definition werden Aliase, Variablen und einfache mathematische Ausdrücke verwendet sowie ein Menüeintrag aktiviert. Die Hauptsequenz führt die beiden Makros in einem Abstand von 250 μs aus. Der Trigger des Oszilloskops wird auf Kanal 1 gestellt.

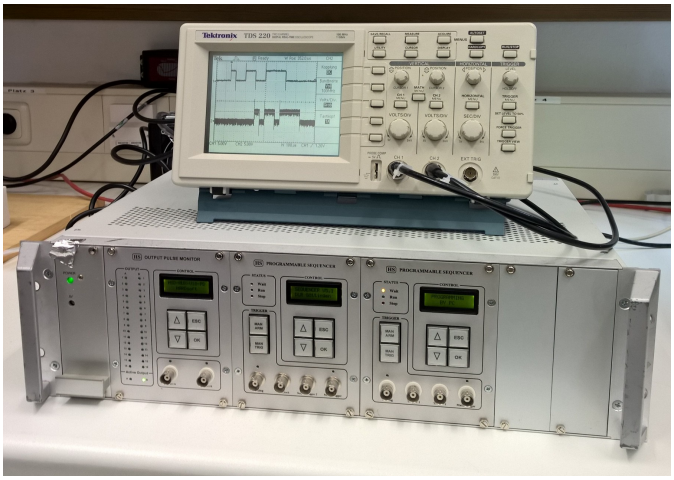
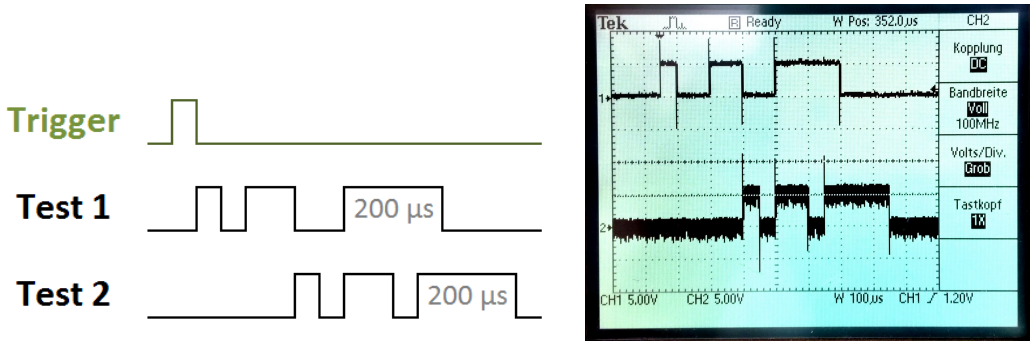


Abbildung 19: Testaufbau für den Akzeptanztest

Durch ein manuelles Auslösen des Triggers wird die Sequenz ausgeführt. In Abbildung 20 ist die erwartete Ausgabe, der Anzeige des Oszilloskops gegenüber gestellt. Auf dieser Basis wird entschieden, dass *Seco2015* den Akzeptanztest bestanden hat.



(a) Erwartete Ausgabe des Oszilloskops nach Ausführung des Akzeptanztests

(b) Ausgabe des Oszilloskop nach Ausführung des Akzeptanztests

Abbildung 20: Vergleichsparameter des Akzeptanztests

```
variables {
  delay, "standard-delay" 1,  100,  1000;
  h,   "High-time",       1,    50,     100;
  }
channels {
  1, c1,    "Test1";
  2, c2,    "Test2";
  }
settings {
  "workingmode" = "continue";
  }
menus {
  default   "standard_delay", delay = 50;
  }
macro canal1{
  1,  delay, h;
  1,  delay+150, h*2;
  1,  delay+350, h*4;
  }
macro canal2{
  2, 50, h;
  2, 150, h*2;
  2, 300, h*4;
  }
sequence 1 {
  canal1;
  canal2 (offset = 250);
  }
```

Quellcode 13: *Sequenzdatei* zum Durchführen des Akzeptanztests

6.5 Ergebnisse

Zur Sicherstellung der Softwarequalität sind Tests Bestandteil der Anwendung *Seco2015*. Die Komponenten- und Integrationstests werden mit Hilfe des Python-Frameworks `unittest` umgesetzt.

Es wird eine Reihe an Testfällen konstruiert, mit denen die Teilsysteme von *Seco2015* unabhängig voneinander und im Zusammenspiel getestet werden. Dies umfasst insbesondere folgende Funktionen:

- Definition von Variablen, Aliasen und Makros
- Menü-Definitionen

- Sequenz-Definitionen
- Struturierungsmechanismen Makro und For-Schleife
- Korrektes Auslösen von Ausnahmefehlern
- Berechnung von mathematischen Ausdrücken
- Optimierung und Übersetzung

Zur Bewertung der Testabdeckung wird das Skript `coverage.py` ausgeführt. Die ermittelte Statement-Coverage[1] bei der Ausführung von Testfällen liegt in *Seco2015* insgesamt bei 88%. Die Code-Abdeckung für die einzelnen Module der Anwendung ist in Tabelle 14 aufgeschlüsselt.

Python-Modul	**Teilsystem/Funktion**	**Abdeckung**
events	Auflösung der Ereignisse	98%
seqParser	Parser und Laufzeitumgebung	92%
sequencer	Optimierer, Schnittstelle zur Hardwareansteuerung	75%
sequencerFactory	Verbinden eines Sequenz-Generators	100%
sequencerV5	Übersetzer und Hardwareansteuerung	61%
values	Auflösen mathematischer Ausdrücke	95%

Tabelle 14: Codeabdeckung bei Ausführung der Komponenten- und Integrationstests

Die verhältnismäßig geringe Ausführungsabdeckung bei den Modulen der Hardwareansteuerung lässt sich damit erklären, dass für das Ausführen einer einfachen Sequenz nur die Grundfunktionen der Bibliothek angesprochen werden. Tests für das Aktivieren von Einstellungen sind beispielsweise nicht implementiert.

Auf dem finalen Entwicklungsstand konnte eine Vielzahl der Tests erfolgreich absolviert werden. Aus den Protokollen in Abschnitt 6.3 geht hervor, dass Fehler bei zirkulären Makroaufrufen, sowie dem Definieren komplexer Terme ohne die Verwendung von Klammern auftreten. Diese Fehler schränken den Compiler in der Praxisanwendung jedoch nicht ein und können in Zukunft behoben werden. Im Vergleich zu *Seco2004* wird eine Qualitätssteigerung erreicht, da das Thema Softwarequalität in Form von umfangreichen Tests bei der Entwicklung von *Seco2015* adressiert wird.

[1] Anteil ausgeführter Anweisungen

7 Fazit

Dieses Kapitel zieht ein Fazit über die Lösung der Aufgabenstellung dieser Bachelorarbeit Hierzu wird zunächst in Abschnitt 7.1 eine Zusammenfassung der praktischen Ergebnisse gegeben. In Abschnitt 7.2 wird die Erfüllung der Kernanforderungen evaluiert. Ausgehend vom aktuellen Entwicklungsstand werden in Abschnitt 7.3 mögliche Weiterentwicklungen vorgeschlagen.

7.1 Zusammenfassung

Im Rahmen dieser Bachelorarbeit wurde ein Compiler zur Programmierung von Sequenz-Generatoren entwickelt. Er erfüllt die Anforderungen an die Echtzeitsynchronisierung von Messkomponenten und behebt eine Reihe von Problemen, die im Zusammenhang mit der zur Zeit verwendeten Software *Seco2004* aufgetreten sind.

Da das Softwaredesign der alten Anwendung nur begrenzt modernem Software-Engineering entspricht, wurde die Entscheidung getroffen, eine komplett neue Anwendung zu entwickeln. Der Ausgangspunkt für die Entwicklung stellt die Repräsentation der Ablaufsteuerung dar. Hierzu wurde der aktuelle Stand der Technik analysiert und verschiedene Ansätze zur Modellierung von Ereignisfolgen miteinander verglichen. Auf dieser Basis wurde die Entscheidung getroffen, eine textbasierte domänenspezifische Sprache zur Konfiguration der Sequenz-Generatoren zu entwerfen.

Die Analyse der mit der Sprache abzubildenen Problemdomäne ergab, dass die Sprachsyntax in *Seco2004* als Basis für die neue Sprache genutzt werden kann. Auf Basis von Praxiserfahrungen von Anwendern konnte eine zielgerichtete Erweiterung des Funktionsumfangs erreicht werden. Beim Design der Sprache lag vor allem die syntaktische Nähe zur alten Konfigurationssprache sowie anderen GPLs im Fokus.

Um eine wartungsfreundliche und erweiterbare Architektur zu konzeptionieren, wurden die einzelnen Komponenten von *Seco2015* isoliert betrachtet. Die neue Sprachsyntax wurde formal in einer Grammatik definiert. Auf Basis der Grammatik wird ein Parser generiert, sodass zum einen die Konsistenz der Sprache, aber auch die Zuverlässigkeit des Parsers

erhöht wird. Der Parser generiert ferner ein an die Grammatik angelehntes Objektmodell. Das Objektmodell wird von einer Laufzeitumgebung verwaltet, welche die abstrakte Repräsentation der Sequenzen in eine generische Ereignisfolge umwandelt.

Die generische Ereignisfolge ist unabhängig vom verwendeten Sequenz-Generator und wird anschließend vom Übersetzer optimiert und in die Zielsprache transformiert. Zur Demonstration der Funktionsfähigkeit wurde die Hardwareansteuerung für einen Hardsoft Sequenz-Generator v5.1 implementiert.

Die Qualität der Software wurde schließlich durch Komponenten-, Integrations- und Akzeptanztests sichergestellt.

7.2 Evaluation

Im Folgenden wird evaluiert, ob die Lösung der Aufgabenstellung den gestellten Anforderungen genügt.

Kernanforderung der Umsetzung war es, einen zuverlässigen und leicht erweiterbaren Compiler zu entwickeln. Durch die Verwendung eines Parsergenerators und das Durchführen von Softwaretests können viele Programmabläufe validiert werden. Die Architektur ist konsequent auf einfache Erweiterungen ausgelegt und es existieren allgemeine Schnittstellen für die Hardwaresteuerung. Zwischen den einzelnen Teilsystemen bestehen wenig Abhängigkeiten, sodass sie einfach ausgetauscht werden können. Die erste Kernanforderung ist somit erfüllt.

Eine weitere Kernanforderung war es, die Entwicklung einer *Sequenzdatei* zu verbessern. Eine konsistente Sprachdefinition bildet die Grundlage hierfür. Die neue Sprachsyntax ist durch die Unterstützung von Makros und Schleifen übersichtlicher und besser an die Problemdomäne der Ablaufsteuerung angepasst. Die technische Validierung einer programmierten Sequenz stellt in verschiedenen Bereichen sicher, dass keine Fehlansteuerung möglich ist. Die zweite Kernanforderung konnte somit erfüllt werden.

7.3 Ausblick

Seco2015 demonstriert die Funktionsfähigkeit des entwickelten Compilers. Damit wird der Grundstein für weitere Entwicklungen und das Ersetzen von *Seco2004* in der Abteilung gelegt.

Eine dafür notwendige Erweiterung ist das Entwickeln einer grafischen Benutzeroberfläche. Diese sollte dem Anwender zur Laufzeit die Möglichkeiten bieten, Parameter anzupassen und die Ausführung von Sequenzen zu koordinieren. Ein weiterer Schritt wäre es, eine Entwicklungsumgebung für *Sequenzdateien* umzusetzen. Sie könnte den Anwender bei der Erstellung von *Sequenzdateien* durch Syntaxhervorhebung, automatische Vervollständigung und Syntaxvalidierung während des Editierens unterstützen.

Die Umsetzung von *Seco2015* ist auf einfache Erweiterbarkeit ausgelegt. Eine denkbare Erweiterung wäre die Unterstützung verschiedener Sequenz-Generator Hardwarerevisionen. In der Abteilung werden zum Teil auch Funktionsgeneratoren von NI zur Synchronisation genutzt. In diesem Umfeld bietet es sich an, den Compiler auch für andere Zielplattformen zu erweitern.

Um die Software-Qualität auch für zukünftige Entwicklungen sicherzustellen, können die Tests automatisiert werden. Die Ausführung alle Komponenten- und Integrationstests könnte automatisch bei Buildvorgängen oder Commits initiiert werden.

Literaturverzeichnis

[1] Stasicki, B. ; Meier, G. E. A.: Computer-controlled Ultra-High-Speed Video Camera System. In: *Proc. SPIE* 2513 (1995), S. 196–208. http://dx.doi.org/10.1117/12.209602. – DOI 10.1117/12.209602. – http://dx.doi.org/10.1117/12.209602, abgerufen am 17.08.2015

[2] Reinke, R.: *Neuentwicklung eines Programmpaketes zur Ansteuerung eines PIV Aufnahmesystems mit Hilfe eines Sequenzers – Konzeption, Entwicklung und Funktionstest*, Berufsakademie Mannheim, Diplomarbeit, 2003

[3] Raffel, M. ; Willert, Ch. E. ; wereley, s. ; Kompenhans, J.: *Particle Image Velocimetry - A Practical Guide*. 1st ed. 1998. Corr. 2nd printing. Berlin, Heidelberg : Springer, 1998. – ISBN 978–3–540–63683–0

[4] Kompenhans, J. ; Raffel, M. ; Dieterle, L. ; Dewhirst, T. ; Vollmers, H. ; Ehrenfried, K. ; Willert, C. ; Pengel, K. ; Kähler, C. ; Schröder, A. ; Ronneberger, O.: Particle Image Velocimetry in Aerodynamics: Technology and applications in wind tunnels. In: *Journal of Visualization* 2 (2000), Nr. 3-4, S. 229–244. http://dx.doi.org/10.1007/BF03181440. – DOI 10.1007/BF03181440. – ISSN 1343–8875. – http://dx.doi.org/10.1007/BF03181440, abgerufen am 17.08.2015

[5] Stasicki, B. ; Ehrenfried, K. ; Dieterle, L. ; Ludwikowski, K. ; Raffel, M.: Advanced synchronization techniques for complex flow field. In: *4th International Symposium on Particle Image Velocimetry* (2001). – http://velocimetry.net/downloads/pi6_1188.pdf, abgerufen am 17.08.2015

[6] TSI: *LASERPULSE SYNCHRONIZER MODEL 610036*, 2014. – http://www.tsi.com/uploadedFiles/_Site_Root/Products/Literature/Spec_Sheets/Model%20610036_US_5001385_WEB.pdf, abgerufen am 17.08.2015

[7] Taylor, Z.J. ; Gurka, R. ; Kopp, G.A. ; Liberzon, A.: Long-Duration Time-Resolved PIV to Study Unsteady Aerodynamics. In: *Instrumentation and Measurement, IEEE Transactions on* 59 (2010), Dec, Nr. 12, S. 3262–3269. – http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5464317&tag=1, abgerufen am 17.08.2015

[8] National Instruments: *Tools for Digital and Analog Modulation/Demodulation Communications Analysis*. – http://sine.ni.com/nips/cds/pages/image?imagepath=/images/products/us/pxie-5450_l.jpg&title=NI%20PXIe-5450&

oracleLang=d, abgerufen am 17.08.2015

[9] National Instruments: *Systemdesignsoftware LabVIEW*. – http://www.ni.com/labview/d/, abgerufen am 17.08.2015

[10] National Instruments: *Signalgeneratoren*. – http://www.ni.com/signalgenerators/d/, abgerufen am 17.08.2015

[11] National Instruments: *Modulation Toolkit - Modulated Signal Processing for LabVIEW*. – http://sine.ni.com/nips/cds/view/p/lang/de/nid/210568, abgerufen am 17.08.2015

[12] National Instruments: *Tools for Digital and Analog Modulation/Demodulation Communications Analysis*. – http://www.ni.com/pdf/products/us/032753305101.pdf, abgerufen am 17.08.2015

[13] Object Management Group: *OMG Unified Modeling Language Superstructure v2.4.1*. 2011. – http://www.omg.org/spec/UML/2.4.1/, abgerufen am 10.09.2015

[14] Graf, S. ; Ober, I. ; Ober, I.: A Real-time Profile for UML. In: *International Journal on Software Tools for Technology Transfer* (2006), S. 113–127. – ISSN 1433–2779. – http://link.springer.com/article/10.1007/s10009-005-0213-x, abgerufen am 17.08.2015

[15] Bailey, J. ; Papamarkos, G. ; Poulovassilis, A. ; Wood, P.T.: An Event-Condition-Action Language for XML. In: *Web Dynamics*. Springer Berlin Heidelberg, 2004. – ISBN 978–3–642–07377–9, S. 223–248. – http://link.springer.com/chapter/10.1007/978-3-662-10874-1_10, abgerufen am 17.08.2015

[16] Arpaia, P. ; Fiscarelli, L. ; La Commara, G. ; Romano, F.: A Petri Net-Based Software Synchronizer for Automatic Measurement Systems. In: *Instrumentation and Measurement, IEEE Transactions on* 60 (2011), Jan, Nr. 1, S. 319–328. – http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5613935, abgerufen am 17.08.2015

[17] Mernik, M. ; Heering, J. ; Sloane, A.: When and How to Develop Domain-Specific Languages. In: *ACM Computing Surveys* 37 (2005), S. 316–344. – http://people.cis.ksu.edu/~schmidt/505f10/WhenDSL.pdf, abgerufen am 17.08.2015

[18] Arpaia, P. ; Fiscarelli, L. ; La Commara, G. ; Petrone, C.: A Model-Driven Domain-Specific Scripting Language for Measurement-System Frameworks. In: *Instrumentation and Measurement, IEEE Transactions on* (2011), S. 3756–3766. –

http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5942162 , abgerufen am 17.08.2015

[19] Deshmukh, M. ; Weps, B. ; Isidro, P. ; Gerndt, A.: Model Driven Language Framework to Automate Command and Data Handling Code Generation. In: *Aerospace Conference, 2015 IEEE*, 2015, S. 1–9. – http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7118991, abgerufen am 17.08.2015

[20] Ghosal, A. ; Henzinger, T. A. ; Kirsch, C. M. ; Sanvido, M.: Event-Driven Programming with Logical Execution Times. In: *Hybrid Systems: Computation and Control* Bd. 2993. Springer Berlin Heidelberg, 2004. – ISBN 978–3–540–21259–1, S. 357–371. – http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.93.9923&rep=rep1&type=pdf, abgerufen am 17.08.2015

[21] *Defect Prevention and Detection in Software for Automated Test Equipment*. – =http://www.osti.gov/scitech/servlets/purl/895743, abgerufen am 17.08.2015

[22] Hedtstück, U.: *Einführung in die Theoretische Informatik - Formale Sprachen und Automatentheorie*. überarbeitete Auflage. München : Oldenbourg Verlag, 2012. – ISBN 978–3–486–71404–3

[23] Aho, A.: *Compilers : Principles, Techniques and Tools*. Boston : Pearson/Addison Wesley, 2007. – ISBN 0–321–48681–1

[24] Gueting, R.: *Uebersetzerbau : Techniken, Werkzeuge, Anwendungen*. Berlin Heidelberg New York Barcelona Hongkong London Mailand Paris Singapur Tokio : Springer, 1999. – ISBN 3540653899

[25] German National Research Center for Information Technology ; Fraunhofer Institute for Computer Architecture and Software Technology: *The Catalog of Compiler Construction Tools - Freeware and Commercial Rescources for Compiler Writers*. – http://catalog.compilertools.net/, abgerufen am 17.08.2015

[26] Parr, T. J. ; Quong, R. W.: ANTLR: A Predicated-LL(k) Parser Generator. In: *Software: Practice and Experience* 25 (1995), Nr. 7, S. 789–810

[27] Eclipse XTEXT: *What is XTEXT?*. – https://www.eclipse.org/Xtext/, abgerufen am 30.08.2015

[28] Oracle Corporation: *Java Compiler Compiler tm (JavaCC tm) - The Java Parser Generator*. – https://javacc.java.net/, abgerufen am 30.08.2015

[29] Lesk, M. E. ; Schmidt, E.: Lex - A Lexical Analyzer Generator. (1975). – http://ken-cc.googlecode.com/svn/trunk/doc/lex.pdf, abgerufen am 30.08.2015

[30] Johnson, S. C.: Yacc: Yet Another Compiler-Compiler. (1975). – http://ken-cc.googlecode.com/svn/trunk/doc/yacc.pdf, abgerufen am 30.08.2015

[31] Flex Project: *flex: The Fast Lexical Analyzer*. 2008. – http://flex.sourceforge.net/, abgerufen am 30.08.2015

[32] Free Software Foundation: *Introduction to Bison*. 2014. – https://www.gnu.org/software/bison/, abgerufen am 30.08.2015

[33] Beazley, D.: *PLY (Python Lex-Yacc)*. – http://www.dabeaz.com/ply/, abgerufen am 30.08.2015

[34] LLC, Tangient: *PyParsing - Introduction*. – http://pyparsing.wikispaces.com/Introduction, abgerufen am 30.08.2015

[35] Wile, D.: Lessons learned from real DSL experiments. In: *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on*, 2003

[36] Balzert, H.: *Lehrbuch Der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb* -. 3. Aufl. Berlin Heidelberg New York : Springer-Verlag, 2011. – ISBN 978–3–827–42246–0

[37] Gamma, E. ; Helm, R. ; Johnson, R. ; Vlissides, J.: *Design patterns : Elements of Reusable Object-Oriented software*. Reading, Mass : Addison-Wesley, 1995. – ISBN 0–201–63361–2

[38] Hills, M. ; Klint, P. ; Storm, T. van d. ; Vinju, J.: A Case of Visitor versus Interpreter Pattern. In: *Objects, Models, Components, Patterns* Bd. 6705. Springer Berlin Heidelberg, 2011. – ISBN 978–3–642–21951–1, S. 228–243. – http://homepages.cwi.nl/~storm/publications/visitor.pdf, abgerufen am 01.09.2015

[39] International Software Testing Qualifications Board: *Certified Tester - Foundation Level Syllabus*. 2011. – http://www.istqb.org/downloads/finish/16/15.html, abgerufen am 07.09.2015

[40] Foster, E.: *Software engineering : a methodical approach*. New York : Apress, 2014. – ISBN 978–1–4842–0848–9

Anhang

A Gewünschte Funktionalität anhand einer Beispielsequenz

```
variables {
  a,   "Tau,[us]",          4,     80,    1000;
  qs1, "FL-QS delay, 1",  220,    360,     499;
  qs2, "FL-QS delay, 2",  220,    360,     499;
  cd,  "Camera delay",   9937,   9956,    9977;
  f,   "Laser Period",  99000, 100000,  105000;
  h,   "High-time",       100,    100,     100;
  n,   "Frames",            1,    200,    2000;
}

settings {
  ArmingSlope   = "rising"; # case-insensitiv
  Resolution    = 5; # ns
  Terminator    = "true";
  TriggerSlope  = "falling";
  Active        = "high";
  TriggerSource = "Generator";
  WorkingMode   = "Alternative";
  Generator     = 1e6/f; # Anpassung mit Variable
}

channels {
  17,        "Flashlamps 1";
  18, qs1,   "Q-switch 1"; # optionaler Alias. Eigener Namespace
  19,        "Flashlamps 2";
  20, qs2,   "Q-switch 2";
  21,        "Camera";
  22, start, "Started";
  23, stop,  "Stopped";
}

menus {
      "High Energy",      qs1=220, qs2=220;
      "Low Energy",       qs1=360, qs2=qs1; # Formel in Assignment
default "Flash-lamps only", qs1=500, qs2=500, qs1, qs2; # Default-
    Einstellung
      "Laser off",        qs1=500, qs2=500, qs1, qs2, 17, 19; # Channel-
          Alias oder Nummer in Deaktivierung
}

define laserTrigger{ # Sequenz-Makro
  17,  cd - qs1,     h; # Formel in Pegel-Zeit
  qs1, cd,           h; # Channel-Alias statt Nummer
```

```
42   19,  cd - qs2 + a, h;
43   qs2, cd + a,       h;
44 }
45
46 sequence 0 wait=f/2 { # Sequenz mit Mindestlaenge vor neuem Trigger
47   LASER_TRIGGER();
48 }
49
50 sequence 1 { # Messsequenz mittels Alternative-Modus
51 start, 0, h;
52   for n offset=0 wait=f {
53   # Schleife mit Offset und Verzoegerung. Offset relativ zu Sequenz oder
         aeusserer Schleife.
54   # Verzoegerung zwischen Durchlaeufen, aber nicht nach letztem Durchlauf
         , um naechsten Trigger nicht zu  verpassen.
55   # Offset der Schaltvorgaenge in der Schleife relativ zum Offset des
         aktuellen Schleifendurchgangs
56   21, 0, h;
57   LASER_TRIGGER();
58   }
59   stop, n*f, h;
60 }
```

B EBNF der bisherigen Syntax

```
ignore '#(*)\n'
ignore whitespaces & line-feeds

alphas  = a..z | A..Z
mums = 0..9
alphanums = alphas | nums
text = (alphanums)+
boolean = 'true' | 'false'
unsignedInt = (nums)+
signedInt = ('+'|'-')? unsignedInt
floatnumber = signedInt ('.' + unsignedInt)?
variable = (alphas|'_') + text
value = varibale | floatnumber
channel = variable | unsignedInt
identifier = '"'  text  '"'

atom = (('-')? value) | ('('  mathExpression ')')
factor = atom  ('^' factor)?
term = factor  (('*'|'/') factor)?
mathExpression = term (('+'|'-') term)?

#variable scope
variableDefinitions = (variable ',' identifier ',' float ',' float ','
    float ';')*
variableScope = 'variables' '{' variableDefinitions '}'

#settings scope
settingsAssignments = (variable '=' (mathExpression | boolean |
    identifier) ';')*
settingsScope = 'settings' '{' settingsAssignments '}'

#channel scope
channelDefinitions = (unsignedInt (',' variable)? ',' identifier ';')*
channelScope = 'channels' '{' channelDefinitions '}'

#menu scope
varAssignment = variable '=' mathExpression
menuOperator = varAssignment | channel
menuEntry = identifier (',' menuOperator)* ';'
```

```
38 defaultMenu = 'default' menuDefinition
39 menuEntries = (menuEntry)* defaultMenu (menuEntry)*
40 menuScope = 'menu' '{' menuEntries '}'
41
42 #sequence scope
43 events = (channel ',' mathExpression ',' mathExpression ';')+
44 sequenceScope = 'sequence' '{' events '}'
45
46 seqFile = variableScope settingsScope channelScope menuScope
      sequenceScope
```

C Grammatik der Sprache und das abgebildete Objektmodell

C.1 Allgemeine Definitionen

alphas	=	a..z | A..Z
nums	=	0..9
alphanums	=	alphas | nums
text	=	(alphanums)+
addop	=	'+' | '-'
multop	=	'*' | '/'
floatnumber	=	(addop)? nums ('.' + nums)?
variable	=	(alphas | '_')? + text
value	=	variable_name | floatnumber
channel	=	variable_name | floatnumber
identifier	=	' " ' text ' " '
boolean	=	'true' | 'false'

C.2 Mathematische Operationen

atom	=	(addop)? (value | ('(' math_expression ')'))
factor	=	atom ('^' factor)?
term	=	factor (multop term)?
math_expression	=	term (addop math_expression)?

Alle mathematischen Ausdrücke erben vom Interface `IValue`. Das entsprechende Klassendiagramm ist in Abbildung 16 auf Seite 63 dargestellt.

C.3 Bereich variables

variableDefinitions	=	(variable_name ',' identifier ',' floatnumber ',' floatnumber ',' floatnumber ';')*
variableScope	=	'variables' '{' variableDefinitions '}'

Das entsprechende Klassendiagramm ist in Abbildung 12 auf Seite 53 dargestellt.

C.4 Bereich settings

settingsAssignments	=	(variable_name '=' (math_expression | identifier) ';')*
settingsScope	=	'settings' '{' settingsAssignments '}'

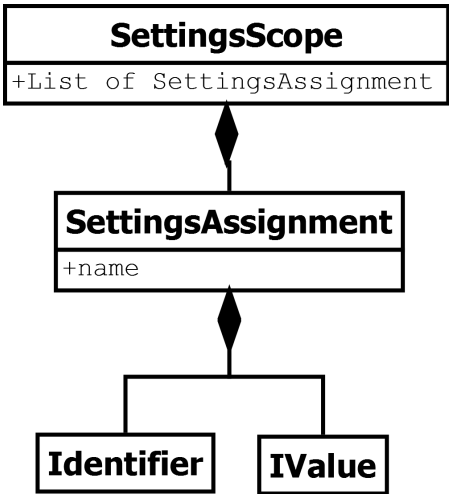


Abbildung 21: UML-Klassendiagramm des Bereiches *settings*

C.5 Bereich channels

channelDefinitions	=	(floatnumber (',' variable_name)? ',' identifier ';')*
channelScope	=	'channels' '{' channelDefinitions '}'

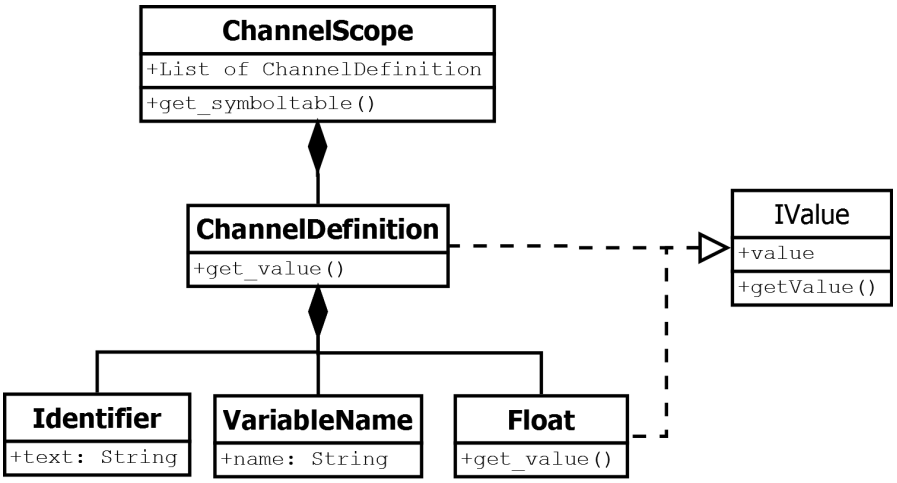


Abbildung 22: UML-Klassendiagramm des Bereiches *channels*

C.6 Bereich menu

inactive_channel	=	',' channel
var_assignment	=	variable_name '=' math_expression
menu_definition	=	identifier (var_assignment)* (inactive_channel)* ';'
default_menu	=	'default' menu_definition
menuScope	=	'menu' '{' (menu_definition)* default_menu (menu_definition)* '}'

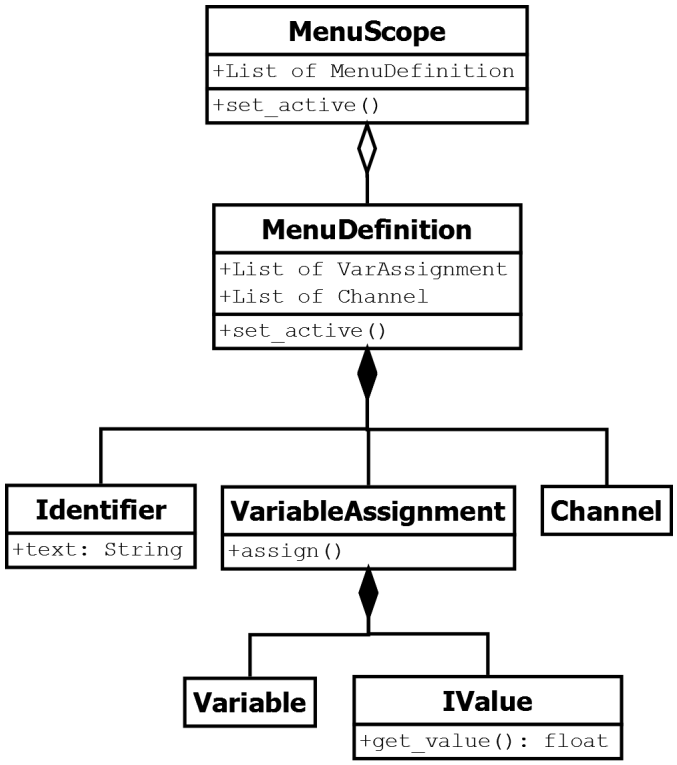


Abbildung 23: UML-Klassendiagramm des Bereiches *menu*

C.7 Events im Bereich der Sequenz- und Makrodefinition

offset_wait	=	'(' ('offset' '=' math_expression)? ('wait' '=' math_expression)? ')'
macro	=	variable_name + (offset_wait)? + ';'
transition	=	channel ',' math_expression ',' math_expression ';'
for_loop	=	'for' math_expression (offset_wait)? '' (event)+ ''
event	=	transition | for_loop | macro
macro_definition	=	'maco' variable_name (offset_wait)? '' (event)+ ''
sequence_definition	=	'sequence' floatnumber + (offset_wait)? + '{' (event)+ '}'

Alle Events erben vom Interface `IEvent`. Das entsprechende Klassendiagramm ist in Abbildung 15 auf Seite 62 dargestellt.

C.8 Hauptanwendung Seco2015

seqFile	=	variableScope (settingsScope)? channelScope (menuScope)? (macro_definition)* (sequenceDefinition)+

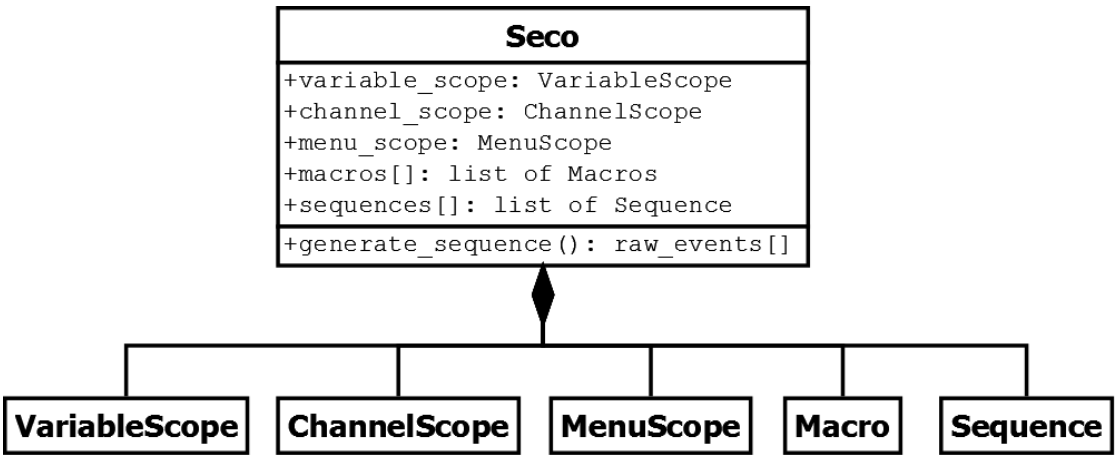


Abbildung 24: UML-Klassendiagramm der Hauptanwendung