DHBW
Duale Hochschule
Baden-Württemberg
Mannheim

DLR

Automatische Codegenerierung für nutzerfreundliche mathematisch-epidemiologische Modelle

Bachelorarbeit

für die Prüfung zum

Bachelor of Science

des Studiengangs Informationstechnik

der Dualen Hochschule Baden-Württemberg Mannheim

von

Maximilian Betz

Abgabe am 18. November 2022

Bearbeitungszeitraum:	07.06.22 – 30.08.22
Matrikelnummer, Kurs:	5986245, TINF19IT1
Abteilung:	Institut für Softwaretechnologie: High-Performance Computing
Ausbildungsbetrieb:	Deutsches Zentrum für Luft- und Raumfahrt e.V.
Betreuer des Ausbildungsbetriebs:	Martin J. Kühn
Gutachter der Dualen Hochschule:	Reiner Hüchting

Erklärung

Ich versichere hiermit, dass ich meine Bachelorarbeit mit dem

Thema

Automatische Codegenerierung für nutzerfreundliche mathematisch-epidemiologische Modelle

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.*

* falls beide Fassungen gefordert sind

Köln, den 18. November 2022

Inhaltsverzeichnis

Abbildungsverzeichnis

Quelltextverzeichnis

Zusammenfassung

Zur Simulation der Verbreitung des Virus SARS-CoV-2 bietet das MEmilio Softwarepaket mathematisch-epidemiologische Modelle und zusätzliche Simulationstools. Durch Python-Bindings wird die C++-Kernbibliothek von MEmilio nach Python exportiert, wodurch mehr Personen das Softwarepaket benutzen können. Um die Entwicklung der Bindings einfacher zu gestalten, wird mit dem Compiler-Frontend Clang eine automatische Codegenerierung für die modellspezifischen Teile der Bindings entwickelt. Es wird auf die Problematiken bei der Erstellung eines solchen Codegenerators eingegangen und unterschiedliche Ansätze zur Lösung diskutiert. Der entwickelte Codegenerator besitzt durch ein modulares Konzept Möglichkeiten zur Erweiterung und Anpassung. Der Programmentwurf konnte erfolgreich für einfache ODE-SIR-typische Modelle umgesetzt werden. Dieser kann für andere Codegeneratoren als Grundlage dienen.

Abstract

The MEmilio software package provides various mathematical models and simulation tools to forecast the spread of SARS-CoV-2. With Python-Bindings to the C++ implementation, MEmilio is accessible to more users. The compiler front end Clang is used to develop an automatic code generator for the model specific parts of the bindings. The problems involved in creating such a code generator and different approaches to solving them are discussed. The developed code generator has possibilities for expansion and adaptation through a modular concept. The program was successfully implemented for simple ODE-SIR-type models and can serve as a basis for other code generators.

1 Einleitung

Infektionskrankheiten waren in der Vergangenheit eine große Gefahr für die Menschheit. Pandemien, wie die spanische Grippe von 1918-1920 oder die Pest von 1346-1353, führten zu Millionen von Todesfällen [33]. Anhand der derzeitigen Pandemie des Virus SARS-CoV-2 wird klar, dass solch eine Gefahr auch heute noch relevant ist. Der Beginn der Pandemie war für viele Menschen eine große Herausforderung. Sie waren in einer Situation mit sehr viel Ungewissheit. Eine Vorbereitung auf durch die Pandemie entstehende gesundheitliche, soziale und berufliche Probleme ist bei fehlendem Wissen über die Verbreitung von SARS-CoV-2 schwierig.

Um die Infektionsausbreitung besser zu verstehen, bietet das Softwarepaket MEmilio C++-Simulationstools für die Covid-19 Pandemie unter Berücksichtigung von Faktoren wie Bevölkerungsgruppen, Infektionszuständen, Ortsauflösung, nichtpharmazeutische Interventionen, Verhalten der Bevölkerung und mehr. Die erhaltenen Daten können als Informationsquelle von Entscheidungsträger*innen in der Politik und Wirtschaft genutzt werden, um passende Regelungen im Umgang mit der Pandemie zu treffen.

Um die Zielgruppe von MEmilio zu erhöhen, werden die Simulationstools zusätzlich für Python zugänglich gemacht. Python stellt als Programmiersprache eine große Basis an Benutzer*innen [38, 29, 27] und ist daher eine Möglichkeit die Software einem breiteren Anwendungskreis zu öffnen. Die Übertragung nach Python wird durch Python-Bindings implementiert. Diese sind für die Entwickler*innen zeitaufwändig zu erstellen. Sie brauchen über die Programmiersprache C++ hinaus auch ein Verständnis für Python, sowie die verwendete Bibliothek zum Binden des C++-Codes.

Ziel dieser Arbeit ist somit, einen Ansatz zu erarbeiten, bei dem die Python-Bindings der C++-Modelle automatisch generiert werden. Ein Prototyp soll programmiert werden, der die automatische Generierung für die ODE-SIR-typischen Modelle von MEmilio durchführt. Für die spätere Implementierung von weiteren Modelltypen und das Ausbauen der bestehenden soll der Generator einfach erweiterbar gestaltet werden.

Die folgende Arbeit beginnt mit einer Einleitung in die mathematische Modellierung einer Pandemie und darauf aufbauend mit einer praktischen Umsetzung dieser im Softwarepaket MEmilio. Danach sollen die Python-Bindings vorgestellt werden, um ein Verständnis für den zu generierenden Code zu erlangen. Daraufhin wird der Codegenerator für die modellspezifischen Python-Bindings mit einem Programmentwurf, einer Beschreibung der Aufgaben der einzelnen Klassen und eine kurze Diskussion über den Programmieraufwand vorgestellt. Zuletzt wird eine Testung des Generators durchgeführt, ein Ausblick für die Zukunft des Prototyps gegeben und ein abschließendes Fazit gezogen.

2 Mathematische Modellierung einer Pandemie

In diesem Kapitel wird eine Einführung in die mathematische Modellierung von Infektionskrankheiten gegeben. Aufgrund der Relevanz des Gebiets wird es schon lange erforscht. Bereits im Jahr 1927 definierten W. Kermack und A. McKendrick die einhergehende Problematik [16]. Eine Kurzfassung der Definition lautet:

> Ein*e (oder mehrere) Infizierte*r wird in eine Gruppe von Personen, welche anfällig für die Infektionskrankheit ist, eingeführt. Die Infektion verbreitet sich von kranken zu gesunden Personen als eine Kontaktinfektion. Jede*r Infizierte*r durchläuft ihre Krankheitsphase und wird entweder durch Genesung oder den Tod von der Menge an Kranken herausgenommen.

Anhand der Problemstellung können mathematisch-epidemiologische Modellierungen entwickelt werden. Die erstellten Modelle sind dabei jedoch immer nur eine Approximation, vereinfachen also die Realität. Auf Grundlage des zu betrachtenden Aspekts sollte ein passendes Modell gewählt werden. Hierfür werden verschiedene Modellarten differenziert, welche die Verbreitung einer Infektionskrankheit verschieden darstellen sowie unterschiedliche Vorteile bieten. Die bekanntesten sind dabei die gewöhnlichen Differentialgleichungs (Abk. ODE aus dem Englischen: ordinary differential equation) SIR-typischen Modelle [3, 16, 25, 21]. Die ODE-Modelle fokussieren sich beispielsweise auf Altersgruppen und Metapopulationen [22], das Testen der Pendler*innen in Metapopulationen [21] oder die Betrachtung von sich ändernden Virusvarianten und Impfungen [19].

Abgesehen davon gibt es Integro-Differentialgleichungsmodelle [15], agentenbasierte Modelle (Abk. ABM) [25, 17] oder Bayessche-Monte-Carlo Ansätze [34].

Das ODE-SEIR-Modell

Innerhalb dieser Arbeit wird der Fokus auf ODE-SIR-typische Modelle gesetzt. Daher wird zur Erklärung der Funktionsweise der ODE-Modellart ein ODE-SEIR-Modell genauer betrachtet. Der Unterschied des ODE-SEIR-Modells im Vergleich zum SIR-Basismodell besteht darin, dass die Latenzzeit nach der Transmission mit einem Virus berücksichtigt wird. Die Latenzzeit ist dabei der Zeitraum, in dem eine Person Virusträger ist, aber noch nicht Andere anstecken kann.

Solch eine Erweiterung des Modells kann sehr wichtig für die Güte des Modells sein. Es wird zwar komplexer, liegt aber auch näher an der Realität. Zum Beispiel die Latenzzeit falsch zu modellieren oder sogar komplett zu ignorieren, kann zu ungenauen Ergebnissen führen. So scheint es, dass solch ein Fehler immer zu einer Unterschätzung der Basisreproduktionszahl R_0 führt [41].

Beim ODE-SEIR-Modell sei die Gesamtpopulation N konstant. Diese wird in 4 Kompartimente aufgeteilt:

- **Anfällige** (engl. susceptible, S), welche gesund sind mit der Möglichkeit angesteckt zu werden.
- **Exponierte** (engl. exposed, E), die den Virus tragen, aber noch nicht ansteckend sind.
- **Infektiöse** (engl. infected, I), die den Virus tragen und ansteckend für Andere sind.
- **Genesene** (engl. recovered, R), welche nicht nochmal krank werden können.

Für jeden Zeitpunkt gilt somit $S+E+I+R=N$. Zu jedem Simulationsschritt lassen sich die Veränderungen in den Kompartimenten berechnen. Ein Simulationsschritt sei dabei ein festgelegtes Zeitintervall. Die möglichen Übergänge verändern sich entsprechend der Infektionsausbreitung, welche durch das ODE-SIR-typische Modell dargestellt werden soll. Die für das ODE-SEIR-Modell möglichen Übergänge werden in Abbildung 2.1 durch Pfeile dargestellt.

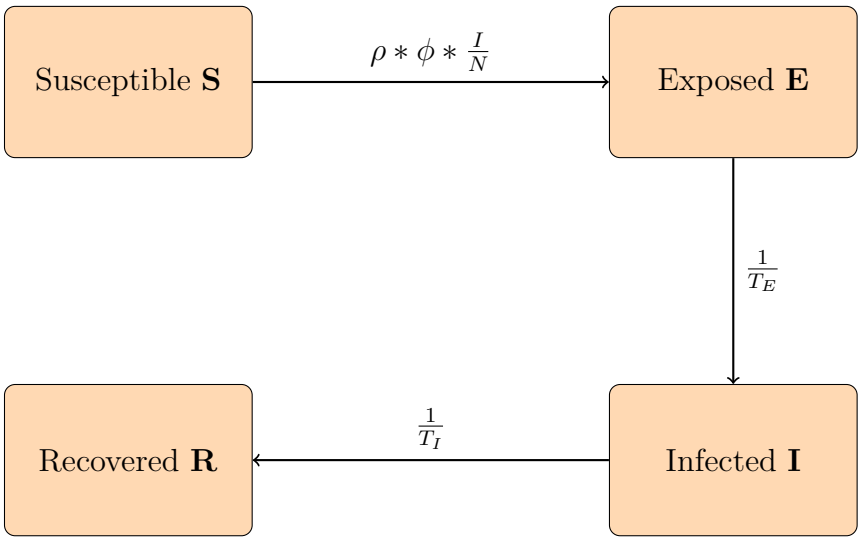

Abbildung 2.1: Darstellung des ODE-SEIR-Modells mit Übergängen und den Faktoren zur Berechnung der Veränderungen in den Kompartimenten. Darstellung und Gleichungen nach dem Schema des Papers [22] erstellt.

Der Faktor auf den Übergangspfeilen wird mit der Menge an Personen im Ausgangskompartiment multipliziert, um den Übergang im Einheitszeitschritt zu bestimmen. Daraus kann die Veränderung in den einzelnen Kompartimenten berechnet werden. Die Formeln ergeben sich aus den eingehenden und ausgehenden Pfeilen eines Kompartiments aus Abbildung 2.1. Mit diesen lässt sich das Verhalten des Modells beschreiben. Mit numerischen Lösungsverfahren kann bei gegebenen Anfangswerten eine Annäherung der Lösung des ODE-Systems berechnet werden.

$$\frac{dS}{dt} = -S\rho\phi\frac{I}{N} \tag{2.1}$$

$$\frac{dE}{dt} = S\rho\phi\frac{I}{N} - \frac{1}{T_E}E \tag{2.2}$$

$$\frac{dI}{dt} = \frac{1}{T_E^I}E - \frac{1}{T_I}I \tag{2.3}$$

$$\frac{dR}{dt} = \frac{1}{T_I^R}I \tag{2.4}$$

Die verwendeten Variablen ρ, ϕ, T_E^I und T_I^R definieren das Verhalten der Pandemie. Die Werte der Variablen verändern sich, je nachdem welche Infektionsausbreitung von dem Modell dargestellt werden soll.

Parameter ρ Gibt die Anzahl an Kontakten pro Einheitsschritt an.

Parameter ϕ Gibt das Übertragungsrisiko des Virus bei einem Kontakt an. Der Term $\rho * \phi$ ergibt die effektiven Kontakte.

Parameter T_E^I Durchschnittliche Anzahl an Tagen, die eine Person exponiert ist.

Parameter T_I^R Durchschnittliche Anzahl an Tagen, die eine Person infektiös ist.

Modifikationen des ODE-SEIR-Modells

Das Kapitel wird mit einigen Erweiterungsmöglichkeiten für das ODE-SEIR-Modell beendet. Die Ansätze können bei der Modellierung betrachtet werden und dabei helfen, das zu lösende Problem bestmöglich darzustellen.

Für eine genauere Repräsentation der Realität kann die Gesamtpopulation in mehr Infektionszustände aufgeteilt werden. Durch das Hinzufügen der Zustände *Hospitalized*, *ICU* und *Dead* können der Krankheitsverlauf und die damit zusammenhängende ärztliche Behandlung bei einer Infektion besser dargestellt werden [22, 18].

Auch eine veränderte Anordnung der Kompartimente ist bei ODE-SIR-typischen Modellen möglich. So beschreibt ein ODE-SIS-Modell den Verlauf einer Pandemie ohne einen Endzustand, in welchem die Bevölkerung immun gegen das Virus ist. Genesene Personen werden zurück in das Kompartiment *Susceptible* übertragen [32]. Kompartimente können somit mehrfach durchlaufen werden.

Das dargestellte ODE-SEIR-Modell fokussiert sich auf eine Epidemie. Solch ein Modell muss die rapide Ausbreitung einer Infektionskrankheit meist innerhalb eines kleinen Zeitraumes darstellen. Daher kann dafür eine konstante Gesamtpopulation angenommen werden. Für einen endemischen Verlauf müsste die Veränderung der Population durch Geburten und Todesfälle angepasste werden [3].

Aus Realdaten lassen sich Unterschiede in Faktoren wie dem Krankheitsverlauf oder der Sterblichkeitsrate bezogen auf das Alter erkennen [26, 28, 23]. Auch Kontaktmuster unterscheiden sich je nach Altersgruppe, beispielsweise durch das soziale Umfeld oder dem beruflichen Stand. Daher könnte eine Aufteilung der Population in Altersgruppen implementiert werden. Eine passende Modellierung hat Fred Brauer im Zusammenhang mit dem Choleraerreger entwickelt [2].

3 Softwarepaket MEmilio

MEmilio (aus dem Englischen *high performance Modular EpideMIcs simuLatIOn software*) [5] ist ein Softwarepaket zur Simulation der Ausbreitung von Infektionskrankheiten. Es wird vom Institut Softwaretechnologie des Deutschen Zentrums für Luft- und Raumfahrt in Zusammenarbeit mit dem Helmholtz Zentrum für Infektionsforschung [12] entwickelt. Es ist im Rahmen der Pandemie des Virus SARS-CoV-2 entstanden und soll mit mathematischen Modellen dessen Infektionsausbreitung simulieren. Eine große Aufgabe stellt hierbei die Analyse von Realdaten der Pandemie dar, um die Modelle dem Covid-19 Pandemieverhalten anzupassen.

Mit der Software soll eine Analyse von verschiedenen Faktoren ermöglicht werden. So soll Entscheidungsträger*innen der Politik geholfen werden, anhand des simulierten Verlaufs passende nicht-pharmazeutische Interventionen (Abk. NPI) festzulegen. Die NPIs spielen bei der Bekämpfung der Pandemie eine essentielle Rolle und brauchen daher eine gute Informationsgrundlage. Beispiele für NPIs bei einer Pandemie sind Kontaktbeschränkungen, Ausgangssperren, Home-Schooling-/Home-Office-Regelungen oder eine Maskenpflicht. Zudem werden die Modellierungswerkzeuge mit einer Webvisualisierung für Benutzer*innen zugänglich gemacht.

3.1 Softwareaufbau

Die Struktur des Softwarepakets und damit auch die generelle Funktionsweise wird im folgenden Abschnitt erklärt. Dafür wird zunächst die physische Struktur und danach die logische Struktur des Softwarepakets betrachtet.

3.1.1 Physische Dateistruktur

Die Abbildung 3.1 stellt eine vereinfachte Form der physischen Dateistruktur des MEmilio Softwarepakets dar. Sie zeigt nicht jede einzelne Datei, sondern gibt einen guten Überblick über das Konzept. Beim Erklären der Struktur werden die Begriffe aus der Darstellung verwendet. Grundsätzlich besteht die Software aus zwei Teilen: Dem C++-Hauptteil im Ordner *cpp* und den Python-Paketen im Ordner *pycode*.

Der C++-Teil des Softwarepakets beinhaltet die Kernbibliothek von MEmilio mit den benötigten Modellierungswerkzeugen zur Simulation einer Infektionsausbreitung und einzelne Bibliotheken für die mathematisch-epidemiologischen Modelle. Als Buildsystem wird *CMake* verwendet.

Die Kernbibliothek befindet sich im Ordner *memilio* und wird dort weiter nach folgenden Aufgaben unterteilt [5].

compartments Stellt das Framework zum Erstellen von SIR-typischen Modellen, zum Beispiel die Basisklasse *CompartmentalModel*.

data Werkzeuge zum Einlesen von Inputdaten für die Simulation.

epidemiology Hier befinden sich generelle Werkzeuge für epidemiologische Modelle, beispielsweise für die Umsetzung von Altersgruppen oder Kontaktmustern.

io Implementiert die Möglichkeit zum Einlesen und Ausgeben von Daten aus Dateien und das Framework für Error Handling.

math Bindet verschiedene mathematische Funktionen in MEmilio ein, wie zum Beispiel Lösungsverfahren für die gewöhnlichen Differentialgleichungssysteme.

mobility Implementiert die räumliche Aufteilung für eine Simulation durch ein graphenbasiertes Modellnetzwerk.

utils Stellt weitere Hilfswerkzeuge bereit, welche nicht spezifisch für eine epidemiologische Modellierung sind, wie zum Beispiel eigene Arrays, Logging, eine Zeitklasse zur Repräsentation von Kalenderdaten oder einen Zufallszahlengenerator.

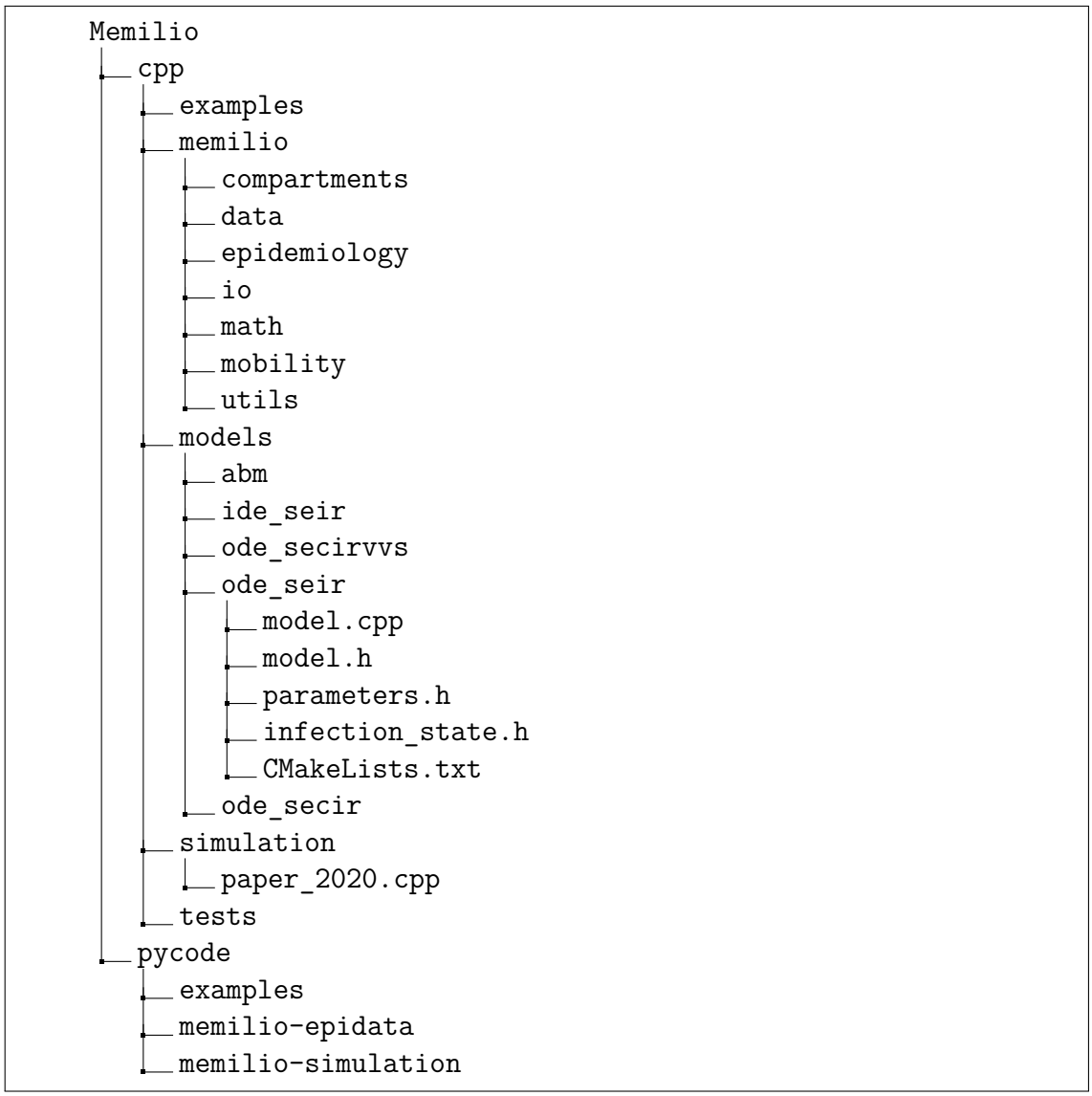

Abbildung 3.1: Physische Dateistruktur des MEmilio Softwarepakets. Nicht jede Datei ist abgebildet und die einzelnen Python-Pakete *epidata* und *simulation* sind ohne Untergliederung dargestellt.

Mit der Kernbibliothek kann ein*e Entwickler*in eigene Modelle implementieren und für Anwendungsfälle simulieren. Die Modelle befinden sich in dem gemeinsamen Verzeichnis *models*. Jedes Modell bekommt dafür einen eigenen Unterordner, in dem seine Source- und Header-Dateien liegen. Zum Zeitpunkt dieser Arbeit wurden bereits ein ABM, ein Prototyp eines IDE-SEIR-Modells und mehrere ODE-SIR-typischen Modelle implementiert.

Als Teil dieser Arbeit wurde das in Kapitel 2 vorgestellte ODE-SEIR-Modell implementiert. Es wird als Beispiel zur Entwicklung in MEmilio genutzt. Das Modell ist bereits in der Darstellung der Dateistruktur abgebildet. Der entsprechende Ordner ist *ode_seir*, in dem sich der modellspezifische Code befindet. Für die Erklärung der Implementation werden die einzelnen Dateien betrachtet und die dadurch realisierten Aspekte des Modells genannt.

Die Kompartimente werden in der Datei *infection_state.h* definiert. Sie werden als Enum-Klasse mit dem Namen *InfectionState* realisiert. Das Enum bekommt für jedes Kompartiment einen Wert. Zusätzlich wird als letzter Wert eine Hilfsvariable *Count* gesetzt. Der Zahlenwert von *Count* entspricht somit immer der Anzahl der Kompartimente.

```
1 enum class InfectionState
2 {
3   Susceptible,
4   Exposed,
5   Infected,
6   Recovered,
7   Count
8 };
```

Quelltext 3.1: Enum der Infektionszustände eines ODE-SEIR-Modells. (C++)

Die Parameter des Modells werden in der Datei *parameters.h* definiert. Das Modell benötigt vier Parameter, wie bereits in Kapitel 2 erklärt. Jeder Parameter wird durch eine eigene Klasse beschrieben. Zum Benutzen im Modell werden alle Parameter durch die Klasse *ParameterSet* aus der Kernbibliothek von MEmilio zusammengefügt. Das ParameterSet wird unter dem Alias *Parameters* weiterverwendet.

Mit den beiden Klassen *Parameters* und *InfectionState* als Grundlage kann das Modell in der Datei *model.h* geschrieben werden. Das Modell verwendet die Basisklasse *CompartmentalModel*. Dies ist eine Template-Klasse, welche als Template-Parameter die Kompartimente, eine Population und ein Set der Parameter der Modellierung bekommt. Alle ODE-SIR-typischen Modelle können von der *CompartmentalModel*-Klasse erben und benötigen daher auch immer eine Population

mit Kompartimenten und ein Set an Parametern. Das SEIR-Modell implementiert einen Konstruktor, welcher intern den Konstruktor der Basisklasse aufruft, sowie die Funktion *get_derivatives()*. In dieser werden die Übergänge zwischen den Kompartimenten bei einem Simulationsschritt berechnet. Die Funktion muss auch jedes andere ODE-SIR-typische Modell implementieren.

```
1 class Model : public CompartmentalModel<InfectionState,
      Populations<InfectionState>, Parameters>
2 {
3   Model() {}
4   void get_derivatives( \\ ... ) const override{
5     \\ ...
6   }
7 };
```

Quelltext 3.2: Minimalimplementierung einer Modellklasse für ODE-Modelle. (C++)

Zum Kompilieren des Modells befindet sich noch eine *CMakeLists.txt* im Verzeichnis.

Der zweite Teil des Softwarepakets sind eigene Python-Pakete, welche im Ordner *Pycode* liegen. Das *epidata*-Paket stellt mehrere Skripte zum Herunterladen und Einlesen von epidemiologischen und für die Simulation relevante Daten aus verschiedenen Quellen. Darunter sind Daten vom Robert-Koch-Institut, von der Bundesagentur für Arbeit, vom Statistischen Bundesamt destatis und von der Johns Hopkins Universität. Das *simulation*-Paket stellt Python-Bindings für den C++-Hauptteil zur Verfügung. Durch die Bindings können Simulationen mit Modellen über Python ausgeführt werden. Die MEmilio Python-Bindings werden in Kapitel 4 genauer erklärt.

3.1.2 Logische Struktur der MEmilio C++-Bibliothek

Die logische Struktur einer Software stellt die Namensräume sowie den Zusammenhang der verfügbaren Klassen und Funktionen mit seinen Komponenten dar. Es gibt also eine abstrakte Übersicht über das Softwarepaket und stellt den Zugriff auf die Funktionalitäten dar. Die logische Struktur der MEmilio C++-Bibliothek mit dem Namen *mio* wird in Abbildung 3.2 dargestellt.

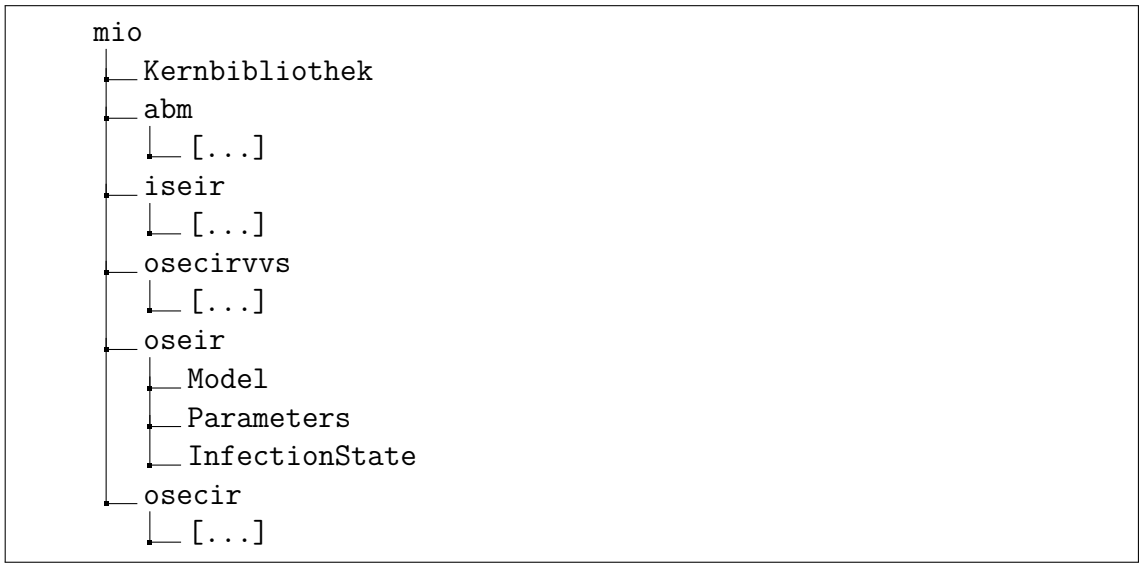

Abbildung 3.2: Logische Struktur der C++-Bibliothek. Innere Knoten stellen die untergliederten Namensräume und Blätter die nutzbaren Modellierungswerkzeuge dar.

Zu erkennen ist, dass die Kernbibliothek mit den Modellierungswerkzeugen aus den verschiedenen Unterordnern nicht in weitere Namensräume untergliedert wird, weil die Kernbibliothek für alle Modelle generisch implementiert ist. Die Modelle dagegen werden jeweils innerhalb eines eigenen Namensraum implementiert. Im Namensraum *oseir* des ODE-SEIR-Modells befinden sich die zuvor beschriebenen Klassen *Model*, *Parameters* und *InfectionState*.

3.2 Erweiterung des ODE-SEIR-Modells

MEmilio bietet zum Zeitpunkt dieser Arbeit zwei weitere ODE-SIR-typische Modelle neben dem SEIR-Modell. Im ODE-SECIR-Modell werden weitere Kompartimente eingeführt. Es teilt die Infektiösen in Personen mit und ohne Symptome auf und ermöglicht die Darstellung schwererer Krankheitsverläufe mit einer Hospitalisierung, einer Behandlung in der Intensivstation und dem Sterben durch Covid-19 [22]. Außerdem verwendet das Modell eine Aufteilung der Population nach Altersgruppen.

Das ODE-SECIRVVS-Modell implementiert das im Paper [19] beschriebene Modell, welches das ODE-SECIR-Modell durch sich ändernden Virusvarianten und Impfungen erweitert.

4 Verwenden von C++-Code über Python API durch Bindings

4.1 Gründe für das Binden von Code

Durch Bindings ist es möglich Programmcode aus einer Programmiersprache in einer anderen Programmiersprache zugänglich zu machen. In dieser Arbeit wird das Übertragen von C++-Code nach Python betrachtet. So kann zum Beispiel C++-Code für Performance-kritische Programmteile innerhalb eines Python-Projekts verwendet oder bereits bestehende, gut ausgebaute und robuste C++-Bibliotheken in Python genutzt werden.

In MEmilio stehen die Modellierungswerkzeuge bereits in C++ zur Verfügung. Das Softwarepaket profitiert so von den vielen robusten C++-Bibliotheken und der guten Performanz der Programmiersprache. Für eine bessere Benutzerfreundlichkeit und eine Erweiterung der Zielgruppe werden die Modellierungswerkzeuge auch für Benutzer*innen über Python aufrufbar gemacht.

4.2 Bibliothek Pybind11

Für die Python-Bindings in MEmilio wird die Bibliothek *Pybind11* [14] verwendet. Mit Pybind11 kann C++ und Python in beiden Richtungen zugänglich gemacht werden. Die Bibliothek fokussiert sich dabei auf C++11 oder neuere Versionen und

verwirft damit die Legacy-Elemente der älteren Versionen. Dadurch soll Pybind11 aufgeräumter und benutzerfreundlicher als vergleichbare Bibliotheken, wie zum Beispiel *Boost.Python*, sein. Zudem wird weiterhin aktiv an Pybind11 gearbeitet, so dass die Bibliothek regelmäßig neue C++- und Python-Funktionen sowie Fehlerbehebungen erhält.

4.2.1 Einrichten eines Pybind11-Moduls

Bevor die ersten Python-Bindings programmiert werden, braucht es einige Einrichtungsschritte. Zum einen muss das Buildsystem eingerichtet werden. Mit dem Python-Paket *scikit-build* [31] wird über eine *setup.py* mit dem Befehl `setup_requires=['cmake']` das Buildsystem *CMake* gestartet. Das Paket *scikit-build* ist eine Erweiterung der Standardbibliothek *setuptools* [1] für das zusätzliche Verwenden von *CMake*. In der Datei *CMakeLists* der Bindings müssen die einzelnen Python-Module, die erzeugt werden sollen, mit den verwendeten cpp-Dateien, in welchen die Bindings stehen, angegeben werden. Zudem müssen die Pfade zu den Funktionalitäten, welche gebindet werden sollen, definiert werden. Mit dem Aufruf der *setup.py* können die Python-Bindings gebaut werden. Somit ist das Bauen der Bindings für Python Entwickler*innen geläufig.

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

Quelltext 4.1: Definieren eines Pybind11-Moduls zum Binden von C++-Code. (C++)

Mit dem eingerichteten Buildsystem kann ein Modul erstellt werden, was im Quelltext 4.1 dargestellt ist. In den folgenden Codeausschnitten wird *py* als Alias für den Namensraum *pybind11* verwendet. Mit dem Makro *PYBIND11_MODULE* wird ein Pybind11-Modul erstellt. Der erste Parameter (example) bestimmt den Namen des Python-Moduls. Der zweite Parameter mit der Variable *m* des Typs *module_*

definiert das Hauptinterface der Bindings. Damit können innerhalb des Makros mit den Funktionen von Pybind11 der C++-Code mit Python verknüpft werden.

4.2.2 Funktionen

Mit dem eingerichteten Modul kann der erste C++-Code gebindet werden. Als Anfang wird eine Funktion nach Python übertragen. Zur Erklärung wird ein Beispiel aus der Pybind11 Dokumentation [13] benutzt. Angenommen eine Funktion zur Addition zweier Integerzahlen liegt im C++-Code vor (siehe Quelltext 4.2).

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

Quelltext 4.2: Binding einer Funktion zum Addieren zweier *int*-Werte. (C++)

Mit der Funktion *def()* des Hauptinterfaces kann die Additionsfunktion für Python exponiert werden. Als erster Parameter wird der Funktionsname in Python gesetzt. Da die API in Python möglichst ähnlich zu der in C++ sein sollte, damit Nutzer*innen die Funktionen finden und weniger dokumentiert werden muss, wird der Name der C++-Funktion übernommen. Nun muss noch definiert werden, was passiert, wenn *add()* in Python aufgerufen wird. Dafür wird ein Zeiger zur C++-Funktion übergeben. Somit wird beim Aufrufen in Python im Hintergrund die C++-Funktion mit den übergebenen Parametern ausgeführt und der Rückgabewert zurück nach Python übersetzt.

Damit Python weiß, was es mit einem Rückgabewert, wie zum Beispiel einem Wert des Typs *int*, machen soll, müssen alle Typen der Rückgabewerte Python bekannt gemacht werden. Die Variablentypen brauchen also eigene Bindings. Pybind11 hat bereits einige Variablentypen standardmäßig implementiert und übersetzt sie automatisch für Python, darunter auch alle Standardtypen aus C++. Somit muss ein C++-Integer nicht noch spezifisch definiert werden.

In Python müssen keine Datentypen der Eingabeparameter von Funktionen definiert werden. Daher könnte solch eine Additionsfunktion neben Integern auch zum Beispiel Doubles addieren. Um das in C++ zu implementieren könnte man die Funktion *add()* überladen. Der vorherige Codeausschnitt wird also durch eine zweite C++-Funktion erweitert.

```
1  int add(int a, int b) {
2    return a + b;
3  }
4
5  int add(double a, double b) {
6    return a + b;
7  }
8
9  PYBIND11_MODULE("example", m) {
10   m.def("add", py::overload_cast<int    ,int    >(&add));
11   m.def("add", py::overload_cast<double ,double >(&add));
12 }
```

Quelltext 4.3: Erweiterung des Bindings zur Addition zweier Zahlen für den Datentyp *double* durch das Überladen der Funktion. (C++)

Um die Funktion zu binden, muss angegeben werden, welche der beiden Funktionen genommen werden soll. Dafür ergänzt man die Übergabe des Funktionszeigers durch *py::overload_cast<int ,int>*. Als Template-Argumenten werden die Typen der Eingabeparameter angegeben. Nun muss noch Zeile 11 hinzugefügt werden, um die Addition von Doubles zu exponieren. Bei einem Aufruf in Python versucht Pybind11 die Überladung der Funktion aufzulösen. Die Auflösung passiert in zwei Schritten. Im ersten Schritt werden alle C++-Funktionen geprüft, ohne eine Typkonvertierung zu erlauben. Findet sich keine passende Funktion, wird der Schritt mit Erlauben einer Typkonvertierung wiederholt. Wenn auch der zweite Durchgang fehlschlägt, wird ein *TypeError* geworfen und an Python zurückgegeben.

Nach dem Bauen des Python-Moduls kann es in Python unter dem Namen *example* importiert werden und die *add()*-Funktion, wie im Codeausschnitt 4.4 zu sehen, verwendet werden.

```python
import example

result_int = example.add(2, 3)
result_double = example.add(4.5, 17.3)
```

Quelltext 4.4: Verwenden einer gebindeten Funktion aus C++ in Python. (Python)

4.2.3 Klassen

Als nächstes soll eine einfache Klasse nach Python übertragen werden. Dafür wird die in Codeausschnitt 4.5 dargestellte Implementierung eines Rechtecks (Codebeispiel aus Quelle [35]) in C++ angeschaut. Dabei ist die Implementierung nicht optimal, sondern dient als Veranschaulichung der Verwendung von Pybind11.

```cpp
class Rechteck {
public:
  Rechteck(int a, int b) {
    laenge = a;
    breite = b;
  }
  void setlaenge(int a) {
    laenge = a;
  }
  int getlaenge() const {
    return laenge;
  }
  int getflaeche() const {
    return laenge * breite;
  }
  int breite;

private:
  int laenge;
};
```

Quelltext 4.5: Klasse zur Bechreibung eines Rechtecks. Implementiert die Seitenkanten als Variablen und eine Funktion zur Berechnung der Fläche. (C++)

Im Folgenden wird in Form von kleinen Ausschnitten der Code für das Binden der Klasse erklärt.

```
1 PYBIND11_MODULE("example", m) {
2   py::class_<Rechteck>(m, "Rechteck")
3     // ...
4 }
```

Quelltext 4.6: Beginn der Python-Bindings der Klasse Rechteck. (C++)

Hier muss zunächst das Modul initialisiert werden (siehe Quelltext 4.6). Um die Klasse Python bekanntzumachen, bietet Pybind11 die Funktion *class_()*. Sie braucht als Parameter das Modulinterface und einen Klassennamen für Python. Die C++-Klasse *Rechteck* wird als Template-Argument übergeben. Die Funktion gibt das Pybind11-Objekt der Klasse Rechteck zurück. Mit dem Objekt können die weiteren Eigenschaften von Rechteck für Python zugänglich gemacht werden.

```
1   .def(py::init<int, int>(), py::arg("a"), py::arg("b"))
```

Quelltext 4.7: Binding des Konstruktors der Klasse *Rechteck*. (C++)

Im Codeausschnitt 4.7 wird der Konstruktor gebindet. Pybind11 stellt dafür *py::init<>()* zur Verfügung. Damit kann der Konstruktor nach dem Prinzip des Bindens einer Funktion übertragen werden. Als Template-Argumente von *py::init<>()* müssen die Typen der Eingabeparameter des Konstruktors gesetzt werden. Zusätzlich werden noch die Parameternamen für den Konstruktor mit der Funktion *py::arg()* definiert.

```
1   .def_readwrite("breite", &Rechteck::breite)
2   .def_property("laenge", &Rechteck::getlaenge, &Rechteck::
     setlaenge)
```

Quelltext 4.8: Binding der Membervariablen Breite und Laenge. (C++)

Im Ausschnitt 4.8 werden die Attribute der Klassen übertragen. Die Breite ist dabei als öffentliche Variable gesetzt und wird daher über *def_readwrite()* nach Python übertragen. Da Python keine Zugangsbeschränkungen hat, werden private Membervariablen durch die Setter- und Getter-Funktion in Python zugänglich gemacht. Dabei können entweder beide Funktionen übertragen werden oder die Variable in Python als Property dargestellt werden. Für das Binden als Property bietet Pybind11 den Befehl *def_property()*. Im Beispiel wird die Länge als Property zugänglich gemacht, wodurch diese direkt bearbeitet werden kann, identisch zum Attribut Breite.

```
1   .def_property_readonly("flaeche", &Rechteck::getflaeche)
```

Quelltext 4.9: Binding der Membervariable Flaeche. (C++)

Genauso kann man auch leicht komplexere Funktionen als Property binden. So kann die Fläche auch als Property Python zugänglich gemacht werden. Die Fläche wird im Codeausschnitt 4.9 nur mit Lesezugriff definiert, da sie auch in C++ von Anwender*innen nicht gesetzt werden kann. Bei kurz berechneten Werten, wie der Fläche, macht die Implementierung als Property das Modell in Python übersichtlicher. Nicht triviale Getter-/Setter-Funktionen sollten als eigene Funktion nach Python übertragen werden [39, 10].

```
1   .def("__str__",
2     [](const Rechteck &r) {
3       return "Das Rechteck hat die Flaeche: " + r.flaeche;
4     };
```

Quelltext 4.10: Hinzufügen einer formatierten Ausgabe über Python-Bindings. (C++)

Als Zusatz wird im Ausschnitt 4.10 noch die Ausgabe eines Objektes der Klasse auf der Kommandozeile angepasst. Dafür bietet Python die *Dunder*-Funktion (Kurz für Double-Underscore-Funktion) *___str___()*. Alle Dunder-Funktionen können mit Pybind11 angepasst werden. Am Beispiel erkennt man auch eine weitere Art die Logik einer Funktion für Python zu definieren. Anstatt eines Zeigers zu einer C++-Funktion wird hier eine Lambdafunktion definiert.

Zuletzt ist für das Binden von Klassen der Umgang mit Vererbungen wichtig. Dafür wird dem Beispiel eine Basisklasse *Figur* (Codebeispiel aus Quelle [9]) hinzugefügt (siehe Quelltext 4.11). Sie kann generelle Eigenschaften einer 2-dimensionalen geometrischen Form implementieren.

```
1 class Figur {
2   Punkt mittelpunkt;
3   Farbe farbe;
4   void verschiebenUm(Punkt v) { mittelpunkt += v; };
5   // ...
6 };
7
8 class Rechteck : public Figur {...};
```

Quelltext 4.11: Erweiterung des Beispiels durch eine Klasse *Figur*, von welcher die Klasse Rechteck erbt. (C++)

Hierfür müsste die Klasse *Figur* und alle dazugehörigen Funktionen und Variablentypen für Python zugänglich gemacht werden. Danach kann das Binding des Rechtecks angepasst werden. Die Funktion *class_()* braucht dafür alle Basisklassen, welche als Template-Argumente nach der eigentlichen Klasse, wie im Codeausschnitt 4.12 zu sehen, übergeben werden. Damit wäre die Klasse *Rechteck* nun in Python benutzbar.

```
1 PYBIND11_MODULE("example", m) {
2   py::class_<Figur>(m, "Figur")
3     // ...
4   py::class_<Rechteck, Figur>(m, "Rechteck")
5     // ...
6 }
```

Quelltext 4.12: Binding der Klasse *Rechteck* mit der Basisklasse *Figur*. (C++)

4.2.4 Enumerationen

Um das Vorgehen bei einer Enumeration (Abk. Enum) anzuschauen, wird ein Beispiel aus MEmilio verwendet. Betrachtet wird das Enum *InfectionState* des ODE-SEIR-Modells. In diesem stehen die möglichen Infektionszustände einer Person.

```cpp
enum class InfectionState
{
  Susceptible,
  Exposed,
  Infected,
  Recovered,
  Count
};

PYBIND11_MODULE("example", m) {
  py::enum_<InfectionState> enum_handle(m, "InfectionState");
  enum_handle.value("Susceptible", InfectionState::Susceptible
   );
  enum_handle.value("Exposed", InfectionState::Exposed);
  enum_handle.value("Infected", InfectionState::Infected);
  enum_handle.value("Recovered", InfectionState::Recovered);
  // ...
}
```

Quelltext 4.13: Darstellung der Bindings eines Enums. Als Beispiel werden die Infektionszustände einer Population des ODE-SEIR-Modells verwendet. (C++)

Pybind11 stellt für Enumerationen die Funktion *enum_()* zur Verfügung. Verwendet wird die Funktion sehr ähnlich zum Binden von Klassen. Über value() werden die Member für Python zugänglich gemacht. Der fertige Code der Bindings ist in Abbildung 4.13 zu sehen.

In MEmilio wird für Python eine erweiterte Form der Enumerationen verwendet. Die Enums sind Python-typisch iterierbar. Um eine Klasse in Python iterierbar zu machen, braucht es eine definierte Sequenz der Klasse und einen Iterator, der den Anfang der Sequenz definiert [20, 40]. Für die Implementierung braucht die Enumeration einen weiteren Wert *Count*. Dieser wird nicht vom Benutzer verwendet, sondern intern zur Bestimmung der Länge der Enumeration gebraucht.

```
1  struct Values {
2  };
3  pybind11::class_<Values>(m, ("_Values"))
4    .def("__iter__", [](Values& /*self*/) {
5      return E(0);})
6    .def("__len__", [](Values& /*self*/) {
7      return (size_t)E::Count;});
```

Quelltext 4.14: Erster Teil der Implementation um das Enum *InfectionState* iterierbar zu machen. (C++)

Zum Erreichen der Iterierbarkeit des Enums braucht es eine Dummyklasse, welche den Iterator des Enums zurückgibt. Die Dummyklasse wird gebraucht, da *___iter___()* keine statische Methode sein darf, weshalb sie nicht dem Enum selbst hinzugefügt werden kann. Im Codeausschnitt 4.14 wird dafür die Klasse *Values* geschrieben. Die Dunder-Funktion *___iter___()* gibt den passenden Iterator zurück, den ersten Member des Enums. Durch die Implementierung von *___len___()* kann die Länge des Enums mit der Funktion *len()* abgefragt werden.

```
1  enum_handle.def_static("values", []() {
2  return Values{};
3  });
4
5  enum_handle.def("__next__", [](E& self) {
6    if (self < E::Count) {
7      auto current = self;
8      self         = E(std::underlying_type_t<E>(self) + std::
     underlying_type_t<E>(1));
9      return current;
10   }
11   else {
12     throw pybind11::stop_iteration();
13   }
14 });
```

Quelltext 4.15: Zweiter Teil der Implementierung um das Enum *InfectionState* iterierbar zu machen. (C++)

Die zwei Funktionen im Codeausschnitt 4.15 werden der Python-Klasse des Enums hinzugefügt. Mit der ersten Funktion kann vom Python-Code aus über das Enum auf die Klasse *Values* zugegriffen werden und so der Iterator übergeben werden. Die zweite Funktion definiert die Sequenz beim Iterieren. Dafür wird in Python *___next___()* verwendet, welche für jeden Aufruf den aktuellen Wert auf den der Iterator zeigt zurückgibt und den Iterator in der Sequenz einen Schritt weiter setzt. Beim Erreichen des Endes wird über die Funktion *stop_iteration()* die Python-Exception *StopIteration* geworfen. Sie beendet das korrekte Iterieren über ein Objekt [20]. In Python wird über das Enum *InfectionState* mit der Schleife `for value in InfectionState.values` iteriert.

4.3 Implementierung innerhalb des Softwarepakets MEmilio

Hier wird die Implementierung von Python-Bindings im Softwarepaket MEmilio vorgestellt. Dafür wird auf die bereits eingeführten Strukturen des Softwarepakets in Kapitel 3 und den Grundlagen der Pybind11-Funktionen zurückgegriffen. Die Python-Bindings von MEmilio bieten das eigene Framework *Pymio*. Es bietet Hilfsfunktionen, um ganze Klassen aus MEmilio zu binden und Redundanz im Code zu verhindern.

Dadurch muss zum Beispiel beim Binden der *CompartmentalModel*-Klasse für unterschiedliche ODE-SIR-typischen Modelle nicht jedes Mal der entsprechende Code geschrieben werden, sondern die Funktion *bind_compartmentalModel()* mit den entsprechenden Argumenten kann aufgerufen werden.

Eine weitere Aufgabe des *Pymio*-Frameworks ist es, die Klassen mit Template-Argumenten zu übertragen. In Python gibt es solche Klassen nicht, daher müssen diese davor aufgelöst werden. Die entsprechende Funktion löst beim Ausführen die Template-Klasse auf, wofür die Funktion auch Template-Argumente braucht.

4.3.1 Übertragen der Strukturen

Damit die Python-Bindings für Entwickler*innen einfach und intuitiv zu erstellen sind, sollte die Struktur der Bindings an die Struktur des C++-Codes angepasst werden. In der Darstellung 4.1 sieht man den Vergleich der beiden physischen Dateistrukturen, links die der Python-Bindings und rechts des C++-Hauptteils.

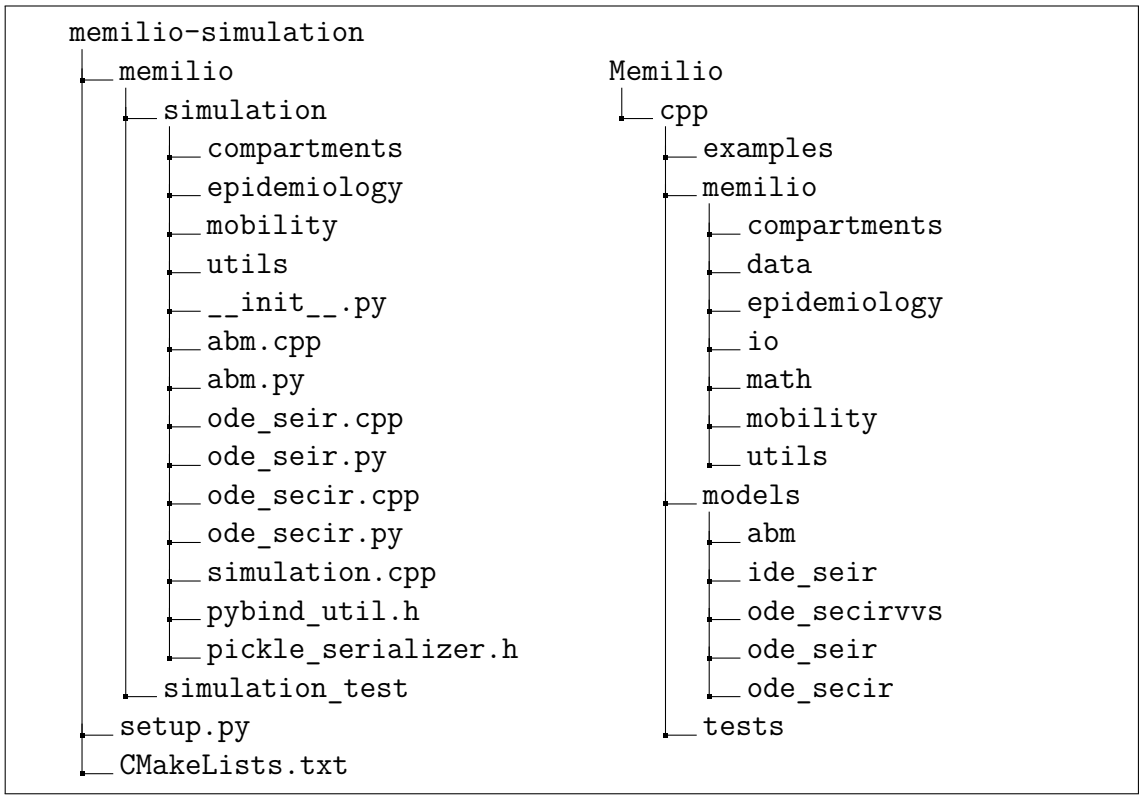

Abbildung 4.1: Vergleich der physischen Dateistrukturen des Pakets der Python-Bindings (links) und des C++-Teils (rechts).

Es ist zu erkennen, dass sich die beiden Strukturen darin unterscheiden, dass die Modelle nicht durch eine Untergliederung in zwei Ordnern von der Kernbibliothek getrennt werden. Ansonsten wird versucht, die Struktur beizubehalten. Zum Beispiel befindet sich die Klassendeklaration von *Index* im C++-Code im Ordner *utils*. Daher befinden sich im *Pycode*-Paket die entsprechenden Bindings der Klasse *Index* in dem gleichnamigen Ordner *utils*. Dadurch kann das Wissen der Struktur des

C++-Code auf die Python-Bindings übertragen werden. Entwickler*innen haben es somit beim Einarbeiten in die Python-Bindings einfacher. Zusätzlich ist zu sehen, dass nicht jeder Ordner in den Python-Bindings vorkommt, da diese nicht alle Funktionalitäten des C++-Codes bindet. Die Ordner der Kernbibliothek und die beiden Dateien *pybind_util.h* und *pickle_serializer.h* bilden das Pymio-Framework. Sie erstellen keine Python-Module, sondern bieten Hilfsfunktionen zum Binden der C++-Kernbibliothek.

Die Modelle werden in der Struktur an Stelle eines Ordners durch zwei Dateien dargestellt. Der C++-Code enthält die Bindings des modellspezifischen Codes. Die Python-Dateien definieren die logische Struktur des endgültigen MEmilio Python-Moduls.

Die *simulation.cpp* erstellt ein Python-Modul mit den restlichen Funktionalitäten, die zur Simulation der Modelle benötigt werden. Die *___init___.py* hilft auch beim Erstellen der logischen Struktur des MEmilio Python-Moduls.

Als nächstes wird die logische Struktur betrachtet. Auch bei dieser sollen sich beide ähneln, damit Benutzer*innen es einfacher haben, die Python-Module zu verstehen. In Abbildung 4.2 ist wieder der Vergleich beider Strukturen dargestellt.

Der einzige Unterschied in den Strukturen ist eine zusätzliche Ebene im Python-Modul. Sie wird benötigt, um eine Unterteilung zwischen dem *epidata*- und dem *simulation*-Untermodul zu erhalten.

Um die logische Struktur zu erhalten und gleichzeitig eine gute physische Struktur zu haben, braucht es einen Zwischenschritt. Das Problem ist, dass die generierten Module von Pybind11 nicht Teil des MEmilio Softwarepakets sind. Daher werden an Stelle davon Pseudomodule generiert. Diese werden nicht von den Benutzer*innen verwendet, sondern werden vollständig in die entsprechenden Python-Dateien importiert. Die logische Struktur wird so durch die physische Struktur der Python-Dateien und deren Namen bestimmt.

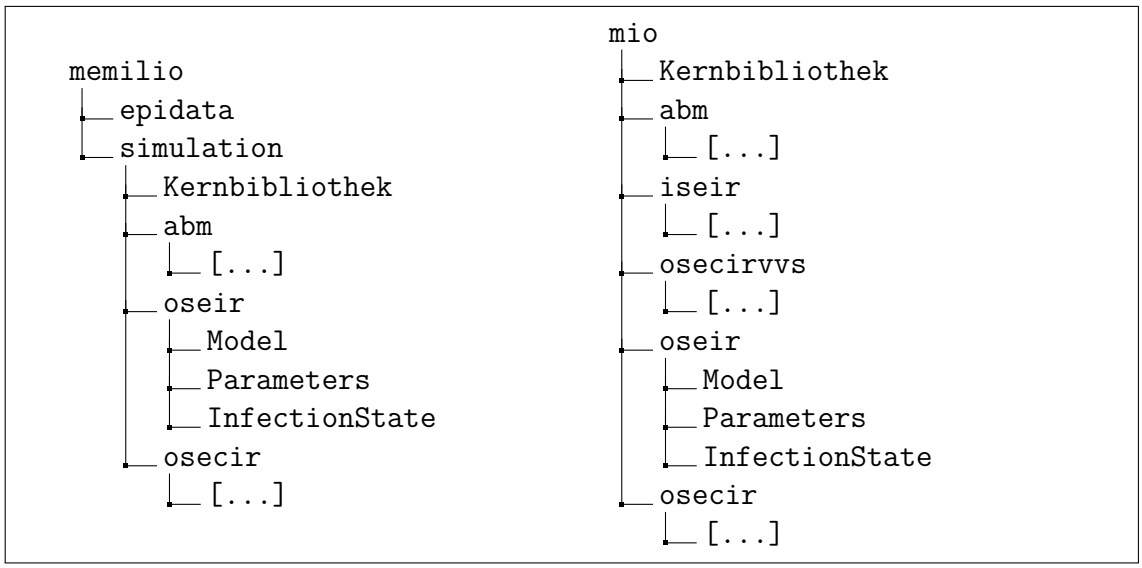

Abbildung 4.2: Vergleich der logischen Dateistrukturen des Pakets der Python-Bindings (links) und des C++-Codes (rechts).

Die Population eines ODE-Modells in MEmilio benutzt als Basis die Klasse *CustomIndexArray*. Sie implementiert ein Array, bei dem eigene definierte Klassen als Indizes benutzt werden können. Jede Population wird durch unterschiedliche Faktoren aufgeteilt, welche als Dimensionen mit eigenem Index im Array dargestellt sind.

Im Codeausschnitt 4.16 befinden sich die benötigten modellspezifischen Bindings für das ODE-SECIR-Modell von MEmilio. Die Population des Modells teilt sich in verschiedene Infektionszustände und Altersgruppen auf. Um die Population zu binden, müssen also die Klassen der beiden Faktoren und die zugehörigen Indizes gebindet werden. Zudem muss das passende *CustomIndexArray* mit einem *MultiIndex* Python zugänglich gemacht werden. Ein *MultiIndex* wird für jedes *CustomIndexArray* mit mehr als einem Index benötigt und fasst die Indizes in einer Klasse zusammen. Zuletzt wird die Population übertragen. Dafür bekommt die dazugehörige Funktion *bind_Population()* als Template-Parameter alle Kategorien der Population übergeben.

4.3.2 Verbesserung der Population-Bindings

In diesem Abschnitt soll nochmal genauer auf die Verknüpfung zwischen den modellspezifischen Bindings und dem *Pymio*-Framework eingegangen werden. Dafür wird das Binden der Klasse *Population* angeschaut. Das Beispiel wird kurz erklärt und anschließend eine Verbesserung vorgestellt, wodurch die modellspezifischen Bindings einfacher zu erstellen sind.

```
1  // Binding der Infection States
2  pymio::iterable_enum<mio::InfectionState>(m, "InfectionState
    ") \ ...
3
4  // Bindings der einzelnen Indizes
5  pymio::bind_Index<mio::InfectionState>(m, "
    Index_InfectionState");
6  pymio::bind_Index<mio::AgeGroup>(m, "Index_AgeGroup");
7
8  // Binding der Klasse AgeGroup
9  py::class_<mio::AgeGroup, mio::Index<mio::AgeGroup>>(m, "
    AgeGroup")
10 .def(py::init<size_t>());
11
12 // Binding des Arrays mit MultiIndex
13 pymio::bind_MultiIndex<mio::AgeGroup, mio::InfectionState>(m
    , "Index_Agegroup_InfectionState");
14 pymio::bind_CustomIndexArray<mio::UncertainValue, mio::
    AgeGroup, mio::InfectionState>(m, "PopulationArray");
15
16 pymio::bind_Population<mio::AgeGroup, mio::InfectionState>(m
    , "Population");
```

Quelltext 4.16: Benötigter modellspezifischer Code zum Binden der Population des ODE-SECIR-Modells. (C++)

Zum Schreiben der modellspezifischen Bindings müssen die Klassen einzeln gebindet werden und alle Kategorien der Population müssen in der *bind_Population()*-Funktion gesetzt werden. Das Ziel ist es diese Schritte in das Pymio-Framework auszulagern

und somit die Arbeit beim Erstellen der Bindings eines neuen ODE-Modells zu verringern.

Die im Quelltext 4.16 dargestellten Funktionen werden alle für die Klasse *Population* benötigt. Je nachdem welche Kategorien sie besitzt, müssen unterschiedliche Klassen übertragen werden. Der Gedanke ist daher, nur die Funktion *bind_Population()* aufzurufen und intern die benötigten Aufrufe der anderen Binding-Funktionen zu setzen. Somit wird innerhalb der Population-Bindings die Funktion *bind_CustomIndexArray()* aufgerufen. Diese kümmert sich um die benötigten Indizes. Dafür wird zum einen rekursiv über alle Kategorien aus den Template-Parametern gegangen und für jede Kategorie die Funktion *bind_Index()* aufgerufen. Zum anderen prüft sie, ob ein *MultiIndex* benötigt wird und macht es gegebenenfalls auch zugänglich. Die Bindings der Klassen der Kategorien lassen sich nicht verallgemeinern und somit nicht in das Pymio-Framework auslagern.

Als weiterer Vorteil werden hiermit alle Indizes eines *CustomIndexArray* automatisch gebindet. Dadurch kann es aber dazu kommen, dass versucht wird die gleiche Klasse doppelt im selben Modul zu übertragen. Dies führt beim Importieren des Moduls in Python zu einem *ImportError*. Daher werden die Aufrufe der Binding-Funktionen in den Population- und Array-Bindings durch Error Handling geschützt. Wenn der explizite Fehler auftritt, wird er abgefangen und ohne die Klasse ein zweites Mal zu binden fortgefahren.

```
1  pymio::iterable_enum<mio::InfectionState>(m, "InfectionState
     ")
2  // ...
3  pymio::bind_Population(m, "Population", mio::Tag<mio::Model
     ::Populations>{});
4  py::class_<mio::AgeGroup, mio::Index<mio::AgeGroup>>(m, "
     AgeGroup")
5  .def(py::init<size_t>());
```

Quelltext 4.17: Verbesserter modellspezifischer Code zum Binden der Population des ODE-SECIR-Modells. (C++)

Der Binding-Funktion der Population sollen nun nicht mehr die Kategorien als Template-Argumente übergeben werden. An Stelle davon wird der Funktion ein weiterer Parameter hinzugefügt. Dieser erhält eine Instanz der Template-Klasse *Tag* der MEmilio Kernbibliothek mit dem Attribut der Population des ODE-Modells. Die Klasse *Tag* ist ein leeres Template-Struct, dass nur dazu da ist, die Typinformationen zu speichern und an die Funktion zu übergeben. Die Funktion *bind_Population()* leitet die Kategorien durch Template-Argument-Deduction ab [4].

Der entstehende Code mit beiden Verbesserungen ist im Quelltext 4.17 zu sehen. Da die Population- und die Modellklassen bei allen ODE-Modellen den gleichen Namen bekommen, muss zum Binden der Population idealerweise nur der korrekte Namensraum gesetzt werden.

5 Automatische Codegenerierung von modellspezifischen Python-Bindings

Im folgenden Kapitel wird ein Prototyp zur automatischen Codegenerierung der modellspezifischen Python-Bindings, welche für die C++-Modelle erstellt werden, entwickelt. Dies soll die Entwickler*innen der C++-Modelle entlasten und den Prozess der Übertragung nach Python beschleunigen. Es wird auf die verwendeten Bibliotheken eingegangen, verschiedene Ansätze bei der Entwicklung vorgestellt und die Umsetzung mit Quelltextbeispielen erklärt. Innerhalb der Codegenerierung der Python-Bindings werden die Analysetools des Compiler-Frontends Clang [37] verwendet. Daher wird Clang zuerst kurz vorgestellt. Danach wird anhand eines Ablaufdiagramms der Programmentwurf des Codegenerators beschrieben. Dies gibt die Basis, um im Anschluss auf die einzelnen Aspekte und Klassen einzugehen.

5.1 Bibliothek Clang

Clang ist ein Compiler-Frontend für C++ und weitere Programmiersprachen. Es wird zusammen mit dem Compiler-Backend LLVM benutzt. Das Ziel war es eine Alternative zum GCC-Compiler zu bieten. Dabei sollte Clang eine bessere Diagnose und Integration mit integrierten Entwicklungsumgebungen (IDE) bereitstellen. Ein weiteres Ziel war es eine Lizenz anzubieten, die mit kommerziellen Produkten kompatibel ist [37].

Als Compiler-Frontend führt es die lexikalische, syntaktische und semantische Analyse durch. Aufgrund des modularen Aufbaus bietet Clang viele Möglichkeiten zur

Erweiterung und Veränderung der parserinternen Abläufe. So bietet es Zugriff auf den Abstrakten Syntaxbaum (Abk. AST vom Englischen abstract syntax tree), der bei der syntaktischen Analyse des Codes erstellt wird. Die Entwickler*innen von Clang haben mit der Bibliothek LibClang [11] die Funktionalität innerhalb von Python verfügbar gemacht.

5.2 Programmentwurf

Für einen guten Überblick über das Konzept der automatischen Codegenerierung der Python-Bindings wird der grober Programmentwurf erläutert. Dafür wird der Programmablauf in Abbildung 5.1 dargestellt.

Am Anfang des Programmablaufs steht der **Scanner**. Die Aufgabe des Scanners ist es, aus dem vorliegenden **C++-Quellcode** eines Modells, die für die Codegenerierung relevanten Informationen zu extrahieren und an einen **Zwischencode** zu übergeben. Der Zwischencode ist eine eigene Datenklasse, die als Schnittstelle zwischen dem Scanner und Generator liegt.

Für das Einlesen des C++-Codes wird Clang benutzt. Clang führt eine syntaktische Analyse des Codes durch und erstellt daraus einen AST. Der AST beinhaltet eine abstrakte Darstellung der Funktionalität des Quellcodes und ist dabei durch die Verwendung von Clang auf die Syntax von C++ spezialisiert.

Die Codegenerierung ist in Python geschrieben. Daher wird zur Weiterverarbeitung des ASTs LibClang verwendet. Der Scanner durchläuft den AST rekursiv und untersucht dabei nacheinander die Knoten. Die relevanten Informationen eines Knoten werden im Zwischencode gespeichert und abschließend führt der Scanner noch eine Überprüfung durch.

Der **Generator** hat die Aufgabe, die Informationen aus dem Zwischencode zu verwenden und damit den **Zielcode**, die Python-Bindings eines Modells, zu erstellen. Der

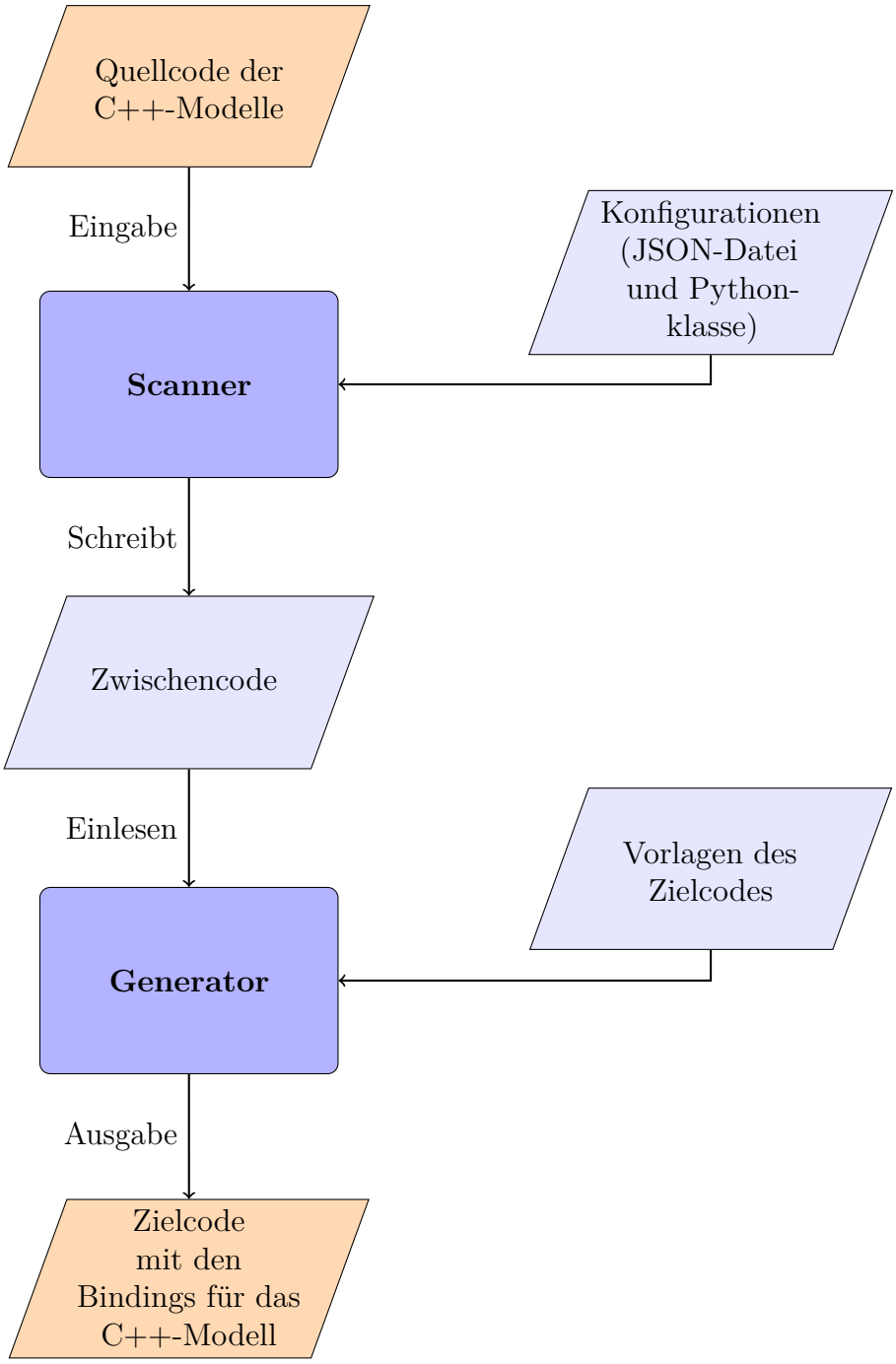

Abbildung 5.1: Ablauf der Codegenerierung. Die blauen Kästen stehen für Teile des Codegenerators, die orangenen den Input und Output. Trapezförmige Kästen stellen Daten dar und die anderen Logikklassen des Codegenerators.

Zielcode liegt dafür als **Vorlage** mit Platzhaltern vor. An deren Stelle kann flexibel generierter Code eingefügt werden. Dafür wandelt der Generator die Informationen der Zwischensprache in Strings um. Diese werden mit dem Begriff **Ersetzungsstrings** definiert, da sie im letzten Schritt die dazugehörigen Platzhalter ersetzen, wodurch aus der Vorlage der vollständige Zielcode generiert wird.

5.3 Der Scanner

5.3.1 Anforderungen an den Sourcecode

Um ein Modell mit Hilfe der Codegenerierung zu binden, werden einige Anforderungen vorausgesetzt. Damit wird ein korrekter Ablauf der Generierung gewährleistet und die Komplexität beschränkt. Hierdurch muss nicht, wie beim Erzeugen von Maschinensprache aus C++-Code, eine vollständige Übersetzung des C++-Codes in die passenden Python-Bindings erfolgen. Bei solch einer Übersetzung wird der generierte Code zum großen Teil von dem Aussehen der Eingabe bestimmt.

Beim Ansatz dieser Arbeit wird vom fertigen Zielcode ausgegangen und daran die aus dem C++-Code zu extrahierenden Informationen definiert. Um die Informationen anhand von Mustern zu erkennen, führt der Scanner eine syntaktische Analyse mit Hilfe des ASTs durch.

Die resultierende Syntax der Python-Bindings wird dabei vom bereits vorliegenden Pymio-Framework vogegeben. Das Framework ist somit auch eine Voraussetzung und verringert die Arbeit des Generators. Aus den vorgegebenen Funktionen und zusätzlichen Voraussetzungen für die Codegenerierung ergeben sich folgende Anforderungen an den C++-Code:

- Es wird eine Modellklasse mit dem Namen *Model* benötigt. Sie besitzt mindestens eine Population und ein Parameterset.
- Alle Modellklassen befinden sich innerhalb eines Namensraums ohne Unterebenen.

- Es braucht eine *cpp*-Datei, die beim Kompilieren alle Ressourcen des Modells verwendet. Dafür muss die C++-Kernbibliothek vorhanden sein.

Außerdem wird von den Benutzer*innen benötigt:

- Clang und LibClang sind in der gleichen Version installiert.
- *CompilationCommands* für die Hauptdatei des Sourcecodes ist vorhanden.
- Mindestens Python 3.7 sowie die Pakete *dataclasses-json* und *tempfile* sind installiert.

5.3.2 Konfiguration

Um die Anforderungen an den Quellcode umzusetzen und gleichzeitig für die Zukunft anpassbar zu machen, sowie die Möglichkeit anwender- und modellspezifische Voraussetzungen anzugeben, wird eine Konfiguration eingeführt, welche in zwei Teile getrennt wird.

Zum einen gibt es eine Konfigurationsdatei im JSON-Format, welche die spezifischen Voraussetzungen beinhaltet. Die Angaben müssen für jedes Modell von dem/der Anwender*in neu gesetzt werden. Die Konfigurationsdatei enthält beispielsweise den Namen der Modellklasse, den Build-Ordner des C++-Quellcodes oder den gewünschten Namen des generierten Python-Moduls. Durch das Verwenden des JSON-Formats sind die Angaben einfach zu finden und anzupassen. Zudem können mehrere Konfigurationsdateien gespeichert und zwischen diesen gewechselt werden.

Zum anderen gibt es eine Konfigurationsklasse. Sie entspricht einer Datenklasse mit Einlesefunktion für JSON-Dateien und wird durch die Python-Pakete *dataclasses* [6] und *dataclasses-json* [24] bereitgestellt. Das Benutzen einer Datenklasse wird im Abschnitt des Zwischencodes genauer erklärt. Die Konfigurationsklasse liest die Informationen der Konfigurationsdatei ein, speichert sie ab und wird anschließend an den Scanner innerhalb seiner Initialisierung übergeben. Während der Initialisierung der Datenklasse werden zudem weitere Parameter definiert. Die Parameter sollen

in der Regel nicht verändert werden und sich an den definierten Anforderungen orientieren.

Jedoch entsprechen zum jetzigen Zeitpunkt nicht alle Modelle des Projektes den Konventionen. Trotzdem soll es möglich sein die Codegenerierung auf solche Modelle anzuwenden, was durch das Definieren der Daten in einer eigenen Konfigurationsklasse ohne großen Aufwand möglich ist. Außerdem verbessert es die Wartbarkeit und Erweiterbarkeit des Codegenerators. So können beispielsweise die Anforderungen in Zukunft durch Veränderungen an nur wenigen Stellen angepasst werden.

5.3.3 Erzeugen des Abstrakten Syntaxbaums

Um die Funktionsweise von Clang darzustellen, wird im folgenden Abschnitt verschiedene Alternativen der Erstellung des ASTs betrachtet. Die dabei auftretenden Probleme werden herausgearbeitet und daraus Voraussetzungen für die Codegenerierung formuliert, welche bereits im Abschnitt der Anforderungen genannt wurden.

Der erste Ansatz ist eine sehr simple Umsetzung. Mit LibClang wird für jede Quelldatei, also Header- und Source-Dateien, die zu dem mathematischen Modell gehören, ein eigener AST gebaut. Der Ansatz benötigt von dem/der Anwender*in keine großen Voraussetzungen. Es werden die Pfade zu allen Dateien des relevanten Modells benötigt. Da diese, wie in Kapitel 3 erklärt, alle in einem eigenen Ordner liegen, braucht der Scanner nur den Pfad zu dem entsprechenden Ordner und kann damit alle Header- und Source-Dateien selbstständig suchen.

LibClang werden die Dateien einzeln übergeben, an Stelle eines vollständigen Programms. Somit handelt es sich dabei nicht um kompilierbaren und ausführbaren Code. Eingebundene Dateien und Bibliotheken werden nicht erkannt, da sie dem Compiler nicht bekannt sind. Trotz dessen kann LibClang aus einem unvollständigem Code einen AST erstellen. Dies liegt an der gewählten Strategie zum Umgang mit Fehlern beim Bau des ASTs. Das Ziel ist es selbst unvollständigen Code so gut wie möglich darzustellen, um damit dem/der Anwender*in die beste Möglichkeit zur Identifizierung von Fehlern bereitzustellen [36]. Beim Auftreten eines Fehlers gibt es 3 Möglichkeiten, wie Clang damit umgeht:

1. Den Fehler selbständig beheben.
2. Ungültigen Knoten darstellen mit einem Statusindikator, der ihn als fehlerhaft markiert.
3. Ungültige Knoten aus dem AST entfernen.

Es wird also versucht den Code so gut wie möglich darzustellen, was das aber am Ende bedeutet ist nicht ganz klar. Mit Berücksichtigung der dritten Fehlerstrategie kann man davon ausgehen, dass die erhaltenen ASTs unvollständig sind und die Menge an fehlenden Informationen für andere Modelle als Eingabe nicht vorhersehbar sind. Für die Codegenerierung relevante Informationen könnten fehlen und somit das Ausführen fehlschlagen. Daher ist der Ansatz trotz der geringen Anforderungen nicht sinnvoll.

Beim nächsten Ansatz wird die Möglichkeit, beim Bau des ASTs mit LibClang zusätzliche Anweisungen anzugeben, verwendet. Hiermit lassen sich die Abhängigkeiten des Codes definieren, wodurch es dem Compiler möglich ist, einen AST für einen funktionierenden Code zu bauen. Um die benötigten Anweisungen für eine korrekte Kompilierung herauszufinden, wird eine *compile_commands.json* verwendet. Die JSON-Datei kann beim Kompilieren eines Projektes mit *CMake* automatisch erstellt werden und beinhaltet für jede cpp-Datei die benötigten Anweisungen. Mit LibClang lassen sich die Anweisungen für die passende cpp-Datei wieder aus der *compile_commands.json* entnehmen, wofür LibClang die Klasse *CompilationDatabase* anbietet.

Hieraus entstehen größere Anforderungen. Es wird eine *compile_commands.json* benötigt, die das relevante Modell beinhaltet. Außerdem braucht das Modell mindestens eine cpp-Datei, es kann also nicht nur aus Headern bestehen, und die cpp-Datei muss alle relevanten Dateien direkt oder indirekt inkludieren, damit sie berücksichtigt werden. Für solch eine cpp-Hauptdatei wird der AST mit den passenden Kompilierungsanweisungen erstellt.

Nach der Implementierung des zweiten Ansatzes beinhaltete der AST deutlich mehr Informationen und Knoten. Trotzdem war der AST nicht vollständig und beim Bau kam es zu Fehlermeldungen. LibClang findet die Standard C-Headers, z.B.

stddef.h, nicht. Das Problem ist auch nicht damit gelöst, den Pfad zu einem Ordner mit den C-Headers als Kompilierungsanweisungen zu übergeben, da dabei neue Fehlermeldungen entstehen. Jedoch entsteht das Problem lediglich beim Verwenden von LibClang und nicht bei Clang.

Der dritte Ansatz benutzt also Clang um den AST mit den gleichen Kompilierungsanweisungen zu bauen [30]. Dafür wird Clang von Python aus mit der Bibliothek *subprocess* [7] auf der Kommandozeile aufgerufen. Der Befehl *-emit-ast* weist Clang an einen AST zu bauen und auszugeben. Die Ausgabe kann von Python aus verwendet werden, jedoch befindet sich der AST noch nicht im passenden Format für LibClang. Dafür wird der AST kurzfristig in eine Datei geschrieben und anschließend von LibClang wieder ausgelesen.

Die Datei wird im Anschluss nicht benötigt. Aufgrund ihrer einzigen Verwendung zur Übertragung des AST aus der Ausgabe der Kommandozeile in ein verwendbares Python-Objekt, wird diese als temporäre Datei erstellt. Dafür wird die Bibliothek *tempfile* [8] verwendet. Eine temporäre Datei kann mittels eines *Context Managers* zum Schreiben und Lesen verwendet werden und wird mit dem Beenden des Managers automatisch gelöscht.

Hier braucht es nun weitere Anforderungen an den/die Anwender*in. Es wird Clang benötigt. Damit das Übertragen des ASTs von der Kommandozeile zu Python funktioniert, müssen Clang und LibClang in der gleichen Version installiert sein, da sonst die ASTs der beiden möglicherweise nicht miteinander kompatibel sind.

5.3.4 Extrahieren der Informationen

Mit der ausgearbeiteten Implementierung kann nun ein vollständiger AST für die Weiterverwendung mit LibClang gebaut werden. Um das Aussehen eines ASTs zu erklären, wird in Abbildung 5.2 der Ausschnitt des ASTs der *Model*-Klasse, bei der Analyse des ODE-SEIR-Modells, angeschaut.

In den Knoten des ASTs befinden sich *Cursor*, welche die Repräsentation von LibClang für ein Objekt aus dem dargestellten Code sind. Ein Cursor hält unterschiedliche

```
mio ... <CursorKind.NAMESPACE>
 |       <SourceLocation file model.h, line 29, column 1>
 |
 |__oseir ... <CursorKind.NAMESPACE>
 |    |          <SourceLocation file model.h, line 31, column 1>
 |    |
 |    |__Model ... <CursorKind.CLASS_DECL>
 |    |    |          <SourceLocation file model.h, line 38, column 5>
 |    |    |
 |    |    |__CompartmentalModel ... <CursorKind.CXX_BASE_SPECIFIER>
 |    |    |                          <SourceLocation file model.h, line
 |    |    |                          74, column 1>
 |    |    |                          Referenziert: CompartmentalModel
 |    |    |                          <CursorKind.STRUCT_DECL>
 |    |    |
 |    |    |__public ... <CursorKind.CXX_ACCESS_SPEC_DECL>
 |    |    |             <SourceLocation file model.h, line 42, column 5>
 |    |    |
 |    |    |__Model() ... <CursorKind.CONSTRUCTOR>
 |    |    |               <SourceLocation file model.h, line 43, column 9>
 |    |    |
 |    |    |__get_derivatives() ... <CursorKind.CXX_METHOD>
 |    |                             <SourceLocation file model.h, line
 |    |                             48, column 9>
```

Abbildung 5.2: AST der Klasse *Model* eines ODE-SEIR-Modells. Dabei wird nur ein Ausschnitt des vollständigen ASTs dargestellt. Knoten die zwei Ebenen unter dem Modellknoten liegen, werden nicht abgebildet. Für jeden Knoten werden die Informationen *CursorKind* und *SourceLocation* angegeben.

Informationen über das Objekt. Im Folgenden werden einige wichtige Informationen beschrieben, welche alle Cursor besitzen.

- Ein Cursor besitzt einen **Anzeigenamen** (engl. displayname) des Objekts. Im dargestellten AST sind Beispiele *mio* oder *Model()*.
- Die **Cursor-Art** (engl. kind) beschreibt das Objekt im Knoten. Cursor-Arten gibt es beispielsweise für Namensräume oder Konstruktoren. In der Darstellung des ASTs sind diese durch *CursorKind.[...]* angegeben.
- Jeder Knoten des ASTs kennt seine **Kinder-** und seinen **Elternknoten**.

- Jeder Cursor besitzt eine Liste mit allen **Tokens** und den dazugehörigen Zeichenketten seines Objekts aus dem Quellcode. Die Tokens werden von Clang durch einen *Tokenizer* während der lexikalischen Analyse erstellt.
- Ein Cursor besitzt immer eine **Anfangs-** und **Endposition**. Positionen beinhalten die Ursprungsdatei, sowie die Zeile und Spalte. Im dargestellten AST sind die Startpositionen der Cursor abgebildet.

Darüber hinaus besitzen manche Cursor-Arten noch zusätzliche Informationen. Beispielsweise ist im abgebildeten AST der Knoten *CompartmentalModel* eine Referenz und hat daher einen Zeiger auf den Knoten des referenzierten Objekts.

Zum Extrahieren der Informationen muss der AST traversiert werden, wobei jeder Knoten besucht wird. Die Reihenfolge, in der die Knoten besucht werden, entspricht der Tiefensuche. Dabei wird ein Pfad zuerst komplett in die Tiefe durchlaufen, bis ein Blatt erreicht wird. Implementiert wurde das Traversieren im Scanner durch rekursive Funktionsaufrufe.

Beim Besuchen eines Knotens wird der entsprechende Cursor überprüft. Als erstes wird betrachtet, ob sich der Cursor in den relevanten Namensräumen befindet. Relevant sind *mio*, der Namensraum der Kernbibliothek von MEmilio, und der modellspezifische Namensraum, welcher in der Konfiguration gesetzt wurde. Dabei müssen die Klassen direkt in den relevanten Namensräumen liegen und nicht in einem Unternamensraum.

Die wichtige Information ist dabei die Art des Cursors. Je nach Art, wird eine entsprechende Funktion aufgerufen, welche den Cursor genauer untersucht. Dafür wurde ein *Switch* umgesetzt, zu sehen im Codeausschnitt 5.1. Die Funktion erhält die Art des Cursors und gibt mit Hilfe des Dictionary *switch* die richtige Funktion zurück. Das Dictionary enthält als Schlüssel die Cursor-Arten und jeder Schlüssel enthält einen Wert mit der Überprüfungsfunktion der dazugehörigen Cursor-Art.

```python
def switch_node_kind(self, kind):
  switch = {
    CursorKind.ENUM_DECL: self.check_enum,
    CursorKind.ENUM_CONSTANT_DECL: self.check_enum_const,
    CursorKind.CLASS_DECL: self.check_class,
    CursorKind.CLASS_TEMPLATE: self.check_class,
    CursorKind.CXX_BASE_SPECIFIER: self.check_base_specifier,
    CursorKind.CONSTRUCTOR: self.check_constructor,
    CursorKind.STRUCT_DECL: self.check_struct,
    CursorKind.TYPE_ALIAS_DECL: self.check_type_alias
  }
  return switch.get(kind, lambda *args: None)
```

Quelltext 5.1: Implementierung eines switch-Statements in Python mit einem Dictionary. (Python)

Im Dictionary befinden sich nur die Cursor-Arten, welche Informationen enthalten, die für den Scanner interessant sind. Für alle anderen wird eine Lambda-Funktion zurückgegeben, welche nur den Wert *None* zurückgibt. Somit wird beim Überprüfen einer nicht benötigten Cursor-Art direkt mit dem nächsten Knoten weiter gemacht. Beim Betrachten der einzelnen Cursor in den Überprüfungsfunktionen werden die benötigten Informationen in den Zwischencode geschrieben.

5.3.5 Finalisierung

Nach dem Extrahieren der Daten führt der Scanner zwei abschließende Aufgaben durch.

Zum einen überträgt er die Informationen, welche später vom Generator benötigt werden, aus der Konfiguration in den Zwischencode. Wie dem Programmentwurf und der Abbildung 5.1 zu entnehmen, soll die Konfiguration nur vom Scanner genutzt werden. Somit ist der Scanner dafür zuständig alle relevanten Informationen aus der Konfiguration zu verarbeiten und zu übertragen. Wenn er dies nicht macht, hat der Generator später auf die Informationen der Konfiguration keinen Zugriff. Der

Zwischencode ist dabei die Schnittstelle zwischen Scanner und Generator und ist somit für den Transfer von Informationen vom Scanner zum Generator verantwortlich.

Als zweite Aufgabe wird die Vollständigkeit und Korrektheit der extrahierten und an den Zwischencode übergebenen Daten geprüft. Wenn ein Fehler beim Einlesen des C++-Codes geschieht, soll dies hier erkannt werden. Ansonsten würde bei fehlenden Daten der Generator nicht funktionieren und entweder beim Ausführen abbrechen oder eine fehlerhafte Datei generieren. In dem Fall wäre es schwer für die Entwickler*innen zu erkennen in welchem Schritt der Codegenerierung der Fehler seinen Ursprung hat. Somit macht die rechtzeitige Überprüfung das Programm für die Entwickler*innen einfacher zu debuggen. Dabei ist es wichtig möglichst viele potentielle Fehlerquellen frühzeitig zu identifizieren. Nur so kann man diese beim Schreiben des Scanners berücksichtigen und den Fehler innerhalb des Scanners prüfen. Mögliche Fehlerquellen sind dabei:

- **Fehlende Konfigurationsdaten** können zum Absturz des Scanners führen, aber auch zu einem nicht vollständigen Zwischencode.
- **Falsche Konfigurationsdaten** führen zu unterschiedlichsten Fehlern. Zum Beispiel kann durch einen falsch angegebenen Modellnamen das Modell nicht gefunden werden oder bei einem falschen Quelldateipfad keine Informationen extrahiert werden.
- Bei einem **Unvollständiger Quellcode** fehlen dem Scanner Informationen. Dadurch könnte der Zwischencode, den der Scanner erstellt, unbrauchbar sein.

Anzumerken ist, dass es meist unmöglich ist, alle Fehlerquellen zu überprüfen. Vor allem bei einer Erweiterung der Funktionalität des Scanners kommen auch neue Fehlerquellen hinzu.

5.4 Der Zwischencode

Die Bezeichnung *Zwischencode*, im Englischen *Intermediate Representation* genannt, kommt aus dem Bereich des Compilerbaus. Er entsteht im Verlauf eines Übersetzungsprozesses und stellt die Verbindung zwischen der Ausgangssprache und der Zielsprache her. Der Zwischencode soll alle benötigten Informationen in einer möglichst effizienten Struktur darstellen. Dadurch soll die anschließende Generierung des Codes in der Zielsprache vereinfacht werden. Weitere Vorteile des Zwischencodes sind, dass auf ihn gut Programmoptimierungen durchgeführt werden können und als Schnittstelle ein Austauschformat bietet.

In diesem Programmentwurf verbindet der Zwischencode den Scanner und den Generator. Durch die Einführung einer Schnittstelle zwischen den beiden Klassen, kann die Funktionalität einer Klasse verändert werden, ohne die andere zu beeinflussen. Somit spielt es beispielsweise für den Generator keine Rolle, woher die Daten des Zwischencodes kommen. Dadurch kann zum Beispiel in Zukunft ein Programm ohne einen Scanner implementiert werden, wovon in Kapitel 6 ein Programmentwurf vorgestellt wird.

```
1  @dataclass
2  class IntermediateRepresentation:
3    namespace           : str   = None
4    model_class         : str   = None
5    python_module_name  : str   = None
6    parameterset        : str   = None
7    parameterset_wrapper: str   = None
8    simulation_class    : str   = None
9    project_path        : str   = None
10   target_folder       : str   = None
11   enum_populations    : dict  = field(default_factory=dict)
12   model_init          : list  = field(default_factory=list)
13   model_base          : list  = field(default_factory=list)
14   population_groups   : list  = field(default_factory=list)
15   age_group           : dict  = field(default_factory=dict)
```

Quelltext 5.2: Darstellung der Datenklasse *IntermediateRepresentation*. (Python)

Der Zwischencode wird als Python-*dataclass* realisiert. Das *dataclasses*-Paket vereinfacht das Erstellen einer Klasse mit dem Nutzen der Speicherung von Daten. Der vollständige Code ist im Codeausschnitt zu sehen. Die Attribute der Klasse werden ab Zeile 3 definiert. Untypisch für Python muss dafür der Datentyp der Attribute angegeben werden. Dadurch wird bereits beim Setzen eines Attributs überprüft, ob der korrekte Datentyp benutzt wird. Als letztes kann noch ein Standardwert für jedes Attribut gesetzt werden. Attribute ohne Standardwert müssen beim Initialisieren eines Objektes der Datenklasse einen passenden Wert übergeben bekommen. Durch den Decorator in Zeile 1 im Codeausschnitt 5.2 wir die Klasse als *dataclass* definiert, wodurch intern eine entsprechende ___*init*___-Funktion, mit den entsprechenden Eingabeparametern, dessen Standardwerte und der Übertragung dieser auf die Objektattribute automatisch erstellt wird.

5.5 Der Generator

Aus dem vollständigen Zwischencode soll der Zielcode, die Python-Bindings eines Modells, erzeugt werden. Die Generierung verwendet dafür eine *Vorlage mit Platzhaltern* und ein *Dictionary mit Ersetzungsstrings*. Die *Generator*-Klasse behandelt dabei alle benötigten Ressourcen und enthält die Logik zur Erstellung des Zielcodes. Um den Ablauf davon zu verstehen, wird ein Teil des Zielcodes und seine Erzeugung betrachtet. Das Beispiel betrachtet den Zielcode für das Binding der Klassen *Model* und *CompartmentalModel* des ODE-SEIR-Modells. Der fertige Zielcode sieht folgendermaßen aus:

```
1 pymio::bind_CompartmentalModel<mio::oseir::InfectionState,
      Populations, mio::oseir::Parameters>(m, "ModelBase");
2 py::class_<mio::oseir::Model, mio::CompartmentalModel<mio::
      oseir::InfectionState, Populations, mio::oseir::Parameters
      >>(m, "Model")
3 .def(py::init<>());
```

Quelltext 5.3: Zielcode der modellspezifischen Python-Bindings der Klassen *Model* und *CompartmentalModel* des ODE-SEIR-Modells. (C++)

Um den Zielcode zu erhalten, braucht es zunächst die passende Vorlage, welche vorgibt, wie der endgültige Code aussehen soll. Abschnitte, die für jedes Modell identisch sind, können direkt in die Vorlage übernommen werden. Modellspezifische Abschnitte werden durch einen Platzhalter ersetzt. Die fertige Vorlage ist im Quelltext 5.4 dargestellt. Platzhalter sind durch die Form *${Identifikator}* markiert, wobei der Identifikator den entsprechenden Platzhalter benennt, um diesen später beim Ersetzen zu erkennen.

```
1 pymio::bind_CompartmentalModel<${model_base_templates}>(m, "${
      model_class}Base");
2 py::class_<${namespace}${model_class}, ${model_base}>(m, "${
      model_class}")
3 ${model_init}
```

Quelltext 5.4: Vorlage der modellspezifischen Python-Bindings der Klassen *Model* und *CompartmentalModel* für ODE-Modelle. (C++)

An dieser Stelle sieht man gut, warum Konventionen bei der Implementierung der Modelle sehr wichtig sind. Je mehr Gemeinsamkeiten die Modelle aufweisen, desto weniger Informationen benötigen Platzhalter in der Vorlage und desto einfacher hat es der Generator. Vorteile durch ähnliche Codestrukturen gelten sowohl für die C++-Modelle, als auch für das Pymio-Framework. Idealerweise verringern sich dadurch auch die Informationen, die der Scanner erkennen muss, wodurch der Programmieraufwand reduziert wird.

Um nun aus der Vorlage den Zielcode zu generieren, müssen die Platzhalter mit dem entsprechenden modellspezifischen Code ersetzt werden. Dafür wird ein *Dictionary* erstellt, welches als Schlüssel die Identifikatoren der Platzhalter verwendet. Jeder Schlüssel erhält als Wert den Ersetzungsstring, welcher an die Stellen des Platzhalters gesetzt wird. Die Identifikatoren kann man dabei auch mehrfach benutzen, wodurch beim Ersetzen an den Stellen der Platzhalter der gleiche Ersetzungsstring eingefügt wird.

Nicht jedes Modell benötigt alle Platzhalter. Wenn eine Stelle ungenutzt bleibt, wird dort ein leerer String eingefügt. Somit enthält der Zielcode keine unerwünschten Zeichen. Nach dem Ersetzen sollte also kein Platzhalter im Zielcode vorhanden sein, sonst ist bei der Generierung ein Fehler passiert.

Die Ersetzungsstrings für das Dictionary werden mit den Informationen des Zwischencodes erstellt. Einige dieser, wie zum Beispiel der String für den Namensraum, können direkt aus dem Zwischencode ausgelesen werden. Andere sind komplexer und werden daher in Hilfsfunktionen aufgebaut. Aus dem Anwendungsbeispiel wird im Quelltext 5.5 die Hilfsfunktion für die Modellkonstruktoren (Identifikator *model_init*) abgebildet.

```
1  def model_init(intermed_repr):
2    str = ""
3    for init in intermed_repr.model_init:
4      if len(init["type"]) > 1:
5        continue
6      elif len(init["type"]) == 0:
7        str += "    .def(py::init<>())\n\t"
8      else:
9        str += (
10         "    .def(py::init<{type}>(), py::arg(\"{name}\"))\n\t
       "
11       ).format(
12         type = init["type"][0],
13         name = init["name"][0]
14       )
15   return str.rstrip() + ";\n"
```

Quelltext 5.5: Erstellung des Binding-Codes der Modell-Konstruktoren als String. (C++)

Die Funktion bekommt als Eingabe den Zwischencode übergeben und verwendet davon die Informationen der Konstruktoren der Modellklasse. Dabei wurde für jeden Konstruktor die Parameternamen und dazugehörigen Typen gespeichert. In der Funktion werden nacheinander für jeden Konstruktor die passende Binding-

Funktion *def()* erstellt. Zurzeit werden jedoch nur Konstruktoren mit weniger als zwei Eingabeparametern berücksichtigt.

Zum Erzeugen des Zielcodes aus der Vorlage und die damit verbundene Ersetzung der Platzhalter wird das Python-Paket *string.Template* verwendet.

Für die benötigten *Includes* wurde ein passender Platzhalter gesetzt, der vorübergehend durch ein Kommentar mit Anweisungen für die Benutzer*innen ersetzt wird. Sie müssen daraufhin selbst die richtigen *Includes* an die Stelle des Kommentars setzen. Zu einem späteren Zeitpunkt kann die automatische Erkennung und das Einfügen der *Includes* ergänzt werden.

5.6 Programmieraufwand durch einen komplexen Codegenerator

Im folgenden Abschnitt wird darauf eingegangen, welche Faktoren die Menge an Aufgaben der automatischen Codegenerierung der Python-Bindings beeinflussen und es wird beschrieben, wie diese unterschiedlichen Faktoren den Programmieraufwand beim Entwickeln des Generators verändern. Zusätzlich werden Ansätze zur Reduktion dargestellt.

Angefangen wird mit dem Sourcecode der Modelle. Dabei bestimmt die Anzahl an unterschiedlichen Modellarten, die erkannt werden sollen, den Aufwand beim Programmieren des Generators. Verschiedene Modellarten können sich in der Struktur ihrer Implementierung und den benötigten Informationen stark unterscheiden. Ein ABM benötigt beispielsweise eine Welt, die als Umgebung für die Agenten basiert, was für ein ODE-SIR-Modell nicht in gleicher Form existiert. ODE-Modelle haben beispielsweise keine Auflösung in einzelne Agenten, sondern rechnen in Populationsgruppen. Aus diesem Grund wurde auch entschieden, die Codegenerierung auf eine Modellart zu beschränken. Als Ausblick wird in Kapitel 6 kurz auf die Integration des ABMs in die Codegenerierung eingegangen.

Aber auch innerhalb einer Modellart können die Strukturen und Funktionalitäten der einzelnen Modelle sich stark unterscheiden. Bei einem Modell mit mehr Funktionen braucht es generell mehr Informationen, um das Modell nach Python zu binden. Um den Faktor einzuschränken, wurden zum einen die Konfigurationen in das Programm eingeführt und zum anderen generelle Anforderungen an die Modelle gesetzt. Dadurch werden die möglichen Eingaben verkleinert. Der Codegenerator kann bestimmte Formate als vorausgesetzt annehmen und muss andere Fälle nicht berücksichtigen. Dabei ist es aber immer ein Abwägen zwischen der Nutzerfreundlichkeit und dem Programmieraufwand. Mehr Einschränkungen bedeuten für die Anwender*innen mehr Arbeit beim Erstellen eines Modells. Sie müssen die Einschränkungen nachlesen und Angaben zu dem verwendeten Modell an die Codegenerierung übergeben. Weniger Einschränkungen bedeuten dagegen mehr Arbeit für die Entwickler*innen, da für den Codegenerator eine bessere Logik implementiert werden muss.

Ein konkretes Beispiel dafür ist die Modellklasse. Ideal wäre es, wenn der Codegenerator automatisch anhand von klaren Eigenschaften die Modellklasse erkennt. Dafür bräuchte es jedoch eine deutlich komplexere interne Logik. Daher wird die Namenskonvention für die Modellklasse vorgeschrieben. Ein weiteres Beispiel ist die Erkennung der Altersauflösung der Population. Auch diese sollte der Codegenerator autonom erkennen. An Stelle davon wird im jetzigen Programmprototyp in der Konfigurations-JSON ein *Boolean*-Wert gesetzt, welcher angibt, ob Altersgruppen verwendet werden. Der Generator weiß also, wie er mit den Altersgruppen umgehen muss, erkennt aber nicht, ob sie berücksichtigt werden müssen.

Ein weiterer Faktor ist das Format der Python-Bindings. Für die Bindings der Modelle werden die Methoden des Pymio-Frameworks verwendet. Je besser das Framework ist, desto weniger Code braucht es in den einzelnen Binding-Dateien der Modelle. Das konnte anhand des Beispiels der Binding-Funktion der Klasse *Population* in Kapitel 4 gezeigt werden. Dabei konnte gezeigt werden, dass es möglich ist die modellspezifischen Bindings zu verkleinern und die benötigten Informationen zu verringern. Als Folge wird die automatische Generierung der Bindings der Modelle einfacher. Ein weitergeführter Ansatz des Beispiels wird als Ausblick in Kapitel 6 diskutiert.

6 Evaluation des Generators

6.1 Testdurchlauf

Nach der Implementierung des Prototyps, soll die Codegenerierung der Python-Bindings validiert werden. Bisher wurden für die Programmierung des Codegenerators das ODE-SEIR- und ODE-SECIR-Modell [22] benutzt. Nun soll das ODE-SECIRVVS-Modell [19] als Sourcecode des Generators verwendet werden. Das Modell ist wie in Kapitel 3 erklärt eine Erweiterung des ODE-SECIR-Modells.

Es soll geprüft werden, ob der Codegenerator auch für neue Modelle funktioniert. Außerdem wird betrachtet, wie er mit unbekannten Funktionalitäten umgeht. Als Erwartungshaltung sollte klar sein, dass Funktionalitäten, die dem Codegenerator noch nicht bekannt sind, nicht in die Python-Bindings übertragen werden können. In dem Test wird lediglich geschaut, ob die bereits implementierten Funktionalitäten erkannt werden und der Codegenerator fehlerfrei läuft.

Nach dem Setzen der Konfigurationen wird die automatische Generierung gestartet. Die Generierung konnte ausgeführt werden, jedoch sind die erstellten modellspezifischen Bindings nicht fehlerfrei. Die Fehler lagen an falsch definierten Namensräume einiger Klassen in der Vorlage, was einfach ausgebessert werden konnten. An den Stellen wurde der passende Identifikator eingefügt. Beim erneuten Ausführen sind diesmal auch die erhaltenen Dateien korrekt. Alle bereits implementierten Funktionalitäten aus dem ODE-SECIR-Modell wurden auch für das ODE-SECIRVVS-Modell erkannt und erstellt.

Um die erhaltenen Dateien in die Python-Bindings zu integrieren, müssen diese in den entsprechenden Ordner verschoben werden. Zudem muss ein Modul für das

neue Modell in die *CMake*-Datei hinzugefügt werden. Die beiden Schritte wurden durchgeführt und die Python-Bindings konnten gebaut werden.

Danach wurde noch getestet, ob das ODE-SECIRVVS-Modell bereits mit eingeschränkter Funktionalität in Python verwendbar ist. Dabei konnte das Anwendungsbeispiel zumindest ohne Fehler ausgeführt werden, die erhaltenen Ergebnisse waren jedoch keine Zahlenwerte, sondern *NaN*-Werte. Trotz dessen sind die generierten Python-Bindings sinnvoll, da sie als Grundlage der vollständigen Bindings des Modells dienen.

6.2 Ausblick & Diskussion

Die Codegenerierung wurde als erster Prototyp implementiert. Währenddessen ist man auf viele Limitation, aber auch Möglichkeiten für die Zukunft gestoßen. Daher soll abschließen auf einige Aspekte als Ausblick eingegangen und kritisch betrachtet werden.

Abwärtskompatibilität der Modelle

Um die Codegenerierung zu bewerten, muss abgeschätzt werden, ob der Zusatzaufwand, der durch das Erstellen des Codegenerators entsteht, von den daraus resultierenden Vorteilen übertroffen wird. Generell gilt für die Codegenerierung, wie auch für die Python-Bindings, eine Abwärtskompatibilität bei ODE-Modellen. Wenn die automatische Generierung für ein Modell funktioniert, dann funktioniert es auch für alle Modelle mit weniger Funktionalitäten. Wenn dagegen ein Modell erweitert wird, muss diese Funktion auch für die Python-Bindings und den Codegenerator geschrieben werden. Diese Annahme konnte anhand des Testdurchlaufes mit dem ODE-SECIRVVS-Modell bestätigt werden.

Das Problem ist, dass nicht jede Modellart von einer Abwärtskompatibilität profitiert. Bei den ODE-Modellen ist dies definitiv sinnvoll, da implementierte Funktionen benutzt werden müssen. Wenn beispielsweise der Wert eines Parameters nicht bekannt

ist, da die Datenlage der Pandemie noch nicht ausreicht, und somit ein Infektionszustand unbrauchbar wird, ist es besser ein Modell zu wählen, welches diese Funktionalität nicht bietet. Benutzer*innen brauchen also sehr individuelle Modell und müssen diese auch als eigene Modell-Paket implementieren. Außerdem macht es in dem Zusammenhang auch Sinn, immer das simpelste Modell zu nehmen, welches für den Anwendungsfall ausreicht, da so weniger Störfaktoren und möglicherweise unbekannte Parameter in der Modellierung benutzt werden.

Bei einem ABM können implementierte Funktionen beim Ausführen einfach unbenutzt bleiben. Dadurch scheint es dort eher der Fall zu sein, dass ein Modell mit allen Funktionalitäten ausreicht. Somit würde diese Modellart nicht von der Abwärtskompatibilität profitieren und eine automatische Generierung wäre ein großer Aufwand ohne richtigen Vorteil. Die Aussage müsste noch genauer überprüft werden.

Andere Modellarten

Im letzten Abschnitt hat man den Vorteil innerhalb einer Modellart betrachtet. Darüber hinaus können auch unterschiedliche Modellarten kompatible Strukturen aufweisen. So kann eine Modellart von der Logik der Scanner-Klasse der ODE-Modelle profitiert. Wenn sich die Modellarten aber zu sehr unterscheiden, braucht jedes Modell einen eigenen Scanner und möglicherweise sogar eine komplett eigene Codegenerierung. Auch dies kann wieder zu einem größeren Mehraufwand führen, als selbstständig die Python-Bindings zu schreiben. Zum jetzigen Zeitpunkt würde man annehmen, dass die ABMs nicht kompatibel mit den ODE-Modellen sind. IDE-Modelle dagegen könnten eine ähnliche Struktur aufweisen und sollten daher genauer analysiert werden.

Das Softwarepaket MEmilio besitzt jedoch an vielen Stellen noch kein festes Konzept. Viele Modelle sind noch relativ neu in der Entwicklung. So gibt es zur Zeit noch kein fertiges IDE-Modell. Dadurch ist nicht eindeutig, wie die Modelle in Zukunft aussehen werden. Auch an einer festen Struktur innerhalb des Projektes wird noch gearbeitet, dabei würde von festen Namenskonventionen, Ordnerstrukturen und festgelegte Anforderungen an ein Modell auch die Codegenerierung profitiert.

Notwendigkeit der Codegenerierung

In Kapitel 4 wurde ein Beispiel zur Verbesserung des Pymio-Frameworks dargestellt. Dadurch konnten einige Funktionsaufrufe aus den modellspezifischen Bindings ausgelagert werden. Zusätzlich wurde in Kapitel 5 erklärt, dass mit solch einer Änderung die Aufgaben der Codegenerierung verkleinert werden.

Ein mögliches Konzept für die Python-Bindings ist, noch mehr in das Pymio-Framework auszulagern, sodass zum Binden beispielsweise nur eine Funktion *bind_model()* aufgerufen wird. Intern werden eigenständig die Membervariablen des übergebenen Modells, ähnlich wie beim Beispiel der Population, abgeleitet.

Außerdem könnten unterschiedliche Modellarten über eine Abfrage im Template-Argument erkannt werden und durch Überladen der Funktion das Modell richtig gebindet werden. Zur Abfrage einer Modellart gibt es bereits erste Funktionen in der Kernbibliothek von MEmilio (siehe Quelltext 6.1). Dafür braucht es ein Konzept der Modellklassen, in dem festgelegt wird, welche Eigenschaften die Klasse mindestens aufweisen muss.

```
1 template <class M>
2 using is_compartment_model =
3   std::integral_constant<bool, (is_expression_valid<
      eval_right_hand_side_expr_t, M>::value &&
4                     is_expression_valid<
      get_initial_values_expr_t, M>::value )>;
5 }
```

Quelltext 6.1: Abfrage einer C++-Modellklasse. Prüft ob die als Template übergebene Klasse dem Konzept eines ODE-SIR-typischen Modells entspricht. (C++)

Wenn die Informationsmenge, die fürs Binden über die C++-Modelle benötigt wird, zu klein wird, ist der Aufwand der Instandhaltung des Codegenerators nicht mehr tragbar. Dagegen spricht jedoch, dass solch ein Konzept wahrscheinlich nie komplett funktioniert. Es wird immer Teile des Bindings geben, welche modellspezifisch definiert werden müssen, wie zum Beispiel die Werte eines Enums.

Erweiterung durch eine Generierung der C++-Modelle

Während der Arbeit wurden die Strukturen der ODE-Modelle betrachtet und Gemeinsamkeiten für die automatische Generierung der Python-Bindings ausgenutzt. Dies könnte man auch auf eine Generierung der C++-Modelle erweitern. Über eine bessere API könnte so ein Modell erstellt werden, ohne die Eigenheiten von C++ oder des MEmilio Softwarepakets zu kennen. Außerdem könnte repetitiver Quelltext automatisch erstellt werden. Beispielsweise könnten die Infektionszustände als Liste angegeben werden und das dazugehörige Enum wird automatisch erstellt.

Bei solch einem Ansatz könnte für die automatische Generierung der Python-Bindings der Scanner aus dem Programmablauf genommen werden. Die benötigten Informationen würden direkt aus der API der C++-Modell in den Zwischencode geschrieben.

7 Fazit

Es wurde ein funktionierender Codegenerator für die modellspezifischen Python-Bindings des Softwarepakets MEmilio programmiert. Er ist für einfache ODE-SIR-typische Modelle benutzbar und wurde mit dem Gedanken der Erweiterbarkeit für neue Funktionalitäten implementiert.

Die Verwendung der Analysetools von Clang für die automatische Codegenerierung hat gut funktioniert. Der Ansatz kann auch für andere Anwendungsfälle als Grundlage dienen und bietet noch viel Potential. Vor allem die Programmstruktur und das Erstellen des ASTs für die Analyse in Python kann in anderen Projekten übernommen werden.

Bei der Umsetzung wurde das ODE-SEIR- und ODE-SECIR-Modell benutzt. Die Aspekte Infektionszustände, Parameter, eine Wrapper-Klasse der Parameter, eine Simulationsklasse für das Modell, Altersgruppen und Ortsauflösung werden bei der automatischen Generierung berücksichtigt. Somit wurde die Zielsetzung der Bachelorarbeit abgeschlossen. Während des Entwicklungsprozesses wurden Problematiken herausgearbeitet und mögliche Ziele für die Zukunft in Kapitel 6 diskutiert.

Literaturverzeichnis

[1] Python Packaging Authority. setuptools 65.3.0. `https://pypi.org/project/scikit-build/`, 2022. Accessed on 28.08.2022.

[2] Fred Brauer, Zhisheng Shuai, and P. van den Driessche. Dynamics of an age-of-infection cholera model. *Mathematical Biosciences & Engineering*, 10(5&6):1335–1349, 2013.

[3] Marc Choisy, Jean-François Guégan, and Pejman Rohani. *Mathematical Modeling of Infectious Diseases Dynamics*, chapter 22, pages 379–404. Wiley, 08 2006.

[4] cppreference. Template argument deduction. `https://en.cppreference.com/w/cpp/language/template_argument_deduction`, 2022. Accessed on 29.08.2022.

[5] DLR-SC. memilio. `https://github.com/DLR-SC/memilio`, 2022. Accessed on 28.08.2022.

[6] Python Software Foundation. dataclasses — Data Classes. `https://docs.python.org/3/library/dataclasses.html`, 2022. Accessed on 25.08.2022.

[7] Python Software Foundation. subprocess — Subprocess management. `https://docs.python.org/3/library/subprocess.html`, 2022. Accessed on 25.08.2022.

[8] Python Software Foundation. tempfile — Generate temporary files and directories. `https://docs.python.org/3/library/tempfile.html`, 2022. Accessed on 25.08.2022.

[9] Wikibooks – Die freie Bibliothek. C++-Programmierung: Polymorphie. `https://de.wikibooks.org/wiki/C%2B%2B-Programmierung:_Polymorphie`, 2014. Accessed on 25.08.2022.

[10] Google. Google Python Style Guide. `https://google.github.io/styleguide/pyguide.html`, 2022. Accessed on 27.08.2022.

[11] Tao He. libclang 14.0.6. `https://pypi.org/project/libclang/`, 2022. Accessed on 25.08.2022.

[12] HZI. Helmholtz Zentrum für Infektionsforschung. `https://www.helmholtz-hzi.de/de/`, 2022. Accessed on 23.08.2022.

[13] Wenzel Jakob. First steps. `https://pybind11.readthedocs.io/en/stable/basics.html`, 2017. Accessed on 25.08.2022.

[14] Wenzel Jakob. pybind11 2.10.0. `https://pypi.org/project/pybind11/`, 2022. Accessed on 25.08.2022.

[15] Alexander Keimer and Lukas Pflug. Modeling infectious diseases using integro-differential equations: Optimal control strategies for policy decisions and applications in Covid-19. preprint on webpage at `https://www.researchgate.net/publication/341265820_Modeling_infectious_diseases_using_integro-differential_equations_Optimal_control_strategies_for_policy_decisions_and_Applications_in_COVID-19`, 05 2020.

[16] William Ogilvy Kermack, A. G. McKendrick, and Gilbert Thomas Walker. A contribution to the mathematical theory of epidemics. *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character*, 115(772):700–721, 1927.

[17] Cliff C. Kerr, Robyn M. Stuart, Dina Mistry, Romesh G. Abeysuriya, Katherine Rosenfeld, Gregory R. Hart, Rafael C. Núñez, Jamie A. Cohen, Prashanth Selvaraj, Brittany Hagedorn, Lauren George, Michał Jastrzębski, Amanda S. Izzo, Greer Fowler, Anna Palmer, Dominic Delport, Nick Scott, Sherrie L. Kelly, Caroline S. Bennette, Bradley G. Wagner, Stewart T. Chang, Assaf P. Oron,

Edward A. Wenger, Jasmina Panovska-Griffiths, Michael Famulare, and Daniel J. Klein. Covasim: An agent-based model of Covid-19 dynamics and interventions. *PLOS Computational Biology*, 17(7):1–32, 07 2021.

[18] Sahamoddin Khailaie, Tanmay Mitra, Arnab Bandyopadhyay, Marta Schips, Pietro Mascheroni, Patrizio Vanella, Berit Lange, Sebastian Binder, and Michael Meyer-Hermann. Development of the reproduction number from coronavirus SARS-CoV-2 case data in Germany and implications for political measures. *BMC Med*, 19(32), 2021.

[19] Wadim Koslow, Martin J. Kühn, Sebastian Binder, Margrit Klitz, Daniel Abele, Achim Basermann, and 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. *PLOS Computational Biology*, 18(5):1–26, 05 2022.

[20] A. M. Kuchling. Functional Programming HOWTO. `https://docs.python.org/3/howto/functional.html?highlight=iterators#functional-howto-iterators`, 2001. Accessed on 25.08.2022.

[21] Martin J. Kühn, Daniel Abele, Sebastian Binder, Kathrin Rack, Margrit Klitz, Jan Kleinert, Jonas Gilg, Luca Spataro, Wadim Koslow, Martin Siggel, Michael Meyer-Hermann, and Achim Basermann. Regional opening strategies with commuter testing and containment of new SARS-CoV-2 variants in Germany. *BMC Infectious Diseases*, 22(333), 2022.

[22] Martin J. Kühn, Daniel Abele, Tanmay Mitra, Wadim Koslow, Majid Abedi, Kathrin Rack, Martin Siggel, Sahamoddin Khailaie, Margrit Klitz, Sebastian Binder, Luca Spataro, Jonas Gilg, Jan Kleinert, Matthias Häberle, Lena Plötzke, Christoph D. Spinner, Melanie Stecher, Xiao Xiang Zhu, Achim Basermann, and Michael Meyer-Hermann. Assessment of effective mitigation and prediction of the spread of SARS-CoV-2 in Germany using demographic information and spatial resolution. *Mathematical Biosciences*, 339(108648), 2021.

[23] Andrew T. Levin, William P. Hanage, Nana Owusu-Boaitey, Kensington B. Cochran, Seamus P. Walsh, and Gideon Meyerowitz-Katz. Assessing the age

specificity of infection fatality rates for COVID-19: systematic review, meta-analysis, and public policy implications. *European Journal of Epidemiology*, 35:1123–1138, 2020.

[24] Charles Li. dataclasses-json 0.5.7. `https://pypi.org/project/dataclasses-json/`, 2022. Accessed on 25.08.2022.

[25] Jian Li, Tao Xiang, and Linghui He. Modeling epidemic spread in transportation networks: A review. *Journal of Traffic and Transportation Engineering (English Edition)*, 8(2):139–152, 2021. Transportation Planning and Operations for Covid-19 Epidemic and Other Emergencies.

[26] Saskia Morwinsky, Natalie Nitsche, and Enrique Acosta. Covid-19 fatality in Germany: Demographic determinants of variation in case-fatality rates across and within german federal states during the first and second waves. *Demographic Research*, 45:1355–1372, 2021.

[27] Aleksandra Orlowska, Christos Chrysoulas, Zakwan Jaroucheh, and Xiaodong Liu. Programming Languages: A Usage-Based Statistical Analysis and Visualization. In *2021 The 4th International Conference on Information Science and Systems*, page 143–148. Association for Computing Machinery, 2021.

[28] Piero Poletti, Marcello Tirani, Danilo Cereda, Filippo Trentini, Giorgio Guzzetta, Valentina Marziano, Sabrina Buoro, Simona Riboli, Lucia Crottogini, Raffaella Piccarreta, Alessandra Piatti, Giacomo Grasselli, Alessia Melegaro, Maria Gramegna, Marco Ajelli, and Stefano Merler. Age-specific SARS-CoV-2 infection fatality ratio and associated risk factors, Italy, February to April 2020. *Eurosurveillance*, 25, 08 2020.

[29] PYPL. PYPL PopularitY of Programming Language. `https://pypl.github.io/PYPL.html`, 2022. Accessed on 23.08.2022.

[30] Gil Barbosa Reis. clang_tu_from_ast.py. `https://gist.github.com/gilzoide`, 2021. Accessed on 25.08.2022.

[31] The scikit-build team. scikit-build 0.15.0. `https://pypi.org/project/setuptools/`, 2022. Accessed on 28.08.2022.

[32] Hongjing Shi, Zhisheng Duan, and Guanrong Chen. An SIS model with infective medium on complex networks. *Physica A: Statistical Mechanics and its Applications*, 387(8):2133–2144, 2008.

[33] Peter Spreeuwenberg, Madelon Kroneman, and John Paget. Reassessing the Global Mortality Burden of the 1918 Influenza Pandemic. *American Journal of Epidemiology*, 187(12):2561–2567, 09 2018.

[34] Olivera Stojanović, Johannes Leugering, Gordon Pipa, Stéphane Ghozzi, and Alexander Ullrich. A Bayesian Monte Carlo approach for predicting the spread of infectious diseases. *PLOS ONE*, 14(12):1–20, 12 2019.

[35] Development Support. Klassen und Objekte. `https://dev-supp.de/codes/cplusplus/klassen-objekte`, 2022. Accessed on 25.08.2022.

[36] The Clang Team. CFE Internals Manual. `https://clang.llvm.org/docs/InternalsManual.html`, 2022. Accessed on 25.08.2022.

[37] The Clang Team. Clang: a C language family frontend for LLVM. `https://clang.llvm.org/`, 2022. Accessed on 25.08.2022.

[38] TIOBE. TIOBE Index for August 2022. `https://www.tiobe.com/tiobe-index/`, 2022. Accessed on 23.08.2022.

[39] Guido van Rossum and Nick Coghlan. PEP 8 – Style Guide for Python Code. `https://peps.python.org/pep-0008/`, 2001. Accessed on 27.08.2022.

[40] Guido van Rossum and Ka-Ping Yee. Pep 234 – Iterators. `https://peps.python.org/pep-0234/`, 2001. Accessed on 27.08.2022.

[41] Keeling MJ Wearing HJ, Rohani P. Appropriate models for the management of infectious diseases. *PLoS Med*, 2(7), 07 2005.