FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Generalisierung automatischer Codegenerierung für mathematisch-epidemiologische Modelle

---

Bachelorarbeit

vorgelegt von

Daniel-Michel Richter

aus

Las Palmas

am

09. September 2025

Gutachter:
Prof. Dr. Alexander Ferrein
Dr. Martin Kühn

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, dass alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht sind und dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegt wurde.

Aachen, den 09. September 2025

Unterschrift

Inhaltsverzeichnis

1 Einleitung **1**

2 Stand der Forschung und Technik **5**
2.1 Binding Bibliotheken 5
2.2 Verarbeitung und Analysen von Abstract Syntax Trees 5
2.3 Nutzung von Python Bindings 7
2.4 Parallelisierung der AST-Erstellung 8

3 Technischer Hintergrund **9**
3.1 MEmilio - Modellierung epidemischer Prozesse 9
3.2 Bibliothek Pybind11 12
3.2.1 Erstellen eines Pybind11-Moduls 12
3.2.2 Funktionen 13
3.2.3 Klassen 14
3.3 Abstrakte Syntaxbäume 16
3.4 Python Multiprocessing 19

4 Eigener Ansatz **21**
4.1 Visualisierung von ASTs 21
4.2 Erstellung mehrerer Abstract Syntax Trees 24
4.3 Parallelisierung der AST-Erstellung 26
4.4 Generalisierung des Scanners 32
4.4.1 Identifikation und Extraktion relevanter Strukturen 34
4.4.2 Verarbeitung der extrahierten Daten 36

5 Experimentelle Ergebnisse **41**
5.1 Experimenteller Aufbau 41
5.2 Statistische Auswertung 43
5.3 Auswertung der extrahierten Informationen 46

6 Zusammenfassung **51**

A Codeverfügbarkeit **53**

Literatur **55**

Abbildungsverzeichnis

3.1 Ablauf der Codegenerierung nach [7]. 11
3.2 Darstellung der `add` Funktion als Abstract Syntax Tree. 17
3.3 Textdarstellung des Abstract Syntax Trees der `add` Funktion. 18

4.1 Ablauf der Codegenerierung nach der Parallelisierung. 29

5.1 Laufzeiten und Codezeilenanzahl der Modelle. 43
5.2 Zusammenhang zwischen Codezeilen und mittlerer Laufzeit der Modelle. 44
5.3 Laufzeiten der parallelen Ausführung der Modelle. 45
5.4 Vergleich der gemessenen Laufzeiten als Balkendiagramm. 46
5.5 Extrahierte Informationen einer Klasse aus dem Abstract Syntax Tree. . 48
5.6 Extrahierte Informationen einer Methode aus dem Abstract Syntax Tree. 49

Verzeichnis Quelltextauszüge

3.1 Definition eines Pybind11-Moduls zur Verknüpfung von C++-Code mit Python. 12
3.2 Definition einer C++-Funktion und deren Anbindung an Python mittels Pybind11. 13
3.3 Erweiterung des Bindings der `add` Funktion für den Datentyp `float` mittels Funktionsüberladung. 14
3.4 Verwendung der `add` Funktion in Python. 14
3.5 Definition und Binding der Klasse `Calculator` mittels Pybind11. 15
3.6 Verwendung der gebindeten `multiply` Methode in Python. 15
3.7 Definition der C++-Funktion `add` zur Addition zweier Ganzzahlen. . . . 16
4.1 Funktion zur parallelen AST Erstellung als Pseudocode. 30
4.2 Eintrag in das Dictionary `general_bindings_dict`. 34
4.3 Funktion `bind_functions` zur Auswahl geeigneter Templates für dass generieren einer Funktion. 38
4.4 Template der `direct_binding` Funktion für dass generieren des Codes für eine Funktion. 39
5.1 Eintrag der Testklasse mit ausgewählten Methoden in das Dictionary. . . 47
5.2 Generierte Bindings aus den Informationen des Dictionary. 49

Kapitel 1

Einleitung

Die moderne Softwareentwicklung erfordert zunehmend die nahtlose Integration unterschiedlicher Programmiersprachen [1, 2]. Die Kombination von C++ und Python vereint die Vorteile beider Sprachen. C++ zeichnet sich durch eine hohe Ausführungsgeschwindigkeit und effiziente Nutzung von Ressourcen aus und ist damit besonders für rechenintensive Simulationen geeignet [3]. Python hingegen bietet eine leicht verständliche Syntax sowie eine große Nutzerbasis [4]. Durch die Verknüpfung beider Sprachen lassen sich leistungsfähige Modelle entwickeln, die gleichzeitig effizient und für eine breite Nutzerbasis zugänglich sind. Um eine effiziente Kommunikation zwischen beiden Sprachen zu ermöglichen, kommen sogenannte *Bindings* zum Einsatz [5].

Diese Bindings stellen den zentralen Aspekt der *PyGen*-Funktionalität aus dem MEmilio-Projekt [6] dar. PyGen ermöglicht die automatische Generierung von *Python-Bindings* aus bestehenden C++-Modellen. Der zugrunde liegende *Generator* sorgt dafür, dass diese Modelle ohne zusätzlichen Aufwand auch in Python verfügbar sind. Die automatische Generierung ist dabei von besonderem Vorteil. Anstatt fehleranfällige und zeitaufwendige Bindings von Hand zu schreiben, können neue oder geänderte Modelle schnell und konsistent für Python zugänglich gemacht werden. Dadurch wird die Anwendung und Weiterentwicklung der Modelle für eine breitere Nutzerbasis deutlich erleichtert. Die vorliegende Arbeit knüpft inhaltlich an die Arbeit [7] an, in der die grundlegende PyGen-Funktionalität erstmals konzipiert und implementiert wurde. Im Rahmen dieser Arbeit wurde die bestehende PyGen-Funktionalität erweitert und verbessert.

Ein zentrales Element in diesem Prozess sind die *Abstract Syntax Trees* (ASTs), die eine strukturierte Repräsentation des Quellcodes liefern. Diese ASTs sind häufig verschachtelt und schwer lesbar [8]. Ohne geeignete visuelle Aufbereitung ist es schwierig, fehlerhafte Strukturen zu erkennen. Die manuelle Analyse solcher Bäume ist zudem zeitaufwendig und fehleranfällig. Eine zentrale Erweiterung bestand daher in der Entwicklung einer Visualisierungskomponente, um die ASTs verständlich darzustellen und so den Entwicklungs- und Debugging-Prozess zu erleichtern.

Darüber hinaus ist die Erstellung der Abstract Syntax Trees mit einem nicht zu unterschätzenden Rechenaufwand verbunden, wie auch in [9] aufgezeigt wird. Insbesondere bei der Erstellung aus mehreren Quelldateien ist dieser Aufwand erhöht, was bei sequentieller Verarbeitung zu deutlichen Verzögerungen im Erstellungsprozess führen kann. Um die Laufzeiten zu verkürzen, wurde die Parallelisierung der AST-Erstellung als Lösungsansatz gewählt. Eine gezielte Parallelisierung kann den gesamten Entwicklungsprozess beschleunigen.

Ein zentraler Bestandteil von PyGen ist der sogenannte *Scanner*, der die Struktur des Codes aus dem AST analysiert und relevante Informationen extrahiert. Auf diese Weise dient er als Vermittler zwischen dem vorhandenen Code und den automatisierten Mechanismen zur Generierung von Bindings. Dabei treten typische Herausforderungen auf. Die Modelle müssen anpassbar auf unterschiedliche Anwendungsfälle sein, gleichzeitig aber erweiterbar für neue Anforderungen und langfristig wartbar bleiben. Der Scanner unterstützt diese Ziele, indem er eine klare Trennung zwischen Codeanalyse und Generierung ermöglicht und so die Flexibilität, Modularität und Nachhaltigkeit der Funktionalität fördert.

Allerdings werden in der aktuellen Implementierung in der Scanner-Komponente der PyGen-Funktionalität für jede gesuchte Funktion im AST spezifisch, manuell definierte *Binding-Templates* erstellt. Dieses Vorgehen ist nicht nur aufwendig und fehleranfällig, sondern führt auch zu redundantem Code und erschwert die langfristige Wartung der Funktionalität erheblich. Neue Funktionstypen müssen derzeit jeweils individuell berücksichtigt werden, was die Erweiterbarkeit stark einschränkt und die Einführung neuer Features verlangsamt. Außerdem wächst der Codebestand für die Bindings weiter an, was dazu führt, dass die Struktur an Klarheit verliert und der Code schwieriger nachvollziehbar wird. Diese Arbeit legt den Fokus auf genau diese Herausforderungen. Ziel war es, Methoden zur Visualisierung, Parallelisierung mehrerer ASTs und basierend darauf die Generalisierung des Scanners zu entwickeln, um den Binding-Prozess zu verbessern.

Im Anschluss an die thematische Einführung und die Darstellung der zentralen Problemstellungen folgt eine systematische Aufarbeitung des bestehenden Forschungsstands sowie der technischen Grundlagen. Dabei werden sowohl bisherige Ansätze als auch verwendete Werkzeuge und Bibliotheken analysiert und eingeordnet. Darauf aufbauend werden die im Rahmen dieser Arbeit entwickelten Erweiterungen detailliert vorgestellt. Der Fokus liegt dabei auf der konzeptionellen Herleitung, den konkreten Umsetzungsschritten sowie der Einbettung in die bestehende Systemarchitektur.

Schließlich werden die vorgenommenen Implementierungen durch eine experimentelle Evaluation überprüft und deren Ergebnisse hinsichtlich ihrer Aussagekraft und praktischen Relevanz diskutiert. Den Abschluss bilden eine zusammenfassende Bewertung der zentralen Erkenntnisse sowie ein Ausblick auf mögliche weiterführende Arbeiten und zukünftige Entwicklungspotenziale.

Kapitel 2

Stand der Forschung und Technik

Dieses Kapitel bietet einen Überblick über den aktuellen Stand der Forschung und Technik zur (automatisierten) Generierung von Bindings zwischen C++ und Python sowie der damit verbundenen Arbeiten. Es werden zentrale Konzepte, etablierte Werkzeuge sowie aktuelle Herausforderungen vorgestellt.

2.1 Binding Bibliotheken

Die Bibliotheken Pybind11 [5] und Boost.Python [10] haben sich als zentrale Werkzeuge etabliert, um manuelle Schnittstellen zwischen beiden Sprachen zu ermöglichen. Beide Bibliotheken bieten umfangreiche APIs, mit denen Entwickler explizit Schnittstellen definieren können, um C++-Klassen, Funktionen und Datenstrukturen in Python nutzbar zu machen. Gleichzeitig ist es auch möglich, Python-Funktionen und -Objekte in C++ zu verwenden, sodass eine bidirektionale Integration zwischen beiden Sprachen möglich ist.

In der vorliegenden Arbeit wird die Funktionalität von Pybind11 als Grundlage genutzt, um automatisierte Prozesse für die Generierung von Bindings zu evaluieren und zu erweitern. Dabei wird versucht, die manuelle Arbeit zu reduzieren und eine neue Komponente auf Basis automatisierter Methoden zu entwickeln. Als Weiterentwicklung von Pybind11 existiert zudem Nanobind [11], eine Bibliothek, die ähnliche Funktionalitäten wie Pybind11 bereitstellt, jedoch mit Fokus auf geringeren Speicherverbrauch und schnellere Compiler-Zeiten. Ein Einsatz von Nanobind wurde in dieser Arbeit nicht verfolgt, bietet aber Potenzial für zukünftige Erweiterungen.

2.2 Verarbeitung und Analysen von Abstract Syntax Trees

Ein relevanter Beitrag zur systematischen AST-Verarbeitung und Codeanalyse ist das Clang-Projekt [12], das im Rahmen des LLVM-Projekts [13] entwickelt wurde. LLVM ist ein modulares Compiler-Framework, das Compilerinfrastruktur, Optimierungsstruk-

turen und Codegenerierung für verschiedene Programmiersprachen bereitstellt. Clang selbst stellt einen leistungsfähigen Compiler und Compiler-Frontend für das Parsen, Analysieren und Kompilieren von C und C++ bereit. Besonders hervorzuheben ist die zugängliche und detaillierte AST-API, mit der sich syntaktische und semantische Informationen aus dem Quelltext präzise extrahieren und gezielt verändern lassen. Diese Funktionalität ist insbesondere für die automatische Generierung von Bindings essenziell, da sie eine strukturierte Analyse von Funktionssignaturen, Klassendefinitionen und anderer Sprachelemente ermöglicht. Darüber hinaus erlaubt Clang die Erstellung eigener AST-Tools und bietet damit eine flexible Grundlage für weiterführende Verarbeitungsschritte wie Refactoring [14], Quellcode-Validierung oder eben die Anbindung an andere Programmiersprachen.

Ein Ansatz zur strukturellen Analyse von ASTs im Kontext der automatischen Schwachstellenerkennung wurde von Yamaguchi et al [15] vorgestellt. In dieser Arbeit präsentieren die Autoren ein Verfahren, das es ermöglicht, auf Basis von wenig bekannten Schwachstellen ähnliche fehleranfällige Muster im restlichen Quellcode zu identifizieren.

Kern des Ansatzes ist die Extraktion charakteristischer AST-Substrukturen aus Quellcodefragmenten, die bereits bekannte Sicherheitslücken enthalten. Diese strukturellen Merkmale werden anschließend mithilfe von *Latent Semantic Analysis* untersucht, um semantische Ähnlichkeiten zwischen verschiedenen AST-Teilen zu erkennen. Dabei wandelt die Latent Semantic Analysis die AST-Strukturen in eine numerische Repräsentation um und fasst die Informationen durch eine Dimensionsreduktion zu vergleichbaren Merkmalen zusammen, um verborgene semantische Muster zwischen verschiedenen AST-Teilen zu erkennen. Dabei bedeutet Dimensionsreduktion, dass hoch dimensionale Daten vereinfacht werden, indem ähnliche Informationen komprimiert und auf wenige aussagekräftige Merkmale abgebildet werden. Dabei wird angenommen, dass bestimmte Schwachstellen nicht nur durch konkrete Codezeilen definiert sind, sondern durch wiederkehrende strukturelle Muster, die sich generalisieren und auf andere Codestellen extrapolieren lassen. Das Verfahren erlaubt es somit, über rein textuelle oder syntaktische Übereinstimmungen hinauszugehen und Strukturen zu identifizieren, die potenziell dieselbe Art von Schwachstelle aufweisen. Diese Form der strukturellen Generalisierung ist besonders wertvoll in großen, gewachsenen Codebasen, in denen Sicherheitsanalysen oft händisch oder regelbasiert nur eingeschränkt skalieren.

Die von Yamaguchi et al. [15] vorgestellte Methode zeigt exemplarisch, wie sich strukturierte Informationen aus ASTs zur Identifikation und Generalisierung von Mustern im Quellcode nutzen lassen. Obwohl sich ihr Anwendungsfall auf die Erkennung von Schwachstellen konzentriert, lässt sich der grundlegende Gedanke, das systematische Erkennen und Auswerten von Mustern im AST, direkt auf die Anforderungen in diesem Projekt übertragen.

Visualisierungssysteme für ASTs, wie sie in Graphviz [16] oder in Verbindung mit LLVM eingesetzt werden, spielen eine zentrale Rolle bei der Nachvollziehbarkeit komplexer Programmstrukturen. Diese Tools ermöglichen die Darstellung von ASTs in einer hierarchischen, grafisch interpretierbaren Form, was insbesondere für das Debugging, die strukturelle Analyse und die didaktische Vermittlung hilfreich ist. Graphviz etwa erlaubt über eine einfache, deklarative Beschreibungssprache die Generierung von gerichteten oder ungerichteten Graphen. Auch im LLVM-Projekt existieren Mechanismen zur Visualisierung von internen Repräsentationen, allerdings primär zu Analysezwecken im Compilerbau und weniger im Bereich der Schnittstellengenerierung.

In dieser Arbeit wird Graphviz explizit als Werkzeug zur Visualisierung jener ASTs eingesetzt, die im Rahmen der Binding-Generierung zwischen C++ und Python verwendet werden. Ziel ist es, durch eine automatische, verständliche und reproduzierbare Darstellung der AST-Struktur mögliche Fehlerquellen frühzeitig zu identifizieren und den Entwicklungsprozess, insbesondere bei der Anpassung und Erweiterung der generierten Bindings, zu unterstützen.

2.3 Nutzung von Python Bindings

Projekte wie EpiABM [17], das mithilfe von Pybind11 eine Verbindung zwischen einer in Python implementierten Benutzeroberfläche und einer leistungsoptimierten C++-Kernkomponente herstellt. Dabei dienen die C++-Modelle der effizienten Berechnung agentenbasierter epidemiologischer Modelle. Pybind11 ermöglicht es, die rechenintensiven C++-Funktionen direkt aus Python aufzurufen, ohne dass eine separate Schnittstellenprogrammierung erforderlich ist.

Im Projekt PyBEST [18] wird Pybind11 gezielt eingesetzt, um eine leistungsfähige Python-Schnittstelle für C++-Komponenten zu schaffen. PyBEST dient der elektronischen Strukturrechnung an der Schnittstelle zwischen Chemie und Physik und implementiert komplexe Verfahren zur Beschreibung von Grund- und angeregten Zuständen sowie zur Analyse von Quantenverschränkung. Während das Framework primär in Python entwickelt ist, werden rechenintensive Kernmodule in C++ implementiert und mit Hilfe von Pybind11 eingebunden.

Das Projekt [19] nutzt Codegenerierung, um Finite-Element-Transformationen effizient in Python verfügbar zu machen. Das Modul *FInAT* übernimmt dabei die Handhabung von Interpolations- und Transformationsoperationen. Pybind11 wird dabei im übergeordneten Framework eingesetzt, um C++-Kerne automatisch in Python verfügbar zu machen, sodass die entsprechenden Schnittstellen ohne manuelles Programmieren generiert werden können.

Im Projekt [20] wird Pybind11 benutzt, um automatisch Python-Bindings für C++-Bibliotheken zu generieren. Im Projekt werden die C++-Headerdateien analysiert, und darauf basierend wird automatisch ein abstraktes Repräsentationsmodell der Klassen, Funktionen und Datenstrukturen erstellt, aus dem anschließend die entsprechenden Python-Schnittstellen generiert werden.

Im MEmilio-Projekt werden Pybind11 und automatische Codegenerierung genutzt, um C++-Code für Python verfügbar zu machen. Dabei wird auf die Generalisierung der Bindings geachtet, sodass sie nicht nur für spezifische Funktionen, sondern auch für die modulare Integration verschiedener Komponenten von MEmilio genutzt werden können. Dies ermöglicht eine effiziente Nutzung der C++-Kerne und minimiert gleichzeitig den Overhead. Bisher existiert im Bereich der epidemiologisch-mathematischen Modellierung kein Softwareframework, das eine solche generalisierte, modulare Binding-Infrastruktur bereitstellt.

2.4 Parallelisierung der AST-Erstellung

Die Erstellung von ASTs für C++-Quellcode stellt einen zentralen Schritt in der Binding-Generierung dar. In der Praxis können bei umfangreichen Projekten große Codebasen zu hohen Verarbeitungszeiten führen, insbesondere wenn die AST-Erstellung seriell durchgeführt wird.

Das ROSE-Projekt [21] unterstützt parallele AST-Transformationen und -Analysen. ROSE überführt Quellcode in eine Datenstruktur, die Informationen hält, die eine parallele Traversierung und Bearbeitung mehrerer AST-Knoten ermöglicht. Dadurch lassen sich Analysen und Transformationen beschleunigen und besser auf große Codebasen skalieren.

Obwohl das ROSE-Projekt parallele AST-Transformationen und -Analysen unterstützt, liegt der Fokus primär auf der Bearbeitung und Untersuchung bereits vorhandener ASTs. In der vorliegenden Arbeit hingegen wird ein Ansatz entwickelt, der gezielt die parallele Erstellung von ASTs adressiert. Dieser Ansatz erlaubt es, C++-Code bereits vor der Analyse effizient als AST darzustellen, wodurch die Gesamtverarbeitungszeit reduziert wird.

Kapitel 3

Technischer Hintergrund

Dieses Kapitel bietet einen Überblick über die theoretischen und technischen Grundlagen, die für das Verständnis der in dieser Arbeit entwickelten Ansätze erforderlich sind. Der Fokus liegt dabei auf zentralen Konzepten der Softwareentwicklung, insbesondere im Kontext der automatisierten Schnittstellengenerierung zwischen C++ und Python.

Zu Beginn wird das Softwarepaket MEmilio (aus dem Englischen: *high performance Modular EpideMIcs simuLatIOn software*) [6] vorgestellt, das den praktischen Anwendungskontext dieser Arbeit bildet. Dabei wird auf die mathematisch-epidemiologischen Modelle eingegangen, welche von C++ nach Python überführt werden.
Anschließend wird eine wichtige Softwarekomponente, *MEmilio PyGen*, erläutert, welche in der Praxis zur Erstellung der Bindings verwendet wird. Dabei werden sowohl ihre Eigenschaften als auch im Folgenden eine der zugrunde liegenden Bibliotheken, Pybind11, näher betrachtet.

Ein zentraler technischer Aspekt dieser Arbeit ist die Codegenerierung auf Basis von Abstrakten Syntaxbäumen. Daher werden auch Aufbau, Bedeutung und Einsatzmöglichkeiten von ASTs beschrieben.

3.1 MEmilio - Modellierung epidemischer Prozesse

MEmilio ist eine hochperformante, modulare Simulationssoftware zur Modellierung der Ausbreitung von Infektionskrankheiten. Das System wurde im Kontext der COVID-19-Pandemie entwickelt und befindet sich seither in aktiver Weiterentwicklung durch das Institut für Softwaretechnologie des Deutschen Zentrums für Luft- und Raumfahrt in enger Zusammenarbeit mit Forschenden der Universität Bonn, des Helmholtz-Zentrums für Infektionsforschung und des Forschungszentrums Jülich. Ziel von MEmilio ist es, komplexe epidemiologische Prozesse auf unterschiedlichen Skalen realitätsnah abzubilden und politische Entscheidungsprozesse durch fundierte Modellanalysen zu unterstützen.

Im Zentrum von MEmilio stehen mathematisch-epidemiologische Modelle, die verschiedene Beschreibungsebenen infektiöser Prozesse abbilden können. Dazu zählen *Compartmental models* wie zum Beispiel differentialgleichungsbasierte Modelle [22, 23], Metapopulationsmodelle [24, 25], auf Integro-Differentialgleichungen basierende Modelle [26] sowie agentenbasierte Modelle [27], die das Verhalten einzelner Individuen oder Populationen simulieren. Der modulare Aufbau des Frameworks erlaubt die flexible Kombination unterschiedlicher Modellbausteine, zum Beispiel spezifischer Krankheitsdynamiken mit verschiedenen Mobilitätsmodellen oder Kontaktstrukturen. Dadurch lässt sich die Ausbreitung von Infektionen sowohl auf regionaler als auch auf überregionaler Ebene simulieren und analysieren.

Ein weiterer Aspekt von MEmilio ist die Performanz. Durch gezielte Parallelisierung und optimierte Datenstrukturen kann das System auch großräumige Szenarien mit hoher räumlicher und zeitlicher Auflösung effizient berechnen. Dies ist insbesondere für die Analyse langfristiger Interventionseffekte oder großflächiger Ausbrüche relevant. Die Integration realer epidemiologischer Daten ermöglicht darüber hinaus eine datengetriebene Kalibrierung der Modelle, sodass diese an das tatsächliche Infektionsgeschehen angepasst werden können.

Wichtig ist auch die visuelle Aufbereitung der Simulationsergebnisse. Zu diesem Zweck ist das Framework mit einer webbasierten Visualisierungskomponente gekoppelt, was es erlaubt, Modellverläufe zu untersuchen und Parameterexperimente zu visualisieren. Die Ergebnisse könnten so auch von Entscheidungsträgern ohne tiefere technische Kenntnisse interpretiert werden.

Aufbauend auf der in MEmilio beschriebenen Modellierungs-Infrastruktur wurde im Rahmen dieser Arbeit die PyGen-Funktionalität weiterentwickelt, die die automatische Generierung von Python-Bindings für C++-Komponenten innerhalb der Simulationsarchitektur ermöglicht.

Um das Grundprinzip der automatischen Codegenerierung innerhalb von PyGen nachvollziehbar zu machen, wird im Folgenden ein Überblick über die zentrale Architektur und den Ablauf der Verarbeitung gegeben. Die gesamte Pipeline gliedert sich in mehrere klar abgegrenzte Komponenten, die schrittweise aufeinander aufbauen und in Abbildung 3.1 dargestellt werden.

Der Einstiegspunkt bildet der Scanner, der den Quellcode eines bestehenden C++-Modells analysiert. Ziel ist es, jene Informationen zu extrahieren, die für die spätere Generierung der Python-Bindings notwendig sind. Im Rahmen dieser Traversierung identifiziert der Scanner relevante Sprachelemente wie Klassendefinitionen, Methoden

oder Argumenttypen. Diese Informationen werden in eine spezielle Datenstruktur, die *Intermediate Representation* oder auch *Zwischencode* überführt. Diese Struktur dient als Vermittlungsinstanz zwischen Analyse und Generierung. Sie speichert nicht nur die extrahierten Metadaten, sondern abstrahiert sie zugleich vom konkreten Quelltext.

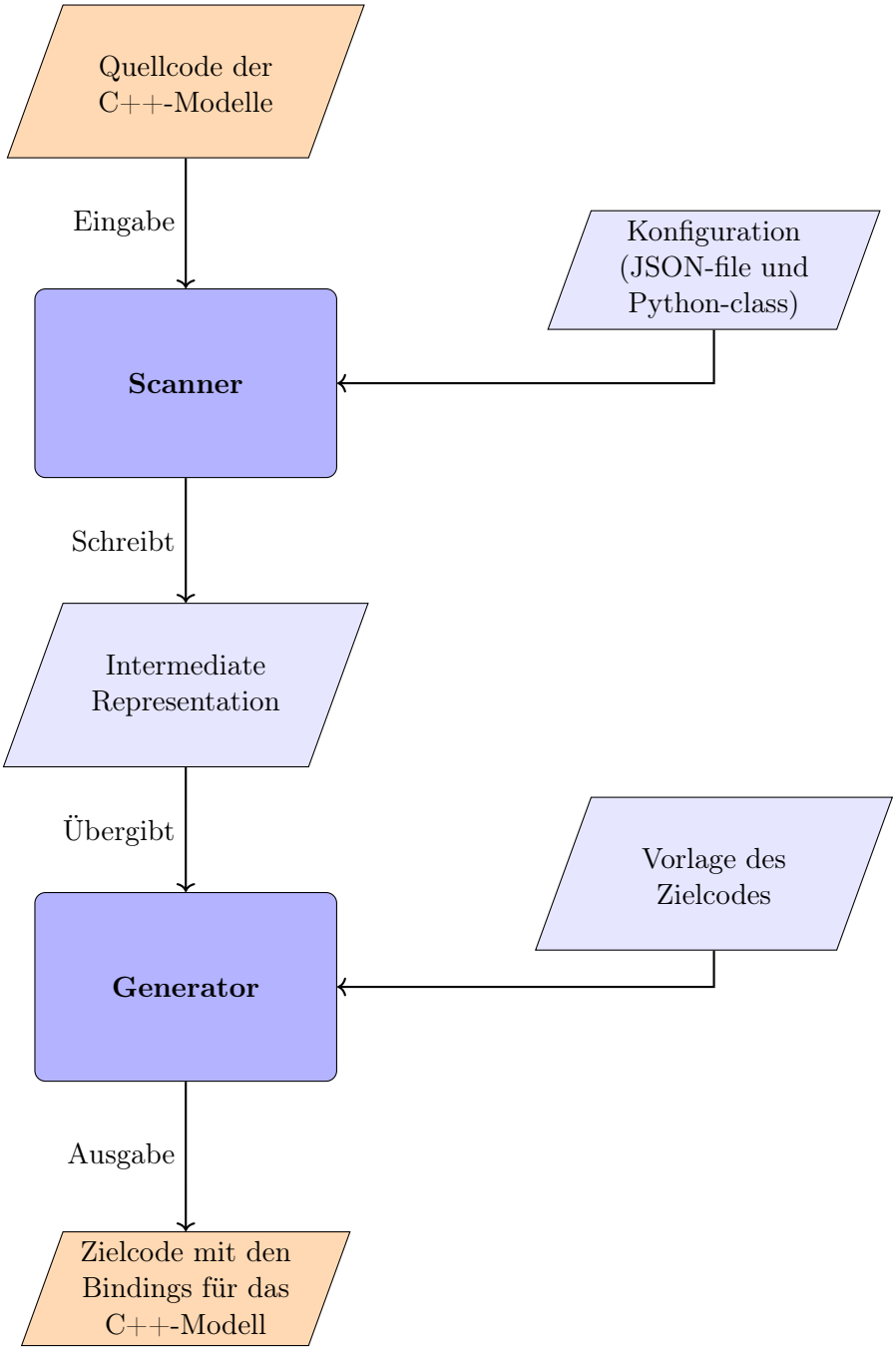

Abbildung 3.1: Ablauf der Codegenerierung nach [7]. Die blauen Kästen stehen für Teile des Codegenerators, die orangen stehen für Input und Output. Trapezförmige Kästen stellen Daten dar und die abgerundeten Kästen, Logikklassen des Codegenerators.

Im nächsten Schritt übernimmt der *Generator* die Verarbeitung dieses Zwischencodes. Basierend auf konfigurierbaren Vorlagen erzeugt er den Zielcode, der schließlich die gewünschten Python-Bindings enthält. Die Trennung von Analyse und Codegenerierung erlaubt eine modulare und erweiterbare Architektur, in der neue Modelle ohne tiefgreifende Änderungen in das Gesamtsystem integriert werden können.

3.2 Bibliothek Pybind11

Pybind11 [5] ist eine Bibliothek, die es ermöglicht, Python-Bindings für bestehende C++-Codebasen zu erstellen. Dadurch können Funktionen und Klassen sowohl von C++ in Python als auch umgekehrt genutzt werden, was eine nahtlose Kombination beider Sprachen erlaubt. Die folgenden Abschnitte erläutern die grundlegende Struktur eines Pybind11-Moduls und zeigen exemplarisch, wie Funktionen und Klassen auf Basis der Pybind11-Dokumentation [28] eingebunden werden können. Der Aufbau orientiert sich dabei an der in der vorherigen Arbeit beschriebenen Struktur [7].

3.2.1 Erstellen eines Pybind11-Moduls

Ein Pybind11-Modul besteht typischerweise aus einer zentralen `PYBIND_MODULE`-Definition. Diese Definition bildet den Einstiegspunkt für das Modul und definiert die Schnittstelle zwischen C++ und Python. Innerhalb dieses Blocks werden sämtliche Funktionen, Klassen, Datentypen und Konstanten, die von C++ nach Python exponiert werden sollen, registriert. Dabei wird jedem Element ein entsprechender Name zugewiesen, unter dem es später im Python-Kontext aufgerufen werden kann.

```
1  namespace py = pybind11 ;
2
3  PYBIND11_MODULE ("example", m ) {
4  ...
5  }
```

Quelltext 3.1: Definition eines Pybind11-Moduls zur Verknüpfung von C++-Code mit Python.

Im Quelltext 3.1 wird ein neues Pybind11-Modul mit dem Namen `example` definiert. Das `PYBIND_MODULE`-Makro dient als Einstiegspunkt für das Modul und verknüpft C++-Code mit Python. Der Parameter `m` stellt ein Objekt dar, über das Funktionen, Klassen und Variablen im Pybind11-Modul registriert und für die Nutzung in Python verfügbar gemacht werden können. Innerhalb dieses Blocks können alle gewünschten Elemente der C++-Bibliothek für die Nutzung in Python bereitgestellt werden. Die Zeile `namespace py = pybind11;` dient dazu, den langen Namespace-Namen abzukürzen und so den Code lesbarer zu gestalten. Diese Zeile wird in den folgenden Beispielen erwartet, aber aus dem Code verworfen.

3.2.2 Funktionen

Ein wesentlicher Bestandteil beim Erstellen eines Pybind11-Moduls ist das Bereitstellen von C++-Funktionen, um diese in Python verfügbar zu machen. Funktionen werden innerhalb des `PYBIND_MODULE`-Blocks mittels der Methode `m.def()` registriert. Dabei wird der C++-Funktion ein Name zugewiesen, unter dem sie später im Python-Kontext aufgerufen werden kann. Funktionsparameter lassen sich explizit benennen und optionale Argumente durch Standardwerte definieren. Dies ermöglicht eine benutzerfreundliche und intuitive Schnittstelle auf Python-Seite, ohne dabei die Flexibilität der zugrunde liegenden C++-Implementierung einzuschränken. Die klare Trennung zwischen Funktionssignatur und Implementierung erleichtert die Wartung und Erweiterung des Moduls.

```
1  int add(int a, int b) {
2      return a + b;
3  }
4
5  PYBIND11_MODULE("example", m) {
6      m.def("add", &add, py::arg("a"), py::arg("b"));
7  }
```

Quelltext 3.2: Definition einer C++-Funktion und deren Anbindung an Python mittels Pybind11.

Im Quelltext 3.2 wird eine einfache C++-Funktion namens `add` definiert, die zwei Ganzzahlen entgegennimmt und ihre Summe zurückgibt. Innerhalb des `PYBIND_MODULE`-Blocks wird diese Funktion über `m.def()` für Python sichtbar gemacht. Der erste Parameter `add` legt fest, wie die Funktion in Python aufgerufen werden kann. Der zweite Parameter `&add` ist ein Funktionspointer und verweist auf die tatsächliche C++-Implementierung. Die nachfolgenden Argumente `py::arg("a")` und `py::arg("b")` dienen der expliziten Benennung der Parameter. Dadurch ist die Funktion nicht nur über einen Funktionsaufruf wie `example.add(3,4)` nutzbar, sondern kann zusätzlich mit benannten Argumenten verwendet werden. Diese Möglichkeit erhöht in komplexeren Modulen die Verständlichkeit, da beim Aufruf unmittelbar ersichtlich ist, welche Werte welchen Parametern zugewiesen werden. Dies ist beispielsweise hilfreich, wenn Funktionen eine größere Anzahl an Parametern besitzen oder mehrere Parameter denselben Datentypen haben und somit bei rein positionsbasierten Aufrufen leicht Verwechslungen entstehen können.

Ein weiteres Merkmal von C++ ist die Funktionsüberladung. Dabei können mehrere Funktionen mit demselben Namen, aber unterschiedlicher Parameteranzahl oder -typen definiert werden. Der Compiler entscheidet anhand der übergebenen Argumente, welche Version zur Laufzeit aufgerufen wird.

```cpp
int add(int a, int b) {
    return a + b;
}

int add(float a, float b) {
    return a + b;
}

PYBIND11_MODULE("example", m) {
    m.def("add", py::overload_cast<int, int>(&add);
    m.def("add", py::overload_cast<float, float>(&add);
}
```

Quelltext 3.3: Erweiterung des Bindings der `add` Funktion für den Datentyp `float` mittels Funktionsüberladung.

Im Quelltext 3.3 wird demonstriert, wie zwei überladene C++-Funktionen mit demselben Namen, jedoch unterschiedlichen Parametertypen (`int` und `float`), mithilfe von Pybind11 für Python verfügbar gemacht werden. Da C++-Funktionen überladen werden können, muss in Pybind11 angegeben werden, welche Signatur gebunden werden soll. Mithilfe von `py::overload_cast<>()` kann explizit angegeben werden, welche Funktionssignatur gemeint ist. Dadurch wird sichergestellt, dass sowohl `add(int, int)` als auch `add(float, float)` korrekt gebunden werden. Python wählt dann zur Laufzeit automatisch die passende Funktionsvariante basierend auf den übergebenen Typen aus.

```python
int_result = example.add(2, 2)
float_result = example.add(2.22, 2.22)
```

Quelltext 3.4: Verwendung der `add` Funktion in Python.

Im Quelltext 3.4 wird veranschaulicht, dass durch die unterschiedlichen Datentypen, also Ganzzahlen und Gleitkommazahlen, automatisch entschieden wird, welche Variante der Funktion aufgerufen wird. Dadurch lässt sich eine flexible und benutzerfreundliche Schnittstelle realisieren, ohne dass der Nutzer sich mit den Details der Überladung in C++ auseinandersetzen muss.

3.2.3 Klassen

Neben Funktionen spielt auch die Exponierung von C++-Klassen eine zentrale Rolle beim Einsatz von Pybind11. Klassen ermöglichen eine objektorientierte Strukturierung des Codes, die sich durch die Bindung in Python nahtlos fortsetzen lässt. Innerhalb des `PYBIND_MODULE`-Blocks werden C++-Klassen mithilfe von `py::class_` registriert. Dabei wird der Klasse ein Name zugewiesen und es können Konstruktoren, Member-Funktionen, Member-Variablen sowie Operatoren explizit gebunden werden. Der grund-

legende Mechanismus besteht darin, mit `py::class_` ein neues Python-Objekt anzulegen, das die gewünschte C++-Klasse repräsentiert. Anschließend können Methoden über `.def()` hinzugefügt werden. So entsteht ein voll funktionsfähiges Python-Pendant zur ursprünglichen C++-Klasse, das direkt instanziiert und verwendet werden kann. Optional lassen sich Dokumentationen, Standardargumente und Zugriffsbeschränkungen integrieren.

Quelltext 3.5 zeigt, wie eine C++-Klasse mit einer Member-Funktion mittels Pybind11 für Python zugänglich gemacht wird. Zudem wird veranschaulicht, wie diese Funktion anschließend in einem Python-Skript aufgerufen wird.

```
1   class Calculator {
2   public:
3       Calculator() = default;
4
5       int multiply(int x, int y) {
6           return x * y;
7       }
8   };
9
10  PYBIND11_MODULE("example", m) {
11      py::class_<Calculator>(m, "Calculator")
12          .def(py::init<>())
13          .def("multiply", &Calculator::multiply, py::arg("x"), py
                ::arg("y"));
14  }
```

Quelltext 3.5: Definition und Binding der Klasse `Calculator` mittels Pybind11.

Im Quelltext 3.5 wird eine C++-Klasse `Calculator` mit einer Methode `multiply` definiert. Mittels `py::class_` wird die Klasse im Python-Modul registriert. Der Konstruktor wird über `.def(py::init<>())` eingebunden, sodass Objekte in Python instanziiert werden können. Die Methode `multiply` wird über `.def(multiply", &Calculator::-multiply)` verfügbar gemacht und erhält eine benannte Parametrisierung. In Python kann die Klasse anschließend, wie in Quelltext 3.6 dargestellt, verwendet werden.

```
1   calc = Calculator()
2   result = calc.multiply(6, 7)
```

Quelltext 3.6: Verwendung der gebindeten `multiply` Methode in Python.

Diese Art der Klassenbindung ist essenziell, um komplexere C++-Strukturen mit Python kompatibel zu machen und gleichzeitig eine modulare Architektur zu ermöglichen.

3.3 Abstrakte Syntaxbäume

Ein zentrales Element bei der automatisierten Verarbeitung von Quellcode stellt die Verwendung von Abstract Syntax Trees (ASTs) dar. ASTs sind strukturierte, hierarchische Darstellungen des Quelltextes, in denen die syntaktischen und semantischen Bestandteile eines Programms formal und eindeutig repräsentiert werden. Anders als der rohe Quellcode, der primär für menschliche Leser gedacht ist, ermöglichen ASTs eine maschinelle Interpretation und Analyse der Codestruktur. Sie abstrahieren von konkreter Syntax wie Formatierung oder Klammerung und stellen stattdessen die logische Struktur des Programms in Form eines gerichteten Baumes dar.

In einem AST wird jeder Knoten einem bestimmten Sprachelement zugeordnet, beispielsweise einer Funktion, einer Klassendefinition oder einem Ausdruck. Die hierarchische Organisation der Knoten erlaubt es, die Einbettung einzelner Konstrukte in übergeordnete Kontexte zu erfassen, etwa einer Variable innerhalb eines Funktionsrumpfs oder einer Methode innerhalb einer Klasse.

Für die im Projekt verwendete C++-Analyse kommt die Clang-Frontend-Infrastruktur zum Einsatz, die im Rahmen des LLVM-Projekts entwickelt wurde. Clang bietet eine AST-API, die nicht nur syntaktische Informationen liefert, sondern auch tiefgehende semantische Verknüpfungen auflöst. Dies ermöglicht, beispielsweise die Zugehörigkeit einer Methode zu einer bestimmten Klasse oder die Rückgabetypen komplexer Signaturen korrekt zu identifizieren [29].

Im Folgenden wird die Funktion `add` in Quelltext 3.7 gezeigt, welche zwei Ganzzahlen entgegennimmt und deren Summe zurückgibt. Diese Funktion stellt eine einfache Implementierung der Addition in C++ dar.

```
1  int add(int a, int b) {
2      return a + b;
3  }
```

Quelltext 3.7: Definition der C++-Funktion `add` zur Addition zweier Ganzzahlen.

Um den inneren Aufbau dieser Funktion aus Sicht des Compilers zu verdeutlichen, wird der zugehörige Abstract Syntax Tree dargestellt. Dies ermöglicht eine formale Analyse der Funktion und dient als Grundlage für weitere Verarbeitungsschritte.

Die in Abbildung 3.2 gezeigte Baumstruktur visualisiert die abstrakte syntaktische Repräsentation der Funktion. Im Gegensatz zur konkreten Quelltextdarstellung abstrahiert der AST von spezifischen Syntaxelementen wie Klammern oder Schlüsselwörtern und stellt ausschließlich die semantisch relevanten Bestandteile der Funktion dar.

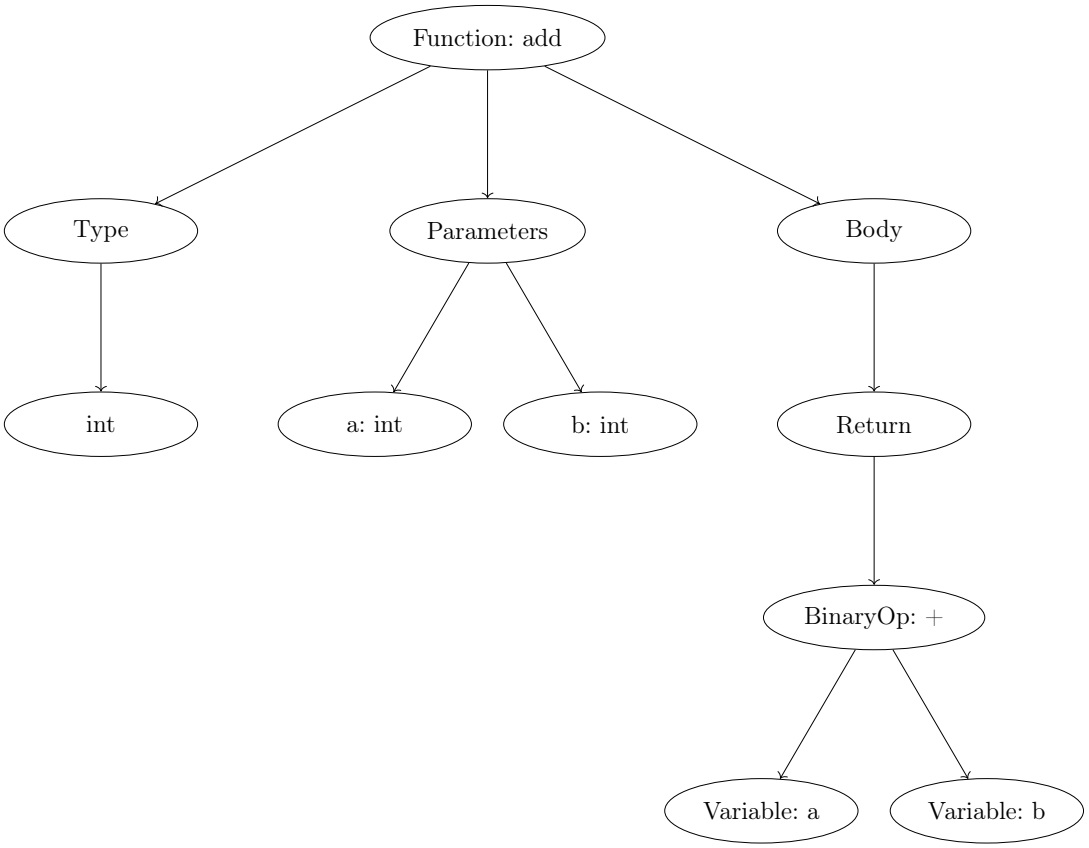

Abbildung 3.2: Darstellung der `add` Funktion als Abstract Syntax Tree.

Der oberste Knoten repräsentiert die Funktionsdeklaration, aus der sich mehrere Teilstrukturen ableiten. So ist der Rückgabewert `int` als eigener Zweig unterhalb des `Type` angegeben. Die beiden Funktionsparameter `a` und `b`, jeweils vom Typ `int`, erscheinen ebenfalls als Kindknoten der Funktionsdeklaration. Der Körper der Funktion besteht aus einer Rückgabeanweisung, die die Summe von `a` und `b` zurückgibt.

Diese abstrahierte Darstellung zeigt, wie die Bestandteile einer scheinbar einfachen Funktion organisiert sind. Sie verdeutlicht, wie Quellcode verarbeitet und analysiert werden kann, um darauf aufbauend Transformationen oder Überprüfungen vorzunehmen. Gerade in der automatisierten Verarbeitung und Analyse von Quelltext stellt der AST eine zentrale Grundlage dar.

Jeder Knoten im AST wird durch einen sogenannten *Cursor* repräsentiert. Ein Cursor ist im Clang-Framework eine zentrale Abstraktion, die es ermöglicht, den AST zu durchlaufen und auf die Details der Knoten zuzugreifen. Konkret ist ein Cursor ein Zeiger oder *Handle* auf einen bestimmten Knoten im AST, zum Beispiel eine Funktion, eine Variable, einen Ausdruck oder eine Anweisung.

```
add ... <CursorKind.FUNCTION_DECL>
        <SourceLocation ...>

  a ... <CursorKind.PARM_DECL>
        <SourceLocation ...>

  b ... <CursorKind.PARM_DECL>
        <SourceLocation ...>

    ... <CursorKind.COMPOUND_STMT>
        <SourceLocation ...>

      ... <CursorKind.RETURN_STMT>
          <SourceLocation ...>

        + ... <CursorKind.BINARY_OPERATOR>
              <SourceLocation ...>

          a ... <CursorKind.DECL_REF_EXPR>
                <SourceLocation ...>

          b ... <CursorKind.DECL_REF_EXPR>
                <SourceLocation ...>
```

Abbildung 3.3: Textdarstellung des Abstract Syntax Trees der `add` Funktion.

Die Abbildung 3.3 zeigt zentrale Eigenschaften, die von Clang AST Cursor beschrieben werden. Diese Eigenschaften sind essenziell, um den Aufbau und die semantische Bedeutung eines AST nachvollziehen zu können. Die Darstellung ist dabei bewusst vereinfacht, da Cursor über eine größere Anzahl an Eigenschaften verfügen, von denen hier nur die für das Verständnis relevanten hervorgehoben werden. Ein Cursor enthält zunächst einen *Anzeigenamen* (engl. display name), der das zugrunde liegende Objekt im Quellcode repräsentiert. In der grafischen Darstellung ist der Name `add` ein Beispiel für einen Anzeigenamen. Weiterhin besitzt jeder Cursor eine *Art*, die den Typ des dargestellten Objekts beschreibt. Cursor-Arten sind zum Beispiel `NamespaceDecl` für Namensräume oder `ConstructorDecl` für Konstruktoren. In der Abbildung sind die Cursor-Arten unter `<CursorKind.ART_DES_CURSORS>` aufgeführt. Darüber hinaus kennt jeder Knoten seine *Eltern- und Kindknoten*. Diese Hierarchie ist essenziell für die Traversierung und Analyse, da sie die Struktur des Programmcodes widerspiegeln. Eine weitere wichtige Eigenschaft ist die Liste aller zugehörigen *Tokens* des Objekts, die aus dem Quellcode stammen. Diese Tokens, zum Beispiel Schlüsselwörter, Operatoren oder Identifier, werden von Clang im Rahmen der lexikalischen Analyse mithilfe eines *Tokenizers* extrahiert. Zudem verfügt jeder Cursor über eine *Start- und Endposition* im Quelltext. Diese Positionen enthalten präzise Angaben zur Ursprungsdatei sowie zur Zeile und Spalte, was eine exakte Rückverfolgbarkeit zum Quellcode ermöglicht.

Einige Cursor-Arten können darüber hinaus auch spezifische Zusatzinformationen enthalten. Ein Cursor vom Typ `DeclRefExpr` verweist auf eine zuvor deklarierte Variable oder Funktion. Seine spezifischen Zusatzinformationen wären unter anderem:

- Auf welches Objekt (z.B. `ParmVarDecl` oder `VarDecl`) wird verwiesen?
- Welchen Typ hat das referenzierte Objekt?
- Ist die Referenz Teil eines bestimmten Ausdrucks (z.B. `BinaryOperator`)?

Um Informationen zu extrahieren, muss der Baum vollständig durchlaufen werden. Dabei erfolgt die Traversierung in Form einer Tiefensuche (engl. depth first search, DFS) [30], bei der jeweils ein Pfad rekursiv bis zu einem Blattknoten verfolgt wird, bevor andere Zweige betrachtet werden. Die Implementierung dieses Traversierungsprozesses erfolgt im Scanner durch rekursive Funktionsaufrufe.

Beim Durchlaufen des ASTs wird jeder Knoten besucht und kann dabei auf spezifische Eigenschaften überprüft werden. Dies ermöglicht es, gezielt nur jene Knoten weiterzuverarbeiten, die für die Analyse oder Extraktion von Informationen relevant sind. Eine häufig genutzte Methode besteht darin, die Cursor anhand ihrer Art (z.B. Funktionsdeklaration, Variablendeklaration, Methodenaufruf) zu identifizieren und zu filtern. Darüber hinaus können auch zusätzliche Kriterien wie die Zugehörigkeit zu einem bestimmten Namensraum, bestimmte Typinformationen oder vorhandene Initialisierungswerte berücksichtigt werden. Um die Analyse zu präzisieren, wird häufig nur ein definierter Teilbereich des AST betrachtet. Es kann vorgegeben werden, dass nur Elemente eines bestimmten Moduls oder Bereichs berücksichtigt werden. Diese strukturierte und selektive Traversierung ermöglicht eine gezielte Extraktion semantischer Informationen aus dem Quellcode und bildet die Grundlage für nachgelagerte Analyseschritte.

3.4 Python Multiprocessing

Das Python-Modul *multiprocessing* [31] bietet eine leistungsfähige Möglichkeit, Programme parallel auszuführen, indem unabhängige Prozesse gestartet werden, die jeweils über einen eigenen Speicherbereich verfügen. Im Gegensatz zu Multithreading, bei dem mehrere Threads denselben Speicher eines Prozesses teilen, verhindert die Prozess-Isolation typische Probleme wie Race Conditions, Deadlocks oder inkonsistente Datenzugriffe. Jeder Prozess arbeitet vollständig autonom, kann aber über bereitgestellte Kommunikationsmechanismen wie *Queue*, *Pipe* oder *Manager* Daten mit anderen Prozessen austauschen, ohne die Sicherheit oder Integrität der Daten zu gefährden. Die grundlegenden Bausteine des Moduls sind die Klasse *Process* zum expliziten Erstellen und Starten von Prozessen sowie *Pool*, das die parallele Ausführung von Funktionen auf mehreren Arbeitern automatisiert und dabei die Zuweisung von Aufgaben an freie

CPU-Kerne optimiert. Für komplexere Szenarien stellt das Modul zusätzliche Synchronisationsmechanismen wie *Lock*, *Event* oder *Semaphore* bereit, um gezielte Zugriffskonflikte bei gemeinsam genutzten Ressourcen zu steuern.

Typische Anwendungsfälle sind rechenintensive Aufgaben, die in unabhängige Teilprozesse aufgeteilt werden können, etwa bei der Verarbeitung großer Datenmengen, Bildverarbeitung oder komplexen Simulationen [32, 33].

Zusätzlich sollten bei der parallelen Verarbeitung in Python mögliche Fallstricke berücksichtigt werden. So kann die Erstellung vieler Prozesse in kurzer Zeit zu einem hohen Speicherverbrauch und Scheduling-Overhead führen, was die Performance unter Umständen verschlechtert. Außerdem ist zu beachten, dass nicht alle Python-Module oder Bibliotheken problemlos in mehreren Prozessen parallel arbeiten können, insbesondere wenn sie globale Zustände verwenden.

Kapitel 4

Eigener Ansatz

In dieser Arbeit zeigten sich bei der Analyse und Verarbeitung von ASTs mehrere zentrale Herausforderungen. Zum einen fehlt es häufig an Überblick hinsichtlich der Struktur und des Aufbaus der erzeugten Bäume, was eine gezielte Analyse und Optimierung erschwert. Zum anderen stellt die sequentielle Verarbeitung mehrerer Quellcode-Dateien ein Effizienzproblem dar. Darüber hinaus ist die bestehende Scanner-Implementierung wenig flexibel und lässt sich nur eingeschränkt auf neue Anforderungen übertragen.

Um diese Probleme zu adressieren, wurde ein konzeptioneller Lösungsansatz entwickelt, der aus drei wesentlichen Bestandteilen besteht, die Visualisierung von ASTs, die Parallelisierung ihrer Erstellung sowie die Generalisierung des Scanners. Diese drei Bausteine bauen methodisch aufeinander auf und bilden zusammen einen ganzheitlichen Ansatz zur Verbesserung der Analyse- und Verarbeitungsschritte. Im Folgenden werden diese Teilaspekte im Detail vorgestellt.

4.1 Visualisierung von ASTs

Zu Beginn des Projekts zeigte sich, dass die bestehende Darstellung der ASTs nur eingeschränkten analytischen Mehrwert bot. Zwar war es möglich, grundlegende strukturelle Elemente innerhalb des ASTs zu erkennen, doch war diese Methode stark vereinfacht. Die ursprüngliche Visualisierung der ASTs stieß in mehreren zentralen Punkten an ihre Grenzen und war für eine tiefergehende Analyse des Programmcodes nur eingeschränkt brauchbar.

Ein zentrales Defizit bestand in der fehlenden Zeilenreferenzierung. Ohne eine explizite Zuordnung von Knoten zu den entsprechenden Zeilen im Quellcode war es nur erschwert möglich, eine visuelle Struktur im Baum mit der tatsächlichen Funktion oder Anweisung im Code zu verknüpfen. Dies erschwerte nicht nur das gezielte Auffinden bestimmter Codeabschnitte, sondern führte insbesondere bei großen Modellen mit verschachtelter Logik zu Verwirrung und unnötigem Suchaufwand.

Ein weiteres Problem lag in der mangelnden Sichtbarkeit von Referenzen und semantischen Zusammenhängen. Gerade wenn Variablen mehrfach verwendet oder über verschiedene Funktionen hinweg weitergereicht werden, blieb die Nachvollziehbarkeit dieser Verbindungen in der einfachen, linearen Darstellung weitgehend verborgen. Ohne visuelle Hinweise auf solche Beziehungen war eine übergeordnete Analyse der Datenflüsse oder Abhängigkeiten kaum möglich.

Hinzu kam, dass die ursprüngliche Visualisierung stets den vollständigen Baum abbildete, unabhängig davon, welcher Ausschnitt des Codes analysiert werden soll. Gerade bei der Untersuchung von einzelnen Funktionen oder kleineren Abschnitten wäre es hilfreich gewesen, nur den relevanten Teilbaum darzustellen, eine Möglichkeit die in der bisherigen Lösung nicht vorgesehen war. Die Folge war nicht nur eine erschwerte Lesbarkeit der Ausgabe, sondern auch eine deutlich längere Generierungszeit, da unnötige Strukturen mit ausgegeben wurden. Zusammengenommen führten diese Limitationen dazu, dass die Visualisierung ihren eigentlichen Zweck, die Unterstützung bei der Analyse, nur unzureichend erfüllen konnte.

Die Konzeption der AST-Visualisierung wurde gezielt darauf ausgerichtet, die im vorherigen Abschnitt identifizierten Schwachstellen der ursprünglichen Darstellung zu beheben und ein Analysewerkzeug zu schaffen, das sowohl funktional als auch nutzerfreundlich ist. Im Zentrum stand dabei das Ziel, die Struktur abstrakter Syntaxbäume nicht nur vollständig, sondern auch nachvollziehbar, selektiv und in Bezug zum Quelltext abbilden zu können.

Ein zentraler konzeptioneller Baustein war die Einführung eindeutiger Identifikatoren und Zeilennummern, die es ermöglichen sollten, jeden Knoten im AST einer konkreten Zeile im Quellcode zuzuordnen und diese mit einer eindeutigen Nummer suchen zu können. Diese ID-Zuweisung und Zeilennummerierung legen den Grundbaustein für die folgenden Punkte.

Es sollte die Möglichkeit geben, anhand der eindeutigen Nummer für jedes Element im Baum, bestimmte Teilbäume gezielt zu extrahieren. Das Konzept sieht vor, einzelne Abschnitte, wie ganze Klassen oder Funktionen, isoliert darzustellen. Durch diese selektive Visualisierung wird eine gezieltere Analyse spezifischer Programmteile ermöglicht, da überflüssige Strukturen ausgeblendet werden und der betrachtete Kontext klarer hervortritt. Dies erleichtert insbesondere die Untersuchung komplex verschachtelter Konstrukte.

Ein weiterer konzeptioneller Schwerpunkt lag auf der Visualisierung semantischer Beziehungen. Um komplexe Abhängigkeiten zwischen Variablen, Kontrollflüssen und Funk-

tionsaufrufen sichtbar zu machen, wurde eine grafische Repräsentation im Baumformat vorgesehen. Diese Darstellung orientiert sich an bewährten Standards für Visualisierungen wie sie durch Werkzeuge wie *Graphviz* ermöglicht werden. Ziel war es, nicht nur eine strukturelle Darstellung des Codes zu generieren, sondern auch die logische und funktionale Struktur greifbar zu machen, etwa durch Knotenverbindungen, Pfeile und hierarchische Ebenen.

Parallel dazu sah das Konzept eine flexible Auswahl an Ausgabeformaten vor. Die Visualisierung sollte nicht ausschließlich als Bild vorliegen, sondern auch in textueller Form als strukturierte Baumansicht in einer Textdatei oder als kompakter Ausschnitt in der Konsole. Dadurch ergeben sich unterschiedliche Anwendungsmöglichkeiten zur schnellen Code-Inspektion oder zur Dokumentation bestimmter Codemuster.

Ein weiterer Bestandteil des Konzepts betrifft die Struktur und den Ablauf der Visualisierung innerhalb der Gesamtarchitektur. Die Visualisierung ist als eigenständiger Verarbeitungsschritt nach der Erzeugung der ASTs und weiteren Analyseprozessen konzipiert. Bei der Erzeugung des Baumes werden jedem Knoten direkt ein eindeutiger Identifikator zugewiesen, woraufhin nach der Erstellung der Baum visualisiert werden kann. Die Zuordnung der Identifikatoren erfolgt bewusst bereits während der AST-Erstellung, um eine nachträgliche Traversierung des gesamten Baumes zur ID-Vergabe zu vermeiden. Letzteres würde einen unnötigen Mehraufwand bedeuten und die Laufzeit erheblich erhöhen, insbesondere bei tief verschachtelten oder sehr umfangreichen Strukturen. Die Trennung von der Vergabe von IDs und Visualisierung erlaubt nicht nur eine bessere Wartbarkeit und Modularität, sondern schafft auch die Möglichkeit, einzelne Komponenten unabhängig voneinander zu erweitern oder auszutauschen.

Ein weiterer Aspekt des Konzepts betrifft die Benutzerfreundlichkeit und die flexible Anwendbarkeit. Die Nutzung der Visualisierung ist so gestaltet, dass sie sich einfach und zielgerichtet in bestehende Arbeitsabläufe integrieren lässt. Die Klasse `Visualization` stellt ein Interface für den Anwender dar, in der die Funktionen die Ausgabe in unterschiedlichen Formaten ermöglicht. Diese Formate lassen sich auch kombinieren, wodurch eine gleichzeitige Generierung unterschiedlicher Repräsentationen möglich wird. So kann beispielsweise eine textuelle Struktur zur schnellen Übersicht sowie eine visuelle Darstellung zur Analyse komplexer Beziehungen erzeugt werden.

Zusammenfassend lässt sich sagen, dass die entwickelte Visualisierungskomponente nicht nur bestehende Defizite in der AST-Darstellung gezielt adressiert, sondern zugleich den Grundstein für weiterführende Anwendungen legt. Durch ihre modulare und erweiterbare Konzeption ist sie praktisch anwendbar und bietet damit ein Werkzeug, das es ermöglicht, die Analyse einfacher zu gestalten. Die Möglichkeit, gezielt bestimmte Codeabschnitte zu analysieren, semantische Beziehungen sichtbar zu machen und verschie-

dene Ausgabeformate flexibel zu kombinieren, macht die Lösung vielseitig einsetzbar. Damit trägt sie wesentlich dazu bei, komplexe Quellcode-Strukturen nicht nur strukturell, sondern auch inhaltlich besser verständlich zu machen.

4.2 Erstellung mehrerer Abstract Syntax Trees

Im Rahmen der AST-basierten Analyse wird jeweils nur der Code der aktuell ausgewählten Quelldatei betrachtet. In den meisten Projekten existiert keine zentrale Datei, die alle Komponenten inkludiert, sodass ein einzelner AST, der alle Teile darstellt, nicht erstellt werden kann.

Im Verlauf der Projektarbeit ergab sich zunehmend die Anforderung, syntaktische Strukturen nicht nur aus einer einzigen Quelldatei zu extrahieren, sondern gleichzeitig zwei unterschiedliche Quelldateien in die Analyse mit einzubeziehen.

Diese Strukturen gehörten inhaltlich zwar zum Modellverhalten, wurden jedoch aus Gründen der Modularität und Übersichtlichkeit in separate Quelldateien ausgelagert. In größeren Codebasen ist es üblich, funktionale Einheiten, wie zum Beispiel spezifische Rechenfunktionen, Hilfsmethoden oder domänenspezifische Erweiterungen, in eigene Dateien auszulagern, um die Hauptkomponenten schlank und wartbarer zu gestalten. Auf diese Weise lassen sich eindeutige Zuständigkeiten innerhalb der Codebasis herstellen, wodurch sich Änderungen klarer zuordnen lassen und unbeabsichtigte Seiteneffekte reduziert werden. Gleichzeitig wird durch die Trennung vermieden, dass redundante Implementierungen entstehen, da klar definierte Komponenten mehrfach wiederverwendet werden können. In diesem Fall diente die Aufteilung auch dazu, Abhängigkeiten zu reduzieren und den Code flexibler erweiterbar zu machen. Die Auslagerung führte jedoch dazu, dass ein vollständiges Bild der Programmstruktur nur durch die Analyse mehrerer Dateien möglich war. Eine alleinige AST-Erstellung aus der `model.cpp` hätte zentrale Informationen über Funktionen, die zur Laufzeit benötigt werden, nicht berücksichtigt.

Statt alle Quelldateien in einem einzigen, zusammengeführten AST zu vereinen, wurde bewusst ein Ansatz gewählt, bei dem für jede Datei ein separater AST erzeugt wird. Ein zentraler AST hätte zwar auf den ersten Blick den Vorteil geboten, alle Informationen in einer gemeinsamen Struktur verfügbar zu machen, wäre in der Praxis jedoch mit erheblichen Nachteilen verbunden gewesen.

Zum einen hätte ein solcher AST die Modularität des Systems eingeschränkt, da alle Quelldateien und Module in einer gemeinsamen Baumstruktur repräsentiert sind. Das könnte dazu führen, das Änderungen und Erweiterungen an einem Modul sich unbeabsichtigt auf andere Teile des ASTs auswirken und bestimmte Teile des AST nicht mehr einzeln handhabbar sind. Zum anderen ist die Traversierung eines großen ASTs, insbe-

sondere bei umfangreichen Modellen mit vielen eingebundenen Quelldateien, deutlich ressourcenintensiver und weniger effizient. Clang bietet zwar die Möglichkeit, einzelne ASTs aus verschiedenen Quelldateien zu einem globalen AST zusammenzuführen, jedoch ist dieser Ansatz komplex und fehleranfällig, da Konsistenzprobleme bei Symbolen und Namensräumen entstehen können und die zentrale Struktur zusätzlichen zu den einzeln erzeugten ASTs weiteren Speicheraufwand verursacht. Außerdem ist unklar an welchen Stellen die einzelnen Bäume korrekt zusammengefügt werden sollten, das Hinzufügen neuer Dateien würde dadurch erschwert, da interne Strukturen zur konsistenten Integration bekannt sein müssten. Ein großer AST bietet gegenüber mehreren getrennten Bäumen keinen entscheidenden Vorteil, da Code-Duplikationen und Abhängigkeiten weiterhin bestehen bleiben. Ein weiterer Grund liegt darin, dass die AST-Erstellung in der Regel getrennt erfolgen muss. Ohne diese Trennung müssten manuell Ziele und Quelldateien konfiguriert werden, was bei großen Projekten kaum praktikabel wäre.

Die gewählte Architektur mit mehreren, voneinander unabhängigen ASTs fördert hingegen eine klare Trennung von Verantwortlichkeiten, eine bessere Übersichtlichkeit und ermöglicht eine parallele Erstellung ohne dass globale Abhängigkeiten entstehen würden. Dadurch wurde es unerlässlich, mehrere ASTs gleichzeitig zu erzeugen, um ein vollständiges Bild des Programmkonstrukts zu erhalten.

Zunächst wurde die AST-Erstellung seriell durchgeführt. Dabei wurde jede Quelldatei einzeln verarbeitet und für jede Datei ein entsprechender Baum erstellt. Ein wesentlicher Nachteil der seriellen Verarbeitung besteht darin, dass die Dateien nacheinander und damit zeitlich blockierend abgearbeitet werden. Das bedeutet, dass jede Datei erst dann verarbeitet werden kann, wenn die vorherige abgeschlossen ist, auch wenn keine Abhängigkeiten zwischen den Dateien bestehen.

Um die Performance der seriellen AST-Erstellung genauer zu bewerten, wurden Laufzeitmessungen an vier verschiedenen Modellen durchgeführt. Diese Modelle unterschieden sich nicht nur hinsichtlich ihrer funktionalen Komplexität, sondern auch deutlich in der Anzahl der generierten Codezeilen im jeweiligen AST. Dabei zeigte sich der Trend, dass mit zunehmender Codegröße in der Regel auch die Zeit für die AST-Erstellung zunahm.

Basierend auf diesen Laufzeitmessungen ließ sich auch ein Zusammenhang zwischen der Anzahl der verarbeiteten Quelldateien und der Gesamtzeit zur Erstellung der ASTs erkennen. Insbesondere zeigte sich, dass sich die benötigte Zeit bei serieller Verarbeitung nahezu linear mit der Anzahl der Dateien verdoppelt. Wenn die Erzeugung eines einzelnen ASTs eine bestimmte Dauer beanspruchte, verdoppelte sich die Zeit bei zwei Dateien, vorausgesetzt beide weisen vergleichbaren Umfang an Codezeilen auf. Diese linear ansteigende Verarbeitungslast stellt in kleineren Projekten zunächst kein großes Pro-

blem dar, führt jedoch bei größeren oder skalierenden Modellstrukturen zu deutlichen Effizienzeinbußen. Die Tatsache, dass alle Dateien nacheinander verarbeitet werden, bedeutet auch, dass ungenutzte Rechenressourcen nicht verwendet werden, während die Verarbeitung einer Datei abgeschlossen werden muss, bevor die nächste beginnt.

4.3 Parallelisierung der AST-Erstellung

Um die zuvor geschilderte serielle Abarbeitung effizienter zu gestalten, wurde ein Ansatz zur Parallelisierung des AST-Erstellungsprozesses gesucht. Um die verfügbaren parallelen Rechenressourcen besser auszunutzen und so die Gesamtverarbeitungsdauer bei mehreren Quelldateien zu reduzieren, wurde zunächst untersucht, welche Parallelisierungsstrategien sich für die spezifischen Anforderungen des Projekts eignen würden.

Ein naheliegender erster Ansatz zur Parallelisierung bestand in der Verwendung von Threads. Python bietet hierfür mit dem *threading*-Modul [34] eine etablierte Möglichkeit, mehrere Aufgaben gleichzeitig auszuführen. In vielen Szenarien, insbesondere bei I/O-lastigen Prozessen wie Dateioperationen oder Netzwerkzugriffen, kann die Verwendung von Threads zu Performancegewinnen führen. In einem ersten Experiment wurde daher versucht, die Erstellung mehrerer ASTs mittels `threading.Thread` parallel auszuführen.

In der Praxis stellte sich jedoch schnell heraus, dass dieser Ansatz nicht die erwarteten Verbesserungen brachte. Die tatsächliche Laufzeit bei threadbasierter Ausführung war teilweise höher als bei der sequenziellen Variante. Dies ließ sich auf eine grundlegende Eigenschaft der Python-Umgebung zurückführen, die sogenannte *Global Interpreter Lock* (GIL). Die GIL ist eine Sperre auf Interpreter-Ebene, die sicherstellt, dass zu jedem Zeitpunkt immer nur genau ein Thread Zugriff auf den Python-Interpreter hat. Selbst wenn mehrere Threads in einem Programm definiert sind, kann dadurch immer nur einer davon zur gleichen Zeit Python-Bytecode ausführen. Dies führt dazu, dass bei CPU-intensiven Operationen, wie der AST-Erstellung, bei der Parser, Tokenizer und komplexe Datenstrukturen beteiligt sind, kein echter Performancegewinn durch Multithreading erzielt werden kann.

Vielmehr müssen sich alle Threads um die Ausführung im Interpreter abwechseln, was in sogenannten *Context Switches* resultiert. Diese Umbrüche im Ausführungskontext sind nicht nur mit Overhead verbunden, sondern führen insbesondere bei rechenintensiven Aufgaben zu schlechterer Skalierbarkeit. Hinzu kommt, dass Thread-Synchronisierung bei gemeinsam genutzten Ressourcen zusätzlichen Verwaltungsaufwand mit sich bringt, ohne dass ein messbarer Geschwindigkeitsvorteil entsteht.

Im Gegensatz zur Thread-basierten Parallelisierung, bei der die GIL die gleichzeitige Ausführung mehrerer Threads im Python-Interpreter verhindert, bildet das Python-eigene *multiprocessing*-Modul [31] die Möglichkeit, Prozesse parallel auszuführen. Der zentrale Unterschied besteht darin, dass multiprocessing nicht mehrere Threads innerhalb eines Prozesses, sondern mehrere Prozesse erzeugt, von denen jeder eine eigene Python-Interpreter-Instanz sowie einen eigenen Speicherbereich besitzt. Dies umgeht die Einschränkungen durch die GIL vollständig und ermöglicht eine tatsächlich parallele Ausführung von Aufgaben. Gleichzeitig erleichtert die Nutzung separater Prozesse die Isolation von Aufgaben, wodurch Fehler in einem Prozess die anderen nicht beeinträchtigt.

Um diese parallele Verarbeitung in die bestehende Architektur zu integrieren, wurde die Struktur des PyGen-Workflows in Abbildung 3.1 überarbeitet. Hierfür wurde eine zentrale Steuereinheit in Form einer eigenen Klasse eingeführt, die unter dem Namen `ASTHandler` konzipiert wurde. Diese Komponente übernimmt die Koordination und Verwaltung der AST-Erstellung für mehrere Quelldateien, sowohl in serieller als auch in paralleler Ausführung. Dadurch konnte die Systemarchitektur so abstrahiert werden, dass die zugrunde liegende Ausführungsstrategie je nach Anzahl der zu verarbeitenden Quelldateien flexibel blieb.

Der Ablauf gliedert sich in folgende Schritte:

C++-*Target* Eingabe: Mehrere C++-Targets (C++-Target 0 bis C++-Target N) dienen als Eingabequelle. Jedes dieser Targets steht für ein definiertes Build-Ziel im Projekt und umfasst die zugehörige Komponente wie Modelle, Header- und Implementierungsdateien. Diese Targets werden im späteren Verlauf nach Python überführt. Die flexible Struktur erlaubt es, beliebig viele Build-Ziele einzubinden und somit auch wachsende Modellgrößen zu unterstützen.

Konfiguration: Die Konfiguration dient als zentrale Schnittstelle zur Übergabe relevanter Pfade und Parameter an die einzelnen Verarbeitungsschritte. Der AST-Handler nutzt diese Informationen, um den zu den jeweiligen C++-Targets gehörenden Sourcecode zu lokalisieren und in eine AST-Repräsentation zu überführen. Der Scanner greift ebenfalls auf Teile der Konfiguration zu, da er neben den AST-Daten auch Pfade zur Modulgenerierung, Build-Datenbank und verwendeter Compiler-Bibliothek benötigt. Diese Angaben sind notwendig, um den Quellcode mit den korrekten Kompilierungsparametern zu parsen, die Analyse auszuführen und die generierten Python-Module am vorgesehenen Zielort abzulegen.

Koordination durch ASTHandler: Die Verarbeitung der Quelldateien wird zentral durch die Klasse ASTHandler gesteuert. Diese Komponente dient als Steuerungseinheit für die AST-Erstellung und stellt sicher, dass jedes Target in einen eigen-

ständigen AST überführt wird. Der ASTHandler ist so konzipiert, dass er sowohl einzelne Targets allein und seriell, als auch parallele Verarbeitungsmodi bei mehreren Targets unterstützt, wobei letzterer auf dem Python-multiprocessing-Modul basiert.

Parallelisierte AST-Erstellung: Der ASTHandler überträgt die Verarbeitung jedes Targets an einen separaten Prozess. Dies ermöglicht die gleichzeitige Erstellung mehrerer ASTs (AST 0 bis AST N), wobei jeder AST unabhängig von den anderen erzeugt wird. Diese Isolation erhöht nicht nur die Verarbeitungsgeschwindigkeit, sondern verbessert auch die Fehlerrobustheit, da etwaige Fehler in einem Target die übrigen nicht beeinflussen.

AST-Ausgabe und Weiterverarbeitung: Die erzeugten ASTs werden als strukturierte Repräsentation gespeichert und stehen im Anschluss für die weitere Analyse zur Verfügung. Jeder AST wird separat geschrieben und ist somit einzeln adressierbar und erweiterbar.

Benutzerdefinierte Auswahl an Strukturen: Individuelle Strukturauswahl, wie Klassen und Funktionen, für die eine passende Python-Anbindung generiert werden soll.

Scanner: Im nächsten Schritt können alle ASTs an den Scanner übergeben werden. Dieser bildet die Schnittstelle zur Binding-Erzeugung und extrahiert alle relevanten Informationen, die durch den Benutzer angegeben werden, aus dem AST. Die Verarbeitung erfolgt modular und sequenziell, sodass immer nur ein AST analysiert wird.

Überführung der Informationen: Die im Scanner extrahierten Informationen werden in einer strukturierten Zwischendarstellung abgelegt. Diese Datenstruktur enthält alle für die weitere Verarbeitung relevanten Modellinformationen in konsolidierter Form und dient als Schnittstelle zum nachfolgenden Generierungsschritt.

Generator: Der Generator greift auf diese Zwischendarstellung zu und verwendet eine definierte Zielcode-Vorlage (Template), um daraus die entsprechenden C++-Bindings zu erzeugen. Dabei werden die Modellinformationen systematisch in syntaktisch korrekten und konsistenten Zielcode überführt.

Fertige C++-Bindings: Im finalen Schritt werden die generierten Bindings von dem Generator in eine separate Quelldatei geschrieben. Diese Datei enthält die vollständig erzeugten Bindings.

Die Einführung des ASTHandler ermöglichte eine klare Trennung zwischen der zentralen Steuerlogik und den operativen Verarbeitungsschritten. Hervorzuheben ist dabei die Fähigkeit der Klasse, dynamisch zu entscheiden, ob die Erstellung der ASTs seriell oder

parallel erfolgen soll. Basierend auf der Anzahl der in einer Liste vorliegenden Quelldateien wählt das System automatisch die geeignete Verarbeitungsstrategie, wodurch bei einzelnen Quelldateien auf die zusätzliche Komplexität von Multiprocessing verzichtet werden kann, während bei mehreren Quelldateien die Vorteile paralleler Verarbeitung genutzt werden.

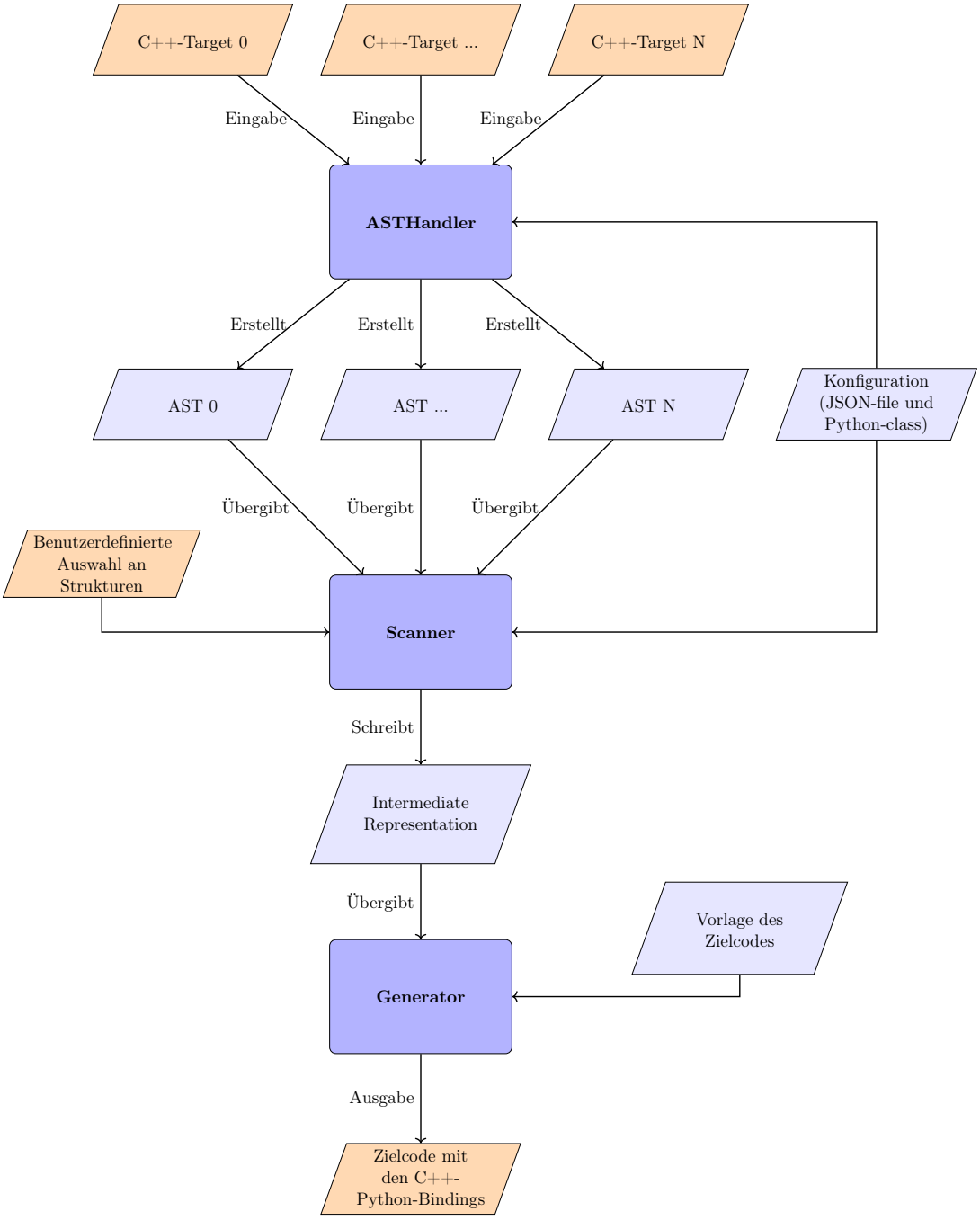

Abbildung 4.1: Ablauf der Codegenerierung nach der Parallelisierung. Die blauen Kästen stehen für Teile des Codegenerators, orangene Kästen stehen für Input und Output. Trapezförmige Kästen stellen Daten dar, abgerundete Kästen Logikklassen des Codegenerators.

Damit sind alle Änderungen im Rahmen der Parallelisierung der AST-Erstellung beschrieben. Abbildung 4.1 zeigt die vollständig überarbeitete Darstellung der PyGen-Funktionalität, auch wenn in diesem Kapitel ausschließlich die Parallelisierung behandelt wurde. Andere Änderungen, wie die Anpassungen am Scanner oder die benutzerdefinierte Auswahl an Strukturen, die nach Python überführt werden sollen, werden im Abschnitt 4.4.1 erläutert.

Der folgende Quelltext 4.1 der Methode `parallel_creation` verdeutlicht die zentrale Funktionsweise der Klasse im Kontext der parallelen Erstellung der ASTs im ASTHandler. Sie demonstriert, wie durch den Einsatz des multiprocessing-Moduls die Verteilung der Verarbeitungsaufgaben auf mehrere Prozesse realisiert wird.

Zunächst wird eine Aufgabenliste erstellt, in der jeder Eintrag die AST-Erzeugung für eine spezifische Quelldatei darstellt. Diese Aufgaben werden über die Funktion `pool.starmap()` auf die verfügbaren Prozesse verteilt, wobei jeder Prozess die Funktion `create_ast` mit der Konfiguration sowie dem Pfad zu einer Quelldatei aufruft. Die AST-Erstellung erfolgt damit unabhängig und isoliert in separaten Prozessen. Nach der parallelen Ausführung werden die erzeugten Ergebnisse gesammelt und durchlaufen eine weitere Verarbeitungsstufe. Für jeden AST-Pfad `ast_path` und jedes AST-Objekt `ast` wird ein eigener Verarbeitungsschritt mittels `process_ast_file` aufgerufen, dessen Resultat, der abschließend verarbeitete AST, in die interne AST-Verwaltungsliste aufgenommen wird. Fehler bei der Verarbeitung der einzelnen Schritte werden dabei gezielt abgefangen und protokolliert, ohne den Gesamtprozess zu unterbrechen.

Ein zentrales technisches Hindernis im Zusammenhang mit der Nutzung von Python multiprocessing bestand in der Notwendigkeit, Daten zwischen unabhängigen Prozessen zu übertragen, ein Vorgang, der eine Serialisierung der zu verarbeitenden Objekte erfordert. Python verwendet hierfür das *Pickling*-Modul [35], bei dem Objekte in eine transportable Byterepräsentation überführt werden, um in anderen Prozessen wiederhergestellt werden zu können. Dies setzt voraus, dass alle beteiligten Objekte *picklable*, also für das Pickle-Protokoll geeignet sind.

Die ursprüngliche Implementierung der AST-Erstellung war für den sequentiellen Gebrauch konzipiert und enthielt mehrere implizite Abhängigkeiten im Kontrollfluss sowie unstrukturierte Zugriffspfade innerhalb der Objektstruktur. Zwar war der generierte AST prinzipiell nicht inkompatibel mit dem Pickling-Prozess, jedoch führten bestimmte ineinander verschachtelte Funktionsaufrufe und die enge Kopplung an kontextabhängige Variablen dazu, dass eine direkte Serialisierung scheiterte. Um die Kompatibilität mit multiprocessing herzustellen, wurde daher keine vollständige Neuentwicklung der AST-Logik vorgenommen. Stattdessen wurde der bestehende Code modularisiert und gezielt in eigenständige, klar definierte Hilfsmethoden ausgelagert.

```
Algorithm PARALLEL_CREATION(config);
Input : Configuration
Result : List of ASTs
try
    results ← pool_starmap(create_ast, (config, source_file) for each
     source_file ∈ source_files);
    foreach (ast_path, ast) ∈ results do
        try
            processed_ast ← process_ast_file(ast_path, ast);
            add_ast_to_list(processed_ast[1]);
        end
        catch
            error
        end
        log "Error message for ast file";
    end
end
catch
    error
end
log "Error message for parallel creation";
```

Quelltext 4.1: Funktion zur parallelen AST-Erstellung als Pseudocode.

Durch diese Umstrukturierung konnten ungewollte Nebenwirkungen eliminiert und die Objektstruktur so weit vereinfacht werden, dass alle relevanten Instanzen ohne zusätzliche Serialisierungsmechanismen picklable wurden. Die explizite Trennung in einzelne Verarbeitungsschritte macht nicht nur die Datenflüsse transparenter, sondern führte auch dazu, dass komplexe Methodenaufrufe durch primitive Datentypen ersetzt werden konnten, die problemlos zwischen Prozessen übertragbar sind.

Diese vergleichsweise einfache, aber wirkungsvolle Maßnahme ermöglicht es, die bestehende AST-Erstellung direkt in parallele Verarbeitungsstrukturen zu integrieren, ohne tiefgreifende Eingriffe in das Datenmodell selbst. Gleichzeitig wurde damit die Grundlage für eine stabilere Architektur geschaffen, in der Erweiterungen, Analyse oder Validierungsschritte unabhängig voneinander und in separaten Prozessen ausgeführt werden können. Pickling war in diesem Kontext nicht nur ein technisches Erfordernis, sondern ein Katalysator für eine saubere Trennung von Zuständigkeiten und für die weitere Modularisierung der Codebasis.

Die Umstellung auf das neue System mit dem zentralen ASTHandler-Modul und der konsequenten Nutzung paralleler Verarbeitungsstrategien bringt eine Reihe signifikanter Vorteile mit sich. Anstelle einer sequentiellen Abarbeitung einzelner Quelldateien wird nun ein koordinierter, parallelisierter Prozess verwendet, der die verfügbaren Rechenressourcen effizienter ausschöpft.

Darüber hinaus fördert die Einführung des Moduls eine klare Trennung der Verantwortlichkeiten innerhalb der Architektur. Die gezielte Zusammenführung der Steuerlogik für die Erzeugung der ASTs in einer dedizierten Komponente erleichtert sowohl die Wartung als auch zukünftige Erweiterungen, etwa im Hinblick auf zusätzliche Analyseoptionen oder alternativen AST-Quellen. Durch die strikt modulare Struktur bleibt das System flexibel und erweiterbar, ohne dabei an Effizienz zu verlieren.

4.4 Generalisierung des Scanners

Mit der in den vorangegangenen Abschnitten dargestellten Parallelisierung der AST-Erstellung sowie der strukturellen Modularisierung im ASTHandler wurde die Grundlage für eine skalierbare und effizient wartbare Architektur geschaffen. Im nächsten Schritt galt es nun, auch den Scanner, als zentrale Komponente zur Analyse und Extraktion der Informationen aus den erzeugten ASTs, von manuellen Prozessen zu entkoppeln und in eine verallgemeinerte Struktur zu überführen.

In der ursprünglichen Implementierung war der Scanner eng an eine manuelle Vorgehensweise zur Analyse der erzeugten ASTs gebunden. Der gesamte Scanprozess beruhte auf einer stark spezifischen und wenig abstrahierten Logik. Für jede neue Funktion, die über das System angebunden werden sollte, musste zunächst der entsprechende Knoten im AST manuell identifiziert und analysiert werden. Diese Lokalisierung erforderte genaue Kenntnisse über die Struktur des jeweils erzeugten ASTs sowie über dessen interne Repräsentation, was den Entwicklungsprozess verlangsamte.

Sobald die Position der Zielfunktion im AST bestimmt war, folgt in einem weiteren Schritt die Implementierung einer eigenen, oft nur einmal verwendeten Extraktionsmethode. Diese Methode war direkt auf die jeweilige Struktur der Funktion abgestimmt und beinhaltete typischerweise eine fest kodierte Extraktion der relevanten Informationen, beispielsweise Funktionsnamen, Rückgabetypen oder Parameter. Diese Informationen wurden anschließend direkt an ein ebenso spezifisches Binding-Template weitergegeben, die den generierten Code für die Anbindung der Funktion formuliert. Auch hier war der Code stark auf das jeweilige Beispiel zugeschnitten und ließ sich kaum wiederverwenden.

Das resultierende System war dadurch nicht nur wartungsintensiv, sondern auch fehleranfällig. Änderungen in der zugrundeliegenden AST-Struktur oder der Funktion führten dazu, dass bestehende Extraktionsmethoden nicht mehr funktionierten oder unerwartetes Verhalten zeigten. Zudem führte der Bedarf, für jede Funktion eine eigene Extraktions- und Bindinglogik zu schreiben, zu einem erheblichen Maß an redundantem Code. Immer wieder wurden ähnliche Strukturen verarbeitet, jedoch in separaten Methoden mit abweichender, aber im Kern identischer Logik.

Die zentrale Zielsetzung der Generalisierung bestand darin, die manuell geprägten Abläufe der AST-Analyse und der Generierung der Bindings durch ein systematisches, deklaratives Verfahren zu ersetzen. Im Fokus stand dabei die Reduktion repetitiver Programmierarbeit sowie die Minimierung potenzieller Fehlerquellen, wie sie in der bisherigen, eng gekoppelten Struktur auftraten. Statt für jede neu hinzukommende Funktion eigene Such-, Analyse- und Generierungslogik in Form von Einzelfalllösungen zu entwickeln, sollte ein Framework geschaffen werden, das diese Aufgaben weitgehend automatisch und auf Basis zentral definierter Regeln ausführt.

Ein erster Kernaspekt der Zielsetzung war die Reduktion manueller Eingriffe in den Analyseprozess. Entwickler sollten nicht länger gezwungen sein, AST-Knoten einzeln zu identifizieren und darauf abgestimmte Extraktionsmethoden zu formulieren. Stattdessen sollte es möglich sein, gesuchte Strukturen deklarativ zu beschreiben und die weitere Verarbeitung automatisiert auszulösen. Auf diese Weise sollte nicht nur der Entwicklungsaufwand reduziert, sondern auch die Einstiegshürde für die Erweiterung des Systems gesenkt werden.

Eng damit verknüpft war die angestrebte Vereinheitlichung des Analyseprozesses. Während zuvor jede Funktion ihren eigenen Codepfad durchlief, sollte fortan ein einheitlicher Mechanismus zur Erkennung, Extraktion und Weiterverarbeitung sämtlicher relevanter Strukturen im AST etabliert werden.

Ein weiterer Bestandteil lag in einer automatischen Weiterverarbeitung der extrahierten Informationen. Die Trennung zwischen der Erkennung von AST-Knoten und der nachfolgenden Generierung von Bindings sollte so gestaltet werden, dass eine einmal identifizierte Struktur direkt an ein universelles Binding-Template-System übergeben werden kann. Dieses sollte es dann ermöglichen, aufgrund von extrahierten Merkmalen die notwendigen Codebausteine zu erzeugen, ohne dass dafür zusätzliche handgeschriebene Bindings nötig sind.

Schließlich war die Erweiterbarkeit ein zentrales Ziel. Die neue Verarbeitung sollte so aufgebaut sein, dass neue Strukturen mit minimalem Aufwand integriert werden können. Die Generalisierung verfolgte daher von Beginn an einen modularen Ansatz, der zukünftige Anpassungen durch einfache Erweiterungen bestehender Strukturen ermöglicht.

Diese Problemstellungen und Zielsetzungen verdeutlichen den Anspruch, den Scanner aus einer starren, manuell gepflegten Sammlung spezifischer Methoden zu einer flexiblen, erweiterbaren Analysewerkzeug zu transformieren. Durch die klare Trennung der Beschreibung der zu suchenden Strukturen und deren automatisierter Weiterverarbeitung

sollte eine robuste Grundlage geschaffen werden, auf der sich unterschiedlichste Sprachkonstrukte effizient erfassen und verarbeiten lassen. Im Folgenden wird dargelegt, wie diese Prinzipien konkret umgesetzt wurden und welche strukturellen Anpassungen im Scanner vorgenommen wurden, um die angestrebte Generalisierung zu realisieren.

4.4.1 Identifikation und Extraktion relevanter Strukturen

Im Zentrum der Generalisierung des Scanners steht die Einführung eines zentralen Dictionaries, welches die bisher verstreuten und manuell gepflegten Informationen zur Zielstruktur im AST in einer einheitlichen, deklarativen Form zusammenfasst. Diese Datenstruktur bildet den Ausgangspunkt für die nachgelagerte automatische Analyse und Verarbeitung der AST-Knoten. Ziel dabei ist es, nicht mehr für jede einzelne Funktion oder Klasse eigenen, hartkodierten Analysecode schreiben zu müssen, sondern stattdessen die relevanten Strukturen vorab in einer Liste zu definieren.

Ein typischer Eintrag in diese Konfigurationsstruktur beschreibt die zu suchende Entität anhand ihrer Eigenschaften, wie etwa Typ (z.B. `function` oder `class`), Name und optional weitere Merkmale wie `cursorkind`. Ein einfaches Beispiel für einen solchen Eintrag wird in Quelltext 4.2 beschrieben.

```
1  general_bindings_dict = [
2      {
3          "type": "function",
4          "name": "draw_sample",
5          "cursorkind": "FUNCTION_TEMPLATE"
6      }
7  ]
```

Quelltext 4.2: Eintrag in das Dictionary `general_bindings_dict`.

Dieser Eintrag beschreibt eine Zielstruktur vom Typ `function` mit dem Namen `draw_sample`, die im Clang-AST als `FUNCTION_TEMPLATE` repräsentiert wird. Statt nun im Code festzulegen, wo und wie draw_sample im AST zu finden ist, reicht die Definition dieses Eintrags aus, die Such- und Extraktionslogik wird später generisch darauf angewendet. Diese zentrale Datenstruktur bietet gleich mehrere Vorteile. Zum einen wird durch die Standardisierung der Einträge eine saubere Trennung zwischen den Daten, also dem, was gefunden werden soll, und der Logik, und wie sie verarbeitet werden soll, geschaffen. Zum anderen erlaubt die einheitliche Beschreibung eine wesentlich einfachere Benutzung und Erweiterung. Wenn eine neue Funktion oder Klasse hinzugefügt werden soll, genügt es, wenn der Benutzer einen neuen Eintrag in das Dictionary schreibt, ohne jegliche Änderung an der zugrundeliegenden Logik vorzunehmen.
Die Struktur des Dictionaries wurde zudem so gewählt, dass sie sich problemlos um zusätzliche Merkmale oder Spezifikationen erweitern lässt.

Diese Konfigurationsliste bildet somit den Einstieg in die neue Scanner-Architektur. Sie ersetzt das frühere manuelle Mapping zwischen AST-Struktur und Binding-Code durch eine zentrale, deklarative Beschreibung und schafft so die Voraussetzung für die vollständig automatisierte AST-Verarbeitung im weiteren Verlauf.

Ein zentrales Element der Scanner-Architektur ist die Einführung einer allgemeinen Extraktionsfunktion, die dafür verantwortlich ist, aus den beim Traversieren identifizierten relevanten AST-Knoten Informationen zu gewinnen. Dabei sind die Knoten relevant, die vorher im Dictionary definiert wurden. Diese Funktion bildet das Bindeglied zwischen der rein strukturellen Erkennung eines Knotens im AST und der semantisch verwertbaren Repräsentation für die spätere Codegenerierung.

Zuvor musste dieser Schritt für jede einzelne Funktion oder Klasse separat implementiert werden. Die Extraktion war dabei jeweils direkt an die Struktur des zu analysierenden Codes gekoppelt. In der neuen generalisierten Architektur übernimmt nun eine einheitliche Funktion diese Aufgabe, unabhängig davon, ob es sich um eine freie Funktion, eine Klassenmethode oder eine komplexere, geschachtelte Struktur handelt. Der entscheidende Vorteil liegt darin, dass die Erkennung nicht länger von spezifischen Namen oder Implementierungen abhängt. Statt für jede einzelne Funktion oder Klasse eigene Anpassungen vorzunehmen, genügt es, lediglich neue C++-Sprachkonzepte zu berücksichtigen.

Die Extraktionsfunktion wird innerhalb der Traversierungsmethode auf jeden besuchten Knoten angewendet. Ihre Aufgabe besteht nicht nur darin, Informationen aus einem gegebenen Knoten zu extrahieren, sondern auch darin, zu prüfen, ob der jeweilige Knoten überhaupt relevant ist. Dazu vergleicht sie systematisch die Merkmale des aktuellen Knotens mit den in der `general_bindings_dict` definierten Zielen. Nur wenn ein Match vorliegt, erfolgt die Extraktion der Informationen. Diese Kombination aus Identifikation und Extraktion ermöglicht eine kompakte, modular aufgebaute Logik. Die von der Funktion extrahierten Informationen werden dann in einem Dictionary `found_info` gespeichert, welches dann einer Liste von Dictionaries `found_bindings` hinzugefügt wird. Diese Dictionaries enthalten folgende Informationen zu einem Knoten:

- `type` - Gibt den allgemeinen Strukturtyp an, zum Beispiel `function` oder `class`. Dient der Zuordnung zur späteren Verarbeitung.

- `name` - Identifier der Funktion oder Klasse im Quellcode.

- `cursokind` - Gibt an, um welche konkreten Clang-AST-Knotenart es sich handelt, was für die Unterscheidung relevanter strukturen essenziell ist.

- `namespace` - Gibt an, in welchem C++-Namespace sich die Funktion oder Klasse befindet. Wichtig für die korrekte Namensauflösung.
- `return_type` - Der Rückgabetyp einer Funktion, relevant für die Signatur beim Generieren des Bindings.
- `arg_type` - Typ der Argumente, die zur Erstellung der Funktionssignatur benötigt werden.
- `arg_name` - Name der Argumente, die zur Erstellung der Funktionssignatur benötigt werden.
- `parent_name` - Name der übergeordneten Struktur.
- `is_const` - Gibt an, ob die Methode `const` ist, entscheidet bei der Erstellung von Wrappern für konstante Memberfunktionen.
- `is_member` - Markiert, ob es sich um eine Klassenmethode handelt.
- `methods` - Wird bei Klassenstrukturen verwendet, um auch deren Methoden abzubilden und zu verarbeiten.

Durch diese systematische Extraktion der relevanten Merkmale entsteht eine strukturierte Zwischenrepräsentation, die als zentrale Datengrundlage für alle weiteren Schritte dient. Unabhängig davon, ob es sich um eine freie Funktion, eine Klassenmethode oder eine Template-Definition handelt, liegen nun alle relevanten Informationen in einheitlicher Form vor. Dies schafft die Voraussetzung für eine generische Weiterverarbeitung der Informationen.

Auch für zukünftige Erweiterungen stellt dieser Ansatz einen klaren Vorteil dar. Neue Strukturen können durch Ergänzungen innerhalb der Extraktionslogik unterstützt werden, ohne dass bestehende Komponenten angepasst werden müssen. Die Extraktionslogik selbst kann über zusätzliche Prüfung von Merkmalen oder Filtern flexibel an neue Anforderungen angepasst werden.

4.4.2 Verarbeitung der extrahierten Daten

Nach der erfolgreichen Identifikation und Extraktion relevanter Strukturen in der Zwischenrepräsentation, ist der nächste zentrale Schritt die Generierung der Bindings. Hier wird beschrieben, wie die zuvor gewonnenen Daten weiterverarbeitet werden, um daraus zielgerichtet Code mittels Template-Strukturen zu erzeugen.
Ziel dieser Phase ist es, aus den allgemeinen, generisch strukturierten Informationen konkrete Bindings in Python zu erzeugen. Dabei werden die im Extraktionsschritt ermittelten Informationen wie Funktionsnamen, Rückgabetypen, Argumente oder Klassenzugehörigkeiten systematisch in Binding-Templates eingefügt.

Diese Templates sind Textbausteine, die durch Platzhalter ergänzt werden und so eine klare Trennung zwischen Inhalt und Darstellung ermöglichen. Abhängig von Typ und Kontext der extrahierten Struktur wie zum Beispiel freie Funktionen, Methoden oder Klassen, werden die passenden Templates ausgewählt und mit den Informationen befüllt. Die Templates fungieren somit als Bindeglied zwischen der AST-basierten Analyse und der konkreten Generierung von Bindings.

Die folgenden Abschnitte beschreiben:

- wie die Auswahl des passenden Template erfolgt,
- den Aufbau der Templates an einem Besipiel,
- wie durch diese automatisierte Vorgehensweise ein erweiterbares und wartbares System entsteht.

Zur Auswahl der passenden Templates wird die Funktion `handle_all_bindings` aufgerufen. Ziel dieser Funktion ist es, auf Basis der Zwischendarstellung `Intermediate-Representation` für jedes erkannte Element, das passende Template aufzurufen. Die Funktion iteriert hierzu über alle in `found_bindings` enthaltenen Einträge. Jeder Eintrag entspricht einem vorher im AST identifizierten Knoten. Die Entscheidung, welches Template angewendet wird, erfolgt dabei anhand des allgemeinen Typs des Elements `type` und bei Klassen der genaueren Unterscheidung über den `cursorkind`.
Die Funktion prüft zunächst, ob es sich bei einem Eintrag um eine Klasse handelt. Falls dies der Fall ist, wird über den `cursorkind` weiter differenziert, um Template-Klassen gesondert zu behandeln. Für reguläre Klassen wird die Funktion `bind_classes` aufgerufen, während Template-Klassen über die Funktion `template_class_wrapper` verarbeitet werden.

Funktionen hingegen werden direkt über die Funktion `bind_functions` an das entsprechende Template übergeben. Weitere Unterscheidungen, wie etwa überladene Funktionen, werden anschließend innerhalb der jeweiligen Funktion entschieden. Diese klare Trennung erlaubt nicht nur eine gezielte Auswahl der Templates, sondern fördert auch die Erweiterbarkeit der Architektur.

Die Funktion `handle_all_bindings` übernimmt somit eine zentrale Steuerungsrolle in der Template-Auswahl. Sie stellt sicher, dass die extrahierten Daten in den jeweils richtigen Kontext überführt werden und bildet die Brücke zwischen Analyse und Codegenerierung.

Ist ein Eintrag in der Liste `found_bindings` vom Typ `function`, so wird dieser an die Funktion `bind_functions` übergeben. Diese übernimmt die Aufgabe, aus den zuvor

extrahierten Informationen einen konkreten Codeausschnitt für das gesamte Binding zu erzeugen. Dabei orientiert sich die Funktion an einem festen Verarbeitungsschema, das flexibel auf unterschiedliche Ausprägungen von Funktionen reagiert.
Im Quelltext 4.3 wird die Funktionsweise der Funktion `bind_functions` erklärt.

```
1   def bind_functions(intermed_repr: IntermediateRepresentation,
        bindings: dict, class_name: str = "") -> str:
2
3       result = ""
4       name = bindings.get("name", "")
5       namespace = bindings.get("namespace", "")
6       arg_types = ensure_list(bindings.get("arg_types", []))
7       arg_names = ensure_list(bindings.get("arg_names", []))
8       py_args = ",".join(f'py::arg("{n}")' for n in arg_names)
9       is_member = bindings.get("is_member", False)
10      parent_name = bindings.get("parent_name", "")
11      scalartype = ScalarType(intermed_repr)
12      prefix = set_prefix(bindings)
13
14      if is_overloaded_function(bindings, intermed_repr.
            found_bindings):
15
16          result += overloaded_function_bindings(
17              name, namespace, scalartype, arg_types, py_args,
                    bindings, prefix)
18
19      elif needs_lambda_binding(bindings):
20
21          result += lambda_binding(name, namespace, scalartype,
                class_name)
22      else:
23          result += direct_binding(
24              name, namespace, scalartype, py_args, prefix, bool(
                    is_member), parent_name)
25
26  return result
```

Quelltext 4.3: Funktion `bind_functions` zur Auswahl geeigneter Templates für dass generieren einer Funktion.

Zunächst werden die relevanten Eigenschaften der Funktion aus dem übergebenen Dictionary ausgelesen, darunter der Funktionsname, der Namespace, die Argumenttypen sowie mögliche Zusatzinformationen wie der Klassenkontext oder die Möglichkeit, dass die Funktion eine Klassenmethode ist. Diese Informationen bilden die Grundlage für die Auswahl des geeigneten Templates. Im nächsten Schritt entscheidet die Funktion, wie die Bindings konkret generiert werden sollen. Dazu werden spezifische Bedingungen geprüft.

- Bei überladenen Funktionen wird auf das `overloaded_functions_binding`-Template zurückgegriffen. Dies erlaubt es, mehrere Funktionssignaturen innerhalb eines Bindings zusammenzuführen.
- Funktionen, die sich nicht direkt binden lassen, etwa aufgrund komplexer Signaturen oder C++-Spezifikationen, werden mittels `lambda_binding` in eine Lambda-Funktion verpackt, um sie dennoch Python-kompatibel verfügbar zu machen.
- Standardfunktionen ohne Besonderheiten werden über das `direct_binding`-Template verarbeitet.

Nach der Auswahl des geeigneten Templates wird dieses aufgerufen, um den finalen Code zu generieren. Im Quelltext 4.4 wird das Template der `direct_binding` Funktion erklärt.

```
1   def direct_binding(name: str, namespace: str, scalartype: str,
        py_args: str, prefix: str, is_member: bool, parent_name: str)
         -> str:
2
3       if not is_member:
4           return (
5               f'\t{prefix}.def('
6               f'\n\t\t"{name}",'
7               f'\n\t\t&{namespace}{name}<{scalartype}>,'
8               f'\n\t\t{py_args}\n\t);\n\n'
9           )
10      else:
11          return (
12              f'\t{prefix}.def("{name}", &{namespace}{parent_name
                    }<{scalartype}>::{name})\n'
13          )
```

Quelltext 4.4: Template der `direct_binding` Funktion für dass generieren des Codes für eine Funktion.

Diese Funktion erzeugt eine Zeichenkette, die den finalen Binding-Code enthält. Dabei unterscheidet die Funktion, ob es sich um eine freie Funktion handelt oder um eine Klassenmethode. Der Unterschied bei der Generierung liegt dabei im `prefix`, dieser verschwindet wenn die Funktion Teil einer Klasse ist und gibt sonst an, an welches pybind11-Modul oder Objekt die `.def`-Anweisung gehängt wird.

Durch diese automatisierte Vorgehensweise entsteht eine äußerst flexible und erweiterbare Architektur. Anstatt für jede einzelne Funktion oder Klasse hartkodierte Bindings zu schreiben, stützt sich die Codegenerierung nun auf zuvor extrahierten Strukturen. Mithilfe von Templates werden daraus automatisch die passenden Bindings erzeugt, die sich für jede konkrete Instanz der jeweiligen Struktur wiederverwenden lassen.

Änderungen oder Erweiterungen im zugrundeliegenden C++-Code wie neue Funktionen oder Klassen müssen somit lediglich durch eine Ergänzung im `general_bindings_dict` berücksichtigt werden. Der eigentliche Generierungsvorgang bleibt von solchen Änderungen unberührt. Auf diese Weise fördert der gewählte Ansatz eine schnelle Integration neuer Features, ohne bestehende Komponenten anzupassen.

Kapitel 5

Experimentelle Ergebnisse

In diesem Kapitel werden die im Rahmen dieser Arbeit entwickelten Konzepte und Methoden anhand praktischer Experimente evaluiert. Ziel ist es, die Funktionalität und Leistungsfähigkeit des implementierten Systems zu überprüfen und die Auswirkungen der zuvor beschriebenen Generalisierung des Scanners auf den Gesamtprozess der Codegenerierung zu analysieren.

Hierzu werden zunächst die Versuchsbedingungen beschrieben, um die Nachvollziehbarkeit der Ergebnisse sicherzustellen. Anschließend werden die gewonnenen Messwerte und Beobachtungen ausgewertet und mit den Zielen der Arbeit verglichen. Dabei liegt der Fokus sowohl auf der qualitativen Bewertung, etwa in Bezug auf die Korrektheit der erzeugten Bindings, als auch auf der Bewertung der Leistungsfähigkeit der Parallelisierung.

5.1 Experimenteller Aufbau

Der erste Punkt des experimentellen Aufbaus behandelt die Parallelisierung der AST-Erstellung. Ziel des Experiments ist es, die Auswirkungen der Parallelisierung auf die Laufzeit der AST-Erstellung zu untersuchen. Dazu werden die Ausführungszeiten verschiedener Modelle gemessen und mit der Anzahl der generierten Zeilen im ASTs in Beziehung gesetzt, um anschließend die Effizienz der parallelen im Vergleich zur seriellen Erstellung zu bewerten.

Ein weiterer zentraler Schwerpunkt des experimentellen Aufbaus liegt in der Bewertung der vom generalisierten Scanner extrahierten Daten sowie der daraus resultierenden Bindings. Dazu werden die vom Scanner extrahierten Daten herangezogen, um die daraus generierten Bindings auf Korrektheit und Vollständigkeit zu überprüfen.

Um die Auswirkungen der Parallelisierung genauer zu erfassen, wird die zentrale Fragestellung wie folgt formuliert: In welchem Maße kann durch die gleichzeitige Erstellung

mehrerer ASTs eine Reduktion der Verarbeitungszeit erzielt werden, und wie unterscheidet sich diese im Vergleich zur seriellen Erstellung?

Für jeden Testlauf wird die vollständige Laufzeit der AST-Erstellung erfasst. Um ein möglichst präzises Bild der Laufzeitverteilung zu erhalten, werden vier statistische Kennwerte erhoben:

- **Minimum** (min): Kürzeste gemessene Ausführungszeit eines Durchlaufs
- **Maximum** (max): Längste gemessene Ausführungszeit eines Durchlaufs
- **Mittelwert** (avg): Arithmetischer Durchschnitt aller Messwerte
- **Median**: Zentralwert der Laufzeitverteilung

Diese Werte liefern nicht nur einen Einblick in die durchschnittliche Leistungsfähigkeit der Parallelisierung, sondern auch in die Schwankungsbreite der Messungen. Insbesondere der Median gibt Aufschluss darüber, ob einzelne Ausreißer das Gesamtbild verzerren.

Zusätzlich zur Laufzeitmessung wird für jeden Testlauf die Anzahl der im AST enthaltenen Zeilen ermittelt. Diese Größe wird als Indikator für den Umfang des zu verarbeitenden Codes angenommen und dient somit als Vergleichsmaßstab für den Verarbeitungsaufwand. Durch die Gegenüberstellung dieser beiden Werte lässt sich untersuchen, ob die Laufzeit annähernd linear mit der AST-Größe skaliert oder ob Abweichungen auftreten.

Aus dem Versuchsaufbau ergeben sich zwei zentrale Erwartungshaltungen. Zum einen wird eine Laufzeitreduktion durch Parallelisierung angenommen, da die gleichzeitige Erstellung mehrerer ASTs die Gesamtdauer im Vergleich zur seriellen Ausführung verkürzen sollte. Zum anderen wird ein linearer Zusammenhang zwischen der Anzahl der AST-Zeilen und der Zeit zur AST-Erstellung erwartet. Dies bedeutet, dass sich die benötigte Erstellungszeit proportional zur Größe des ASTs verhält, sodass ein doppelt so umfangreicher AST in etwa die doppelte Zeit beansprucht.

Der Ablauf der Messungen gestaltet sich wie folgt. Zunächst erfolgt die Vorbereitung der Testdaten, bei der mehrere Codebasen unterschiedlicher Größe ausgewählt werden, aus denen anschließend die jeweiligen ASTs generiert werden. Im ersten Schritt wird eine serielle Messung durchgeführt, bei der sämtliche ASTs nacheinander verarbeitet werden. Im Rahmen der Messungen werden die Laufzeiten erfasst. Jedes Modell wird dabei zehnmal getestet, sodass zufällige Schwankungen reduziert und ein stabiler Durchschnittswert der Laufzeiten bestimmt werden kann. Anschließend wird eine Korrelation mit der AST-Zeilenanzahl hergestellt, um den Zusammenhang zwischen Laufzeit und Umfang des ASTs zu dokumentieren. Darauf folgt die parallele Messung, bei der die

gleichen ASTs unter Einsatz mehrerer Prozesse erstellt werden. Dabei wird ein und dieselbe Codebasis zweimal erstellt. Auch hier werden alle relevanten Messwerte erhoben. Abschließend werden in einer Vergleichsauswertung die Ergebnisse aus serieller und paralleler Verarbeitung gegenübergestellt, um die erzielten Effizienzgewinne zu ermitteln.

Zur Validierung der vom Scanner erfassten Daten wird ein manueller Prüfprozess durchgeführt. Dabei werden dem Scanner gezielt einzelne, klar abgegrenzte Programmstrukturen wie Funktionen oder Klassen als Eingabe übergeben. Nach der Ausführung werden die extrahierten Daten eingehend analysiert und mit den Informationen verglichen, die im entsprechenden AST hinterlegt sind. Dieser Abgleich dient dazu, die Vollständigkeit und Korrektheit der vom Scanner erfassten Informationen zu überprüfen und potenzielle Fehlinterpretationen oder Auslassungen zu identifizieren. Insbesondere wird darauf geachtet, ob alle relevanten Strukturelemente korrekt erkannt und abgebildet wurden, um eine möglichst fehlerfreie Weiterverarbeitung zu gewährleisten. Ein Risiko dieser Vorgehensweise besteht jedoch darin, dass die gewählten Beispiele die Implementierung besonders gut widerspiegeln und dadurch mögliche Fehler unentdeckt bleiben können.

5.2 Statistische Auswertung

Im Folgenden werden die erhobenen Messwerte ausgewertet, um die Leistungsfähigkeit der implementierten Parallelisierung zu beurteilen. Für die Evaluation wurden dabei vier Modelle auf Basis gewöhnlicher Differentialgleichungen (engl. Ordinary Differential Equation, ODE) ausgewählt. Die Modelle *ode_sir*, *ode_seir*, *ode_secir* und *ode_secirvvs* wurden ausgewählt, da diese eine zunehmende Komplexität aufweisen und im Vergleich zu anderen Modellen strukturell sehr ähnlich sind, wodurch eine aussagekräftige Betrachtung hinsichtlich Skalierbarkeit und Performance möglich ist. Anschließend werden die gemessenen Laufzeiten in Bezug zur Anzahl der im AST enthaltenen Codezeilen gesetzt, um mögliche Korrelationen zu analysieren.

Modell	Codezeilen	Mittelwert	Median	Maximum	Minimum
ode_sir	681579	17.44	17.45	18.77	15.99
ode_seir	752797	19.79	19.96	21.77	17.44
ode_secir	936751	26.89	26.66	30.25	24.20
ode_secirvvs	938977	26.58	26.49	29.68	24.60

Tabelle 5.1: Codezeilenanzahl und Laufzeiten der Modelle in Sekunden.

Die Tabelle 5.1 fasst die Laufzeiten und die Anzahl der Codezeilen der vier untersuchten Modelle zusammen. Die Laufzeiten sind in Sekunden angegeben und beinhalten den Mittelwert, Median sowie die Minimal- und Maximalwerte aus zehn Messdurchläufen.

Mit zunehmenden Umfang des zu verarbeitenden Codes steigt die Laufzeit für die AST-Erstellung erwartungsgemäß an. Die gemessenen Unterschiede zwischen den Modellen lassen sich sowohl auf die unterschiedliche Codezeilenanzahl als auch auf die Komplexität der jeweiligen Strukturen zurückführen. Ein genauer Vergleich zeigt, dass das Modell ode_sir mit 681 579 Codezeilen und einer mittleren Laufzeit von 17, 44 Sekunden die schnellste Verarbeitung aufweist. Im Gegensatz dazu benötigt das Modell ode_secir mit 936 751 Codezeilen durchschnittlich 26, 89 Sekunden. Der Anstieg ist somit deutlich, lässt sich jedoch durch die höhere Anzahl zu verarbeitender Codezeilen plausibel erklären.

Auffällig ist, dass die Modelle ode_secir und ode_secirvvs trotz nahezu identischer Codezeilenanzahl leicht unterschiedliche mittlere Laufzeiten aufweisen und das kleinere Modell, in diesem Fall ode_secir, eine höhere Laufzeit aufweist. Diese Differenzen können auf subtile Unterschiede in der internen Struktur der jeweiligen Codebasis zurückzuführen sein, etwa durch variierende Anteile an komplexeren Konstrukten, die den Parsing-Aufwand beeinflussen. Da die Werte jedoch sehr nah zusammen liegen, könnten auch Messungenauigkeiten oder nicht exakt erfassbare Laufzeitunterschiede zu einem Teil der Abweichung beitragen. Der geringe Abstand zwischen Mittelwert und Median in allen Fällen deutet auf eine homogene Verteilung der Laufzeiten hin. Dies wird durch die minimalen Unterschiede zwischen Minimum und Maximum innerhalb der Modelle bestätigt. So beträgt die Schwankungsbreite beim ode_sir Modell nur etwa 2, 78 Sekunden, beim größten Modell ode_secirvvs rund 5 Sekunden. Diese geringen Streuungen sprechen für eine stabile und reproduzierbare Verarbeitung, was insbesondere bei Performance-Messungen von Bedeutung ist.

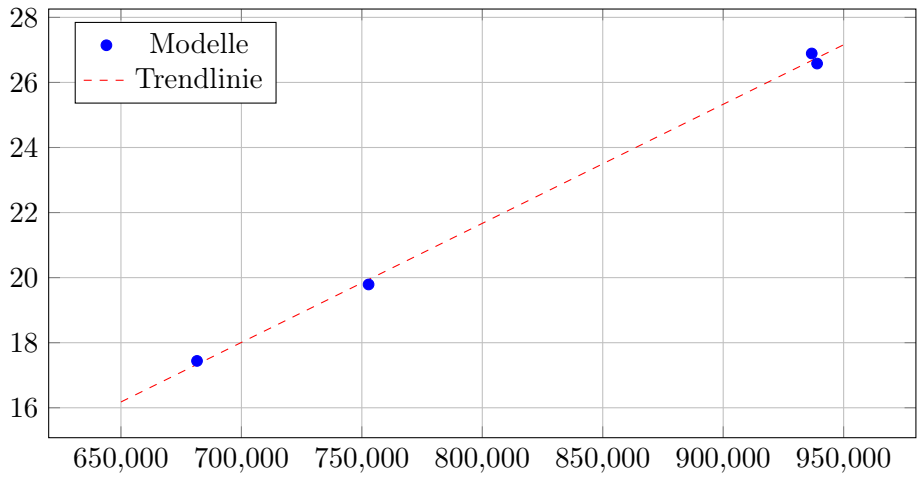

Abbildung 5.2: Zusammenhang zwischen der Anzahl der Codezeilen (x-Achse) und der mittlerer Laufzeit der AST-Erstellung (y-Achse) der Modelle. Blaue Punkte markieren die einzelnen Modelle. Die rote gestrichelte Linie zeigt die lineare Trendlinie, die den allgemeinen Zusammenhang verdeutlicht.

Werden die Daten wie in Abbildung 5.2 in Relation zur Codezeilenanzahl betrachtet, zeigt sich ein klarer positiver Zusammenhang zwischen Umfang und Laufzeit. Mit steigender Modellgröße erhöht sich auch die benötigte Verarbeitungszeit, was die erwartete Korrelation zwischen Komplexität und Aufwand bestätigt. Abweichungen, wie sie zwischen den ode_secir und ode_secirvvs Modellen erkennbar sind, lassen sich auf Parsing-spezifische Overheads oder minimale Unterschiede im Aufbau der Codebasis zurückführen. Eine exakte Charakterisierung des Zusammenhangs wurde im Rahmen dieser Arbeit jedoch nicht im Detail untersucht, da der Schwerpunkt auf der praktischen Evaluation der Generierung lag.

Im Anschluss an die serielle Messung wurde eine Laufzeitmessung durchgeführt, um den Effekt der Parallelisierung zu messen. Dazu wurde jeweils dasselbe Modell zweimal erstellt, um die erwartete Gesamtlaufzeit für zwei ASTs zu ermitteln. Die so berechnete doppelte Laufzeit dient als Referenzwert für die Bewertung der parallelen Verarbeitung. Anschließend wurden die beiden ASTs gleichzeitig mit dem entwickelten Parallelisierungsansatz erstellt, sodass der erzielte Zeitgewinn direkt ablesbar ist. Dieser Vergleich ermöglicht eine klare Aussage darüber, in welchem Maße die Parallelisierung die Gesamtdauer reduziert.

Modell	Mittelwert	Median	Maximum	Minimum
ode_sir	27.49	27.32	29.94	26.28
ode_seir	29.08	29.23	29.88	28.02
ode_secir	39.17	39.10	40.22	38.19
ode_secirvvs	39.91	39.64	42.07	38.42

Tabelle 5.3: Laufzeiten der parallelen Ausführung der Modelle in Sekunden.

Ein direkter Vergleich der in Tabelle 5.3 aufgeführten parallelen Messungen mit den erwarteten Referenzwerten aus der seriellen Ausführung zeigt den erzielten Zeitgewinn. Als Referenz wurden die in Tabelle 5.1 ermittelten mittleren Laufzeiten der seriellen Erstellung jeweils verdoppelt, da bei einer rein sequenziellen Verarbeitung zweier identischer Modelle eine nahezu lineare Skalierung der Laufzeit zu erwarten ist. Für das ode_sir Modell ergibt sich so ein Referenzwert von rund $34,88$ Sekunden. Die gemessene parallele Laufzeit liegt mit durchschnittlich $27,49$ Sekunden darunter, was einer Zeitersparnis von etwa $7,39$ Sekunden entspricht. Ein ähnliches Bild zeigt sich bei dem ode_seir Modell. Der Referenzwert von $39,58$ Sekunden wird in der parallelen Ausführung mit $29,08$ Sekunden um mehr als zehn Sekunden unterschritten. Auch bei den größeren Modellen ist der Effekt klar erkennbar. Das ode_secir Modell weist einen Referenzwert von $53,78$ Sekunden auf, während die parallele Ausführung nur $39,17$ Sekunden benötigt, was eine Reduktion von über 14 Sekunden bedeutet. Bei dem ode_secirvvs Modell reduziert sich die Laufzeit von erwarteten $53,16$ Sekunden auf $39,91$ Sekunden.

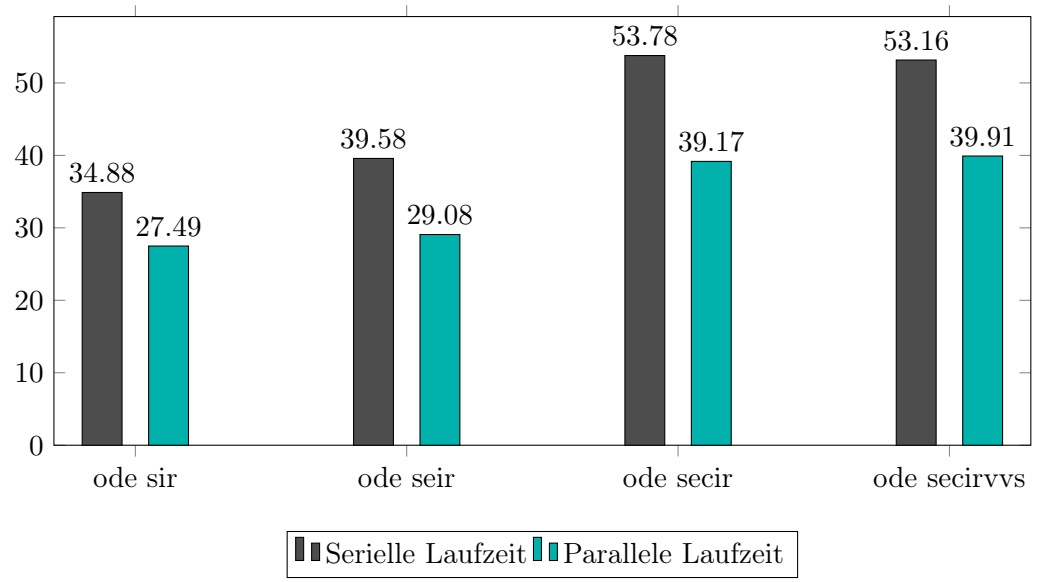

Abbildung 5.4: Vergleich der gemessenen Laufzeiten als Balkendiagramm. Links wird die Laufzeit in Sekunden gezeigt. Die grauen Balken zeigen die doppelten Laufzeiten der seriellen Messung. Die türkisen Balken Zeigen die Laufzeiten der parallelen Messung.

Abbildung 5.4 zeigt die Laufzeiten im Vergleich zur besseren Veranschaulichung der Unterschiede als Balkendiagramm.
In allen Fällen liegen die gemessenen Werte somit deutlich unter den seriellen Referenzzeiten, was die Zeitersparnis des Ansatzes zur Parallelisierung bestätigt. Außerdem ist die Abweichung zwischen Mittelwert und Median in der parallelen Messung äußerst gering, was auf eine stabile und konsistente Performance hinweist. Die maximalen Laufzeiten liegen nur geringfügig über den Medians, und auch die Minimalwerte bewegen sich nah an den Durchschnittswerten, was zusätzlich für die Robustheit des Ansatzes spricht. Allerdings könnte man angesichts der Verwendung von zwei Prozessorkernen theoretisch noch größere Verbesserungen erwarten.

5.3 Auswertung der extrahierten Informationen

Ein zentraler Aspekt dieser Arbeit ist die Analyse und Validierung der vom generalisierten Scanner extrahierten Daten sowie der darauf basierenden generierten Bindings. Einzelne Programmstrukturen dienen dabei als Testfälle, um Vollständigkeit, Konsistenz und Korrektheit der Extraktion und Generierung zu gewährleisten. Ziel der Untersuchung ist es, sicherzustellen, dass alle relevanten Strukturelemente korrekt erfasst und in den generierten Bindings konsistent abgebildet werden. Hierzu werden die generierten Bindings gezielt geprüft.

Im ersten Schritt wird die zu testende Funktion oder Klasse in das Dictionary `general_bindings_dict` aus Quelltext 4.2 eingetragen. Im ersten Testdurchlauf tragen wir die

Klasse `Parameters` und die folgenden Methoden in das Dictionary ein. Bei den ausgewählten Elementen handelt es sich um Klassen und Funktionen aus dem MEmilio-Projekt. Diese wurden bewusst gewählt, da sie einerseits realistische und praxisrelevante Beispiele für die Nutzung des Frameworks darstellen und andererseits eine geeignete Komplexität besitzen, um die Funktionalität der automatischen Binding-Generierung zu überprüfen. Zudem ermöglicht die Verwendung eigener Projekt-Beispiele eine bessere Kontrolle über die Testbedingungen und erleichtert das gezielte Anpassen von Strukturen für spezifische Testfälle.

```
general_bindings_dict = [
    {
        "type": "class",
        "name": "Parameters",
        "cursorkind": "CLASS_TEMPLATE",
        "methods": [
            {
                "type": "method",
                "name": "check_constraints",
                "cursorkind": "CXX_METHOD",
            },
            {
                "type": "method",
                "name": "apply_constraints",
                "cursorkind": "CXX_METHOD",
            },
            {
                "type": "method",
                "name": "get_start_commuter_detection",
                "cursorkind": "CXX_METHOD",
            },
            {
                "type": "method",
                "name": "get_end_commuter_detection",
                "cursorkind": "CXX_METHOD",
            },
            {
                "type": "method",
                "name": "get_commuter_nondetection",
                "cursorkind": "CXX_METHOD",
            }
        ]
    }
]
```

Quelltext 5.1: Eintrag der Testklasse mit ausgewählten Methoden in das Dictionary.

Die Klasse `Parameters` erweitert die Klasse `ParametersBase<FP>` und verwaltet sämtliche Parameter des Modells. Die dazugehörigen Methoden `check_constraint` und

`apply_constraint` prüfen alle Parameter auf plausible Wertebereiche, wobei `apply_constraint` zusätzlich diese Werte automatisch korrigiert. Die Methoden `get_start_commuter_detection` und `get_end_commuter_detection` liefern oder setzen den Start- oder Endzeitpunkt für die Erkennung von *Pendlern*, also Personen, die regelmäßig zwischen Regionen reisen und dadurch Infektionen verbreiten können. Die Methode `get_commuter_nondetection` liefert oder setzt den Anteil unerkannter infizierter Pendler. Da wir nun den Eintrag für das Dictionary gesetzt haben, geht es im nächsten Schritt um die korrekte Extraktion der Informationen aus dem AST. Die extrahierten Informationen werden in Tabelle 5.5 dargestellt.

Beschriftung	Wert
type	class
name	Parameters
cursorkind	CLASS_TEMPLATE
namespace	mio::osecirvvs::
return_type	”
arg_types	”
arg_names	”
parent_name	”
is_const	False
is_member	False
base_classes	ParametersBase
init	name: num_agegroups, type: AgeGroup
template_args	FP
methods	check_constraints apply_constarints get_start_commuter_detection get_end_commuter_detection get_commuter_nondetection

Tabelle 5.5: Extrahierte Informationen einer Klasse aus dem Abstract Syntax Tree.

Die Klasse `Parameters` wird im AST durch verschiedene Attribute beschrieben, die wichtige Informationen über die Struktur und Eigenschaften der Klasse liefern. Diese Eigenschaften wurden bereits in Abschnitt 4.4.1 erläutert und können auf dieses Beispiel angewendet werden.

Jede der gefundenen Methoden hält jeweils auch eine Liste mit Werten von *type* bis *is_member*. Ein Beispiel für einen Eintrag in die *methods*-Liste wird von der Tabelle 5.6 beschrieben.

Nachdem die relevanten Informationen aus dem AST extrahiert wurden, werden diese an den Generator übergeben. Dieser erstellt mithilfe allgemeiner Templates die gewünschten Bindings. Um die Korrektheit der generierten Bindings sicherzustellen, werden diese

mit den ursprünglichen AST-Informationen verglichen. Auf diese Weise lässt sich die Funktionalität des Scanners zuverlässig überprüfen.

Beschriftung	Wert
type	method
name	apply_constraints
cursorkind	CXX_method
namespace	mio::osecirvvs::
return_type	bool
arg_types	”
arg_names	”
parent_name	Parameters
is_const	False
is_member	True

Tabelle 5.6: Extrahierte Informationen einer Methode aus dem Abstract Syntax Tree.

```
1  template <class Parameters, typename FP>
2  void bind_Parameters(py::module& m, const std::string& name)
3  {
4      py::class_<mio::osecirvvs::Parameters<double>, mio::
           osecirvvs::ParametersBase<double>>(m, name.c_str());
5      .def(py::init<AgeGroup>(), py::arg("num_agegroups"))
6      .def("check_constraints", &mio::osecirvvs::Parameters<double
           >::check_constraints)
7      .def("apply_constraints", &mio::osecirvvs::Parameters<double
           >::apply_constraints)
8      .def_property(
9          "start_commuter_detection",
10         [](mio::osecirvvs::Parameters<double> const& self) ->
               double {
11         return self.get_start_commuter_detection();
12         })
13     .def_property(
14         "end_commuter_detection",
15         [](mio::osecirvvs::Parameters<double> const& self) ->
               double {
16         return self.get_end_commuter_detection();
17         })
18     .def_property(
19         "commuter_nondetection",
20         [](mio::osecirvvs::Parameters<double> const& self) ->
               double {
21         return self.get_commuter_nondetection();
22         })
23     ;
24 }
```

Quelltext 5.2: Generierte Bindings aus den Informationen des Dictionary.

Die Funktion `bind_parameters` aus Quelltext 5.2 dient dazu, die C++-Klasse `Parameters` aus dem Namespace `mio::osecirvvs::` für die Nutzung in Python über Pybind11 verfügbar zu machen. In den extrahierten Informationen aus dem AST ist `Parameters` als `CLASS_TEMPLATE`, mit der Basisklasse `ParametersBase` beschrieben. Das Binding bildet dies korrekt ab, indem sie `py::class_<Parameters, ParametersBase<double>` verwendet.

Der Konstruktor der Klasse wird in den extrahierten Informationen als `Parameters(AgeGroup num_agegroups)` angegeben. Dies entspricht exakt der Zeile in der `.def(py::init<AgeGroup>(), py::args(num_agegroups"))` beschrieben wird.
Somit ist der Konstruktor in Python verfügbar und kann mit dem benannten Argument `num_agegroups` aufgerufen werden.

Die Methoden `check_constraints` und `apply_constraints` werden ebenfalls in den Informationen aufgeführt. Im Binding werden diese Funktionen durch `.def("check_constraints", &Parameters::check_constraints)` und `.def("apply_constraints", &Parameters::apply_constraints)` korrekt abgebildet. Der Rückgabewert wird in der Zeile nicht explizit gesetzt, dieser wird implizit aus der C++-Methode übernommen.

Darüber hinaus bildet die Funktion die *Getter* `get_start_commuter_detection`, `get_end_commuter_detection` und `get_commuter_nondetection` korrekt ab. Getter werden automatisch erkannt und als Properties gesetzt, wobei für Getter-only Properties `def_property_readonly` anstelle von `def_property` verwendet werden muss, da `def_property` sowohl Getter als auch *Setter* erwartet.

Zusammenfassend zeigt der Vergleich zwischen Binding und den extrahierten Informationen, dass alle relevanten Eigenschaften wie Namespace, Basisklasse, Konstruktor, Methoden, Rückgabewerte und Template-Parameter korrekt umgesetzt wurden. Die Binding-Funktion stellt sicher, dass die Klasse `Parameters` in Python mit den zuvor deklarierten Methoden nutzbar ist und dass die C++-Struktur und Methoden korrekt abgebildet werden. Allerdings bezieht sich diese Validierung nur auf die getesteten Strukturen, gleiche Aspekte in unterschiedlichen Strukturen sollten damit auch korrekt abbildbar sein, sie garantiert dennoch nicht die Korrektheit für nicht geprüfte Klassen oder Methoden.

Kapitel 6

Zusammenfassung

In dieser Arbeit wurde ein Ansatz entwickelt, um den Prozess der Generierung von Python-Bindings für mathematisch-epidemiologische C++-Modelle schneller, generischer und wartbarer zu gestalten. Dazu wurden mehrere wesentliche Bestandteile umgesetzt. Erstens wurde eine Visualisierungskomponente eingebaut, die es ermöglicht, die Ergebnisse in den ASTs anschaulich darzustellen. Diese Komponente erleichtert sowohl die Analyse als auch die Nachvollziehbarkeit und bietet einen Mehrwert für die Validierung der Ergebnisse.
Zweitens wurde die Parallelisierung der AST-Erstellung realisiert. Durch den Einsatz paralleler Verarbeitung konnte eine Reduktion der Gesamtlaufzeit erzielt werden, da mehrere ASTs gleichzeitig erstellt werden können. Dadurch lassen sich Informationen aus verschiedenen Dateien effizient nutzen.
Drittens wurde der Scanner generalisiert, sodass nicht länger für jede zu generierende Funktion oder Klasse eine eigene Implementierung erforderlich ist. Stattdessen genügt es, die gewünschte Struktur in einem Dictionary anzugeben, woraufhin die Bindings, mithilfe der generalisierten und automatischen Template-Auswahl, generiert werden. Dadurch konnte der Entwicklungsaufwand reduziert und gleichzeitig die Erweiterbarkeit und Wartbarkeit des Systems verbessert werden.
Zusammenfassend zeigt die Arbeit, dass durch die Kombination aus Parallelisierung und Generalisierung ein leistungsfähiges Paket entsteht, das sich flexibel an neue Anforderungen anpassen lässt.

Für zukünftige Arbeiten bieten sich verschiedene Erweiterungen an. So könnten automatische Tests zur Validierung der generierten Bindings implementiert werden, um die Korrektheit noch robuster abzusichern. Darüber hinaus wäre es spannend zu untersuchen, inwieweit der entwickelte Ansatz auf andere Binding Bibliotheken oder Frameworks in anderen Anwendungsfällen übertragbar ist, um ein universell einsetzbares Binding-Framework zu schaffen.

Anhang A

Codeverfügbarkeit

Das MEmilio-Repository ist öffentlich zugänglich unter `https://github.com/SciCompMod/memilio`. Die Visualisierung der ASTs ist unter `https://github.com/SciCompMod/memilio/blob/main/pycode/memilio-generation/memilio/generation/graph_visualization.py` verfügbar, alle weiteren in der Bachelorarbeit vorgenommenen Änderungen sind unter `https://github.com/SciCompMod/memilio/tree/parallel-processing` verfügbar.

Literatur

[1] TechTarget. *Polyglot Programming*. Accessed: 17 July 2025. 2025. URL: https://www.techtarget.com/searchsoftwarequality/definition/polyglot-programming.

[2] JetBrains. *Polyglot Programming Is a Thing*. Accessed: 17 July 2025. 2024. URL: https://blog.jetbrains.com/fleet/2024/04/polyglot-programming-is-a-thing/.

[3] Dmitrii Rassokhin. "The C++ programming language in cheminformatics and computational chemistry". In: *Journal of Cheminformatics* 12.1 (Feb. 2020), S. 10. ISSN: 1758-2946. DOI: 10.1186/s13321-020-0415-y. URL: https://doi.org/10.1186/s13321-020-0415-y.

[4] Vineesh Cutting und Nehemiah Stephen. "A Review on using Python as a Preferred Programming Language for Beginners". In: 8 (Aug. 2021), S. 4258–4263. URL: https://www.researchgate.net/publication/359379004_A_Review_on_using_Python_as_a_Preferred_Programming_Language_for_Beginners.

[5] Wenzel Jakob, Jason Rhinelander und Dean Moldovan. *pybind11 – Seamless operability between C++11 and Python*. Accessed: 17 July 2025. 2017. URL: https://github.com/pybind/pybind11.

[6] J. Bicker, D. Kerkmann, S. Korf, L. Plötzke, R. Schmieding, A. Wendler, H. Zunker u. a. *MEmilio - a High Performance Modular Epidemics Simulation Software*. Accessed: 17 July 2025. 2025. URL: https://github.com/SciCompMod/memilio.

[7] Maximilian Franz Betz. *Automatische Codegenerierung für nutzerfreundliche mathematisch-epidemiologische Modelle*. Bachelorarbeit. Nov. 2022. URL: https://elib.dlr.de/190367/1/BA_Betz_2022_Automatische_Codegenerierung_Bindings.pdf.

[8] Jin Xu, Jian Yang, Wei Li, Tao Xie, Xiangyu Zhang, Hong Mei, Xiaofei Hu und Kewei Zhao. *CodeT: Learning Token-Level Code Representations with Syntax-Aware Pre-Training*. Accessed: 17 July 2025. 2023. URL: https://arxiv.org/abs/2312.00413.

[9] Manish Koshti. *Dynamic code generation in PHP using abstract syntax trees (AST)*. Accessed: 17 July 2025. 2025. URL: https://medium.com/techtrends-digest/dynamic-code-generation-in-php-using-abstract-syntax-trees-ast-c322b4b3e2e1.

[10] David Abrahams und Stefan Seefeld. *Boost.Python Reference Manual*. Accessed: 7 August 2025. 2015. URL: https://boostorg.github.io/python/doc/html/reference/index.html.

[11] Wenzel Jakob. *nanobind: tiny and efficient C++/Python bindings*. Accessed: 10 August 2025. 2022. URL: https://nanobind.readthedocs.io/en/latest/.

[12] LLVM Project. *Clang: a C language family frontend for LLVM*. Accessed: 17 July 2025. 2023. URL: https://clang.llvm.org/.

[13] LLVM Project. *The LLVM Compiler Infrastructure*. Accessed: 4 July 2025. URL: https://llvm.org.

[14] The Clang Developers. *Introduction to the Clang Abstract Syntax Tree (AST)*. Accessed 17 July 2025. LLVM Project, 2023. URL: https://clang.llvm.org/docs/IntroductionToTheClangAST.html.

[15] Felix Yamaguchi, Martin Lottmann und Konrad Rieck. "Generalized Vulnerability Extrapolation Using Abstract Syntax Trees". In: ACM, 2012, S. 359–368. DOI: 10.1145/2420950.2421003. URL: https://www.mlsec.org/docs/2012-acsac.pdf.

[16] AT&T Research. *Graphviz - Graph Visualization Software*. Accessed: 17 July 2025. 2008. URL: https://graphviz.org/.

[17] K. Gallagher, I. Bouros, N. Fan, E. Hayman, L. Heirene, P. Lamirande, A. Lemenuel-Diot, B. Lambert, D. Gavaghan und R. Creswell. "Epidemiological Agent-Based Modelling Software (Epiabm)". In: *Journal of Open Research Software* 12.1 (2024), S. 3. DOI: 10.5334/jors.449. URL: https://doi.org/10.5334/jors.449.

[18] Katharina Boguslawski, Aleksandra Leszczyk, Artur Nowak, Filip Brzęk, Piotr Szymon Żuchowski, Dariusz Kędziera und Paweł Tecmer. "Pythonic Black-box Electronic Structure Tool (PyBEST). An open-source Python platform for electronic structure calculations at the interface between chemistry and physics". In: *Computer Physics Communications* 264 (Juli 2021), S. 107933. ISSN: 0010-4655. DOI: 10.1016/j.cpc.2021.107933. URL: http://dx.doi.org/10.1016/j.cpc.2021.107933.

[19] Robert C. Kirby und Lawrence Mitchell. "Code Generation for Generally Mapped Finite Elements". In: *ACM Trans. Math. Softw.* 45.4 (Dez. 2019). ISSN: 0098-3500. DOI: 10.1145/3361745. URL: https://doi.org/10.1145/3361745.

[20] Dominic Kempf, René Heß, Steffen Müthing und Peter Bastian. "Automatic Code Generation for High-performance Discontinuous Galerkin Methods on Modern Architectures". In: *ACM Trans. Math. Softw.* 47.1 (Dez. 2020). ISSN: 0098-3500. DOI: 10.1145/3424144. URL: https://doi.org/10.1145/3424144.

[21] Dan Quinlan. "ROSE: Compiler support for object-oriented frameworks". In: *Parallel Processing Letters* 10.02n03 (2000), S. 215–226. URL: https://doi.org/10.1142/S0129626400000214.

[22] Jeffrey R. Chasnov. *The SIR Epidemic Disease Model*. Accessed: 7 July 2025. 2020. URL: https://math.libretexts.org/Bookshelves/Applied_Mathematics/Mathematical_Biology_(Chasnov)/04%3A_Infectious_Disease_Modeling/4.03%3A_The_SIR_Epidemic_Disease_Model.

[23] Zhi Luo, Tong Wu, Mélanie Gallopin, Zhongmin Wang, Qunshan Ma, Stewart Brown, Ben Schwikowski, Evan Z. Macosko und Jiawei Zhang. "A single-cell transcriptomic atlas of the human immune response to influenza vaccination". In: *Nature Methods* 17.7 (2020), S. 685–694. DOI: 10.1038/s41592-020-0856-2. URL: https://www.nature.com/articles/s41592-020-0856-2.

[24] Henrik Zunker, René Schmieding, David Kerkmann, Alain Schengen, Sophie Diexer, Rafael Mikolajczyk, Michael Meyer-Hermann und Martin J. Kühn. "Novel travel time aware metapopulation models and multi-layer waning immunity for late-phase epidemic and endemic scenarios". In: *medRxiv* (2024). DOI: 10.1101/2024.03.01.24303602. URL: https://www.medrxiv.org/content/early/2024/10/28/2024.03.01.24303602.

[25] Wadim Koslow, Martin J. Kühn, Sebastian Binder, Margrit Klitz, Daniel Abele, Achim Basermann und Michael Meyer-Hermann. "Appropriate relaxation of non-pharmaceutical interventions minimizes the risk of a resurgence in SARS-CoV-2 infections in spite of the Delta variant". In: *PLOS Computational Biology* 18.5 (Mai 2022), S. 1–26. DOI: 10.1371/journal.pcbi.1010054.

[26] Anna Wendler, Lena Plötzke, Hannah Tritzschak und Martin J. Kühn. "A non-standard numerical scheme for a novel SECIR integro-differential equation-based model allowing nonexponentially distributed stay times". In: *Applied Mathematics and Computation* 509 (2026). ISSN: 0096-3003. DOI: https://doi.org/10.1016/j.amc.2025.129636. URL: https://www.sciencedirect.com/science/article/pii/S0096300325003625.

[27] David Kerkmann, Sascha Korf, Khoa Nguyen, Daniel Abele, Alain Schengen, Carlotta Gerstein, Jens Henrik Göbbert, Achim Basermann, Martin J. Kühn und Michael Meyer-Hermann. "Agent-based modeling for realistic reproduction of human mobility and contact behavior to evaluate test and isolation strategies in epidemic infectious disease spread". In: *Computers in Biology and Medicine* 193 (2025). ISSN: 0010-4825. DOI: https://doi.org/10.1016/j.compbiomed.

2025.110269. URL: https://www.sciencedirect.com/science/article/pii/S0010482525006201.

[28] Wenzel Jakob and others. *pybind11 – Seamless operability between C++11 and Python*. Accessed: 17 July 2025. 2025. URL: https://pybind11.readthedocs.io/en/stable/basics.html.

[29] LLVM Project. *Introduction to the Clang AST*. Accessed: 14 July 2025. 2024. URL: https://clang.llvm.org/docs/IntroductionToTheClangAST.html.

[30] Robert Tarjan. "Depth-First Search and Linear Graph Algorithms". In: *SIAM J. Comput.* 1.2 (Juni 1972), S. 146–160. ISSN: 0097-5397. DOI: 10.1137/0201010. URL: https://doi.org/10.1137/0201010.

[31] Python Software Foundation. *multiprocessing — Process-based parallelism*. Accessed: 31 July 2025. 2024. URL: https://docs.python.org/3/library/multiprocessing.html.

[32] Informatec Digital. *Mehrprozessorsysteme*. Accessed: 17 July 2025. URL: https://informatecdigital.com/de/Mehrprozessorsysteme/.

[33] Real Python. *Python Concurrency: An Intro to Parallel Programming*. Accessed: 17 July 2025. 2023. URL: https://realpython.com/python-concurrency/%5C#parallelizing-data-processing.

[34] Python Software Foundation. *threading - Thread-based parallelism*. Accessed: 17 July 2025. 2024. URL: https://docs.python.org/3/library/threading.html.

[35] Python Software Foundation. *pickle - Python object serialization*. Accessed: 31 July 2025. 2024. URL: https://docs.python.org/3/library/pickle.html.