Praktikum „Ingenieursmäßige Software-Entwicklung“: Deep Learning für Palladio-Modelle

Ewald Rode
Matrikel-Nr. 1727019

Institut für Programmstrukturen und Datenorganisation (IPD)
Betreuender Mitarbeiter: Dipl.-Ing. Daniel Zimmermann

1 Einführung

Palladio ist eine integrierte Softwarearchitektur-Modelllierungsumgebung auf Basis der Eclipse Rich Client Plattform. Palladio ermöglicht Entwicklern Palladio Component Modell(PCM)-Instanzen zu erstellen und aus diesen Modellen Vorhersagen über deren Leistung abzuleiten. Mögliche Ergebnisse dieser Simulationen sind unter anderem Skalierbarkeitsprobleme, Leistungsengpässe oder mangelnde Zuverlässigkeit.

Ein Verfahren zur Simulation von Softwarearchitektur wird in der Palladioerweiterung PerOpteryx implementiert. PerOpteryx ist ein Optimierungsframework zur Verbesserung komponentenbasierter Softwarearchitekturen, basierend auf modellbasierten Qualitätsvorhersagetechniken. PerOpteryx nimmt dafür komplette PCM-Instanzen als Eingabe, manipuliert bestimmte Parameter anhand eines genetischen Algorithmus und analysiert die so kreierten Modelle. Die Analyse geschieht dann mit Hilfe des Layered-Queuing-Network-Solver(LQNS)[3]. PerOpteryx transformiert hierfür die kreierte PCM-Instanz zu einem neuen Modell entsprechend dem LQN XML Schema[1]. LQNS berechnet anhand des Modells unter anderem Antwortzeit und Kosten des Systems. Die Ergebnisse der Simulationen können zur Auswahl einer guten Softwarearchitektur beitragen.

Ziel des Praktikums ist die Anbindung des Business Reporting System(BRS)[7] an das Framework für Maschinelles Lernen TensorFlow[2]. Da PerOpteryx auf der Eclipse Rich Client

[1] http://www.sce.carleton.ca/rads/lqns/lqn-documentation/schema/
[2] https://www.tensorflow.org

Plattform basiert und nicht ohne Weiteres von Maschinellen Lernverfahren verwendet werden kann, die auf TensorFlow basieren, wird die Transformation von PCM zu LQN XML Schema in Python implementiert.

Dieses Praktikum wurde von Ewald Rode und Frederic Born bearbeitet. Der Fokus dieses Berichts liegt auf den durchgeführten Aufgaben von Ewald Rode und verweist an entsprechenden Stellen auf den Bericht von Frederic Born.

2 Grundlagen

In diesem Kapitel werden Grundlagen erläutert, die für dieses Praktikum relevant sind.

2.1 Palladio Component Modell (PCM)

Palladio ist ein Ansatz zur Definition von komponentenbasierten Softwarearchitekturen. Das PCM ist eine domänenspezifische Modellierungssprache, deren Zweck es ist Leistungsprognosen für Softwarearchitekturen zu ermöglichen [1]. Alle PCM-Dateien liegen im XML-Format vor. Jedes Element innerhalb der PCM-Dateien besitzt eine eindeutige Identifikationsnummer(Id), die es ermöglicht von anderen Elementen auf diese zu verweisen. Hierdurch wird eine Navigation innerhalb und zwischen einzelnen Dateien ermöglicht. Die zur Transformation des PCM in das LQN XML Schema benötigten PCM Dateien und deren Relationen untereinander sind in Abbildung 1 dargestellt. Im Folgenden werden die

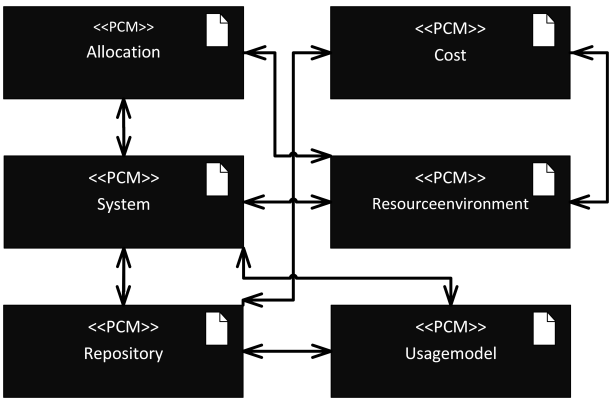

Abbildung 1: Abhängigkeiten zwischen PCM-Dateien.

in diesem Praktikum relevanten PCM-Dateien mit Hilfe von Becker et al. [1] erläutert:

- ***.allocation** wird verwendet um den Allokations-Kontext einer Komponente anhand des Inputs zu erhalten. Ein Spaltenname der Eingabe zeigt auf den Namen eines Allokations-Kontexts innerhalb der Allokations-Datei. Dieser Allokations-Kontext zeigt auf die ID eines Assembly-Kontexts (aus der System-Datei).
- ***.system** beinhaltet Assembly-Kontexte, die auf die Id einer Komponente des Repositories verweist.

- ***.repository** enthält Komponenten, Schnittstellen und Datentypen. Das Repository wird bei der Erstellung von Komponenten-Interface-Prozessoren verwendet (siehe Abschnitt 3). Hierfür sind aus dem Repository alle Komponenten, deren Interfaces, sowie deren dazugehörige Aktionen und Operationen zu entnehmen.
- ***.usagemodel** beschreibt Nutzerverhalten. Unter Anderem werden Wahrscheinlichkeiten des Eintretens bestimmter Aktionen(siehe Repository) spezifiziert.
- ***.resourceenvironment** spezifiziert alle Ressourcen des PCMs, dazu gehören Server und Verlinkungen.
- ***.designdecision** gibt alle möglichen Server-Allokationen für Komponenten (Verlinkung zu Ressourceenvironment), sowie alle möglichen Optionen für Assembled-Komponenten an (Verlinkung zu Repository).
- ***.cost** spezifiziert die Kosten für bestimmte CPU-Server (Verlinkung zu Resourceenvironment), sowie die Nutzung bestimmter Assembled-Komponents (Verlinkung zu Repository). Über den Rahmen dieses Praktikums hinaus wurde zusätzlich eine prototypische Implementierung der Kostenfunktion durchgeführt. Diese konnte jedoch nicht validiert werden, da dies den Umfang des Praktikums überschritten hätte.

Weitere Informationen sind der Wiki-Seite[3] des PCM zu entnehmen.

2.2 PerOpteryx

PerOpteryx ist eine Palladio-Erweiterung zur Optimierung komponentenbasierter Softwarearchitekturen auf Basis modellbasierter Qualitätsvorhersagetechniken [4]. Zur Performanzanalyse wird ein PCM durch PerOpteryx in Layered Queying Networks (LQN)[3] transformiert. Hierfür verwendet PerOpteryx PCM2LQN [5].

2.3 PCM2LQN Transformation

PerOpteryx greift bei der Optimierung der Softwarearchitektur auf das Plugin PCM2LQN [5] zu. PCM2LQN transformiert eine PCM-Instanz in eine LQN-Instanz. Diese entstandene LQN-Instanz ist eine XML Datei, im LQN XML Schema und dient im Folgenden dem LQN-Solver als Eingabe.

2.4 Layered Queueing Network XML Schema (LQN XML Schema)

LQN-Instanzen werden in LQXO Dateien gespeichert. Dabei handelt es sich um Dateien im XML Format, die einer besonderen Grammatik folgen. Es basiert auf der Arbeit von Wu [8] und wurde für die allgemeine Nutzung weiter verfeinert [6]. Eine Übersicht über den Aufbau eines LQN-Modells im LQN XML Schema ist in Abbildung 2 dargestellt, die der Online-Dokumentation [6] entstammt. Die wichtigsten Elemente des LQN XML Schema werden im Folgenden erklärt:

[3] `https://sdqweb.ipd.kit.edu/wiki/Palladio_Component_Model`

- **solver-params** sind Attribute, welche die Funktionsweise des LQN-Solver einstellen[3]. In diesem Praktikum wurden die Standard Werte aus Palladio und PerOpteryx für die Attribute übernommen:
  - conv_val: „0.001“
  - print_int: „10“
  - underrelax_coeff: „0.5“
- **processor** werden verwendet, um die Prozessoren im Modell zu definieren. Folgende Sheduling Typen werden in diesem Praktikum verwendet:
  - **PS** (Processor Sharing) bedeutet, dass der Prozessor alle Aufgaben gleichzeitig ausführt. Die Servicerate des Prozessors ist umgekehrt proportional zur Anzahl der ausgeführten Aufgaben.
  - **FCFS** (First-Come-First-Serve) bedeutet, dass Aufgaben in der Reihenfolge ihres Eingangs erledigt werden.
- **task** enthalten ein oder mehrere Elemente vom Typ Eintrag („entry“) und optional Elemente vom Typ „task-activities“, wenn der Typ des zugehörigen Eintrags „NONE“ entspricht.
- **entry** werden verwendet um Aufgaben („task“) zu definieren. Je nach Attributtyp eines Eintrags-Elements können Einträge auf eine von drei Arten spezifiziert werden. Für uns sind jedoch nur folgende beiden Arten relevant:
  - **PH1PH2** bedeutet, dass der Eintrag über Phasen spezifiziert wird. Die Phasen werden durch eine Aktivität innerhalb eines „entry-phase-activities“-Elements spezifiziert. Der Name dieser Aktivität muss eindeutig sein.
  - **NONE** bedeutet, dass der Eintrag über eine zusammenhängende Abfolge von Aktivitäten (Aktivitätsgraph) spezifiziert wird.
- **activity** bezeichnet eine Aktivität, die zur Erfüllung einer „task“ durchgeführt werden müssen. Eine Aktivität entspricht einer Aktion einer Service-Effect-Specification (SEFF) einer Komponente des Repositories. Aktionen können unterschiedliche Typen annehmen:
  - Start
  - Stop
  - Internal
  - External
  - Branch
  - EntryLevelSystemCall
  - Loop-Actions
  - SetVariable

SetVariable-Aktionen sind der einzige Typ von Aktion der nicht im LQN-Solver implementiert ist. Diese Problematik wird umgangen, indem SetVariable-Aktionen im LQN Modell nicht erstellt werden. Je nach Typ werden Aktivitäten auf unterschiedliche Weise erstellt. Dies ist ausführlich in den Action-Factories unter „Dokumentation/pypopteryx/factories“ dokumentiert.

- **precedence** wird verwendet, um eine Aktivität mit ihrer Nachfolger-Aktivität zu verknüpfen. Jedes Element dieses Typs enthält genau ein Pre-Element und optional ein Post-Element. Das Pre-Element beinhaltet die Vorgänger-, das Post-Element die Nachfolger-Aktivität. Im Falle der Branch-Action muss das Post-Element vom Typ „post-OR“ sein und beinhaltet je eine Nachfolger-Aktivität für eine der möglichen Branches. Jede mögliche Nachfolger-Aktivität besitzt außerdem eine Wahrscheinlichkeit, mit der ein Nutzer diese Aktivität auswählt. Diese Wahrscheinlichkeit ist im Usagemodel oder direkt in der Branch-Action definiert.

- **reply-entry** spezifiziert ein „reply-activity“-Element, welche die letzte Aktivität innerhalb einer Abfolge von Aktivitäten ist, um einen Eintrag, bzw. dessen „task“ erfüllt ist.

```
<lqn-model>
 <solver-params>
  <pragma/>
 </solver-params>
 <processor>
  <task>
 <entry>
  <entry-phase-activities>
   <activity>
   <synch-call/>
   <asynch-call/>
   </activity>
   <activity>...</activity>
  </entry-phase-activities>
   <entry-activity-graph>
    <activity/>
    <precedence/>
   </entry-activity-graph>
 </entry>
  <entry>...</entry>
  <task-activities>
   <activity/>
   <precedence/>
  </task-activities>
 </task>
 <task>...</task>
</processor>
<processor>...</processor>
</lqn-model>
```

Abbildung 2: Grundstruktur einer LQXO-Datei aus [6].

2.5 Layered Queueing Network Solver (LQNS)

Der LQN-Solver[3] wird verwendet um LQN-Modelle analytisch zu lösen[7]. Es handelt sich um ein Kommandozeilen-Werkzeug, dass LQN Modelle als Eingabe nimmt und auswertet. Genauere Informationen sind in der Dokumentation[3] zu finden.

3 Eigener Ansatz

Die Transformation von PCMs wird in PerOpteryx durch das Plugin PCM2LQN umgesetzt. Da PCM2LQN als Eclipse-Plugin realisiert ist, ist der Aufruf aus Python heraus nicht ohne Weiteres möglich. In diesem Praktikum wird daher eine neue Transformation implementiert. Um diese neue Transformation von der Alten abzugrenzen wird diese im Folgenden „**PyCM2LQN**“ genannt.

Bevor PyCM2LQN auf das PCM des BRS angewendet werden konnte, wurde die Transformation anhand eines weniger komplexen Beispiels entwickelt und getestet: dem Simple Heuristic Example[4]. Nachdem die Transformation für dieses Beispiel erfolgreich implementiert und getestet wurde, konnte PyCM2LQN auf das Ziel PCM des BRS erweitert werden.

Der entstandene Quellcode ist im Git-Repository[5] zu finden. Eine ausführliche Dokumentation[6] des Quellcodes inkl. aller Transformationsschritte kann als HTML Dokument eingesehen werden. Um durch die HTML Dokumentation zu navigieren wird, empfohlen den Ordner „Dokumentation/pyopteryx“ herunterzuladen und die Index-Datei in einem Browser zu öffnen.
Der in diesem Praktikum entwickelte Ansatz ist in Abbildung 3 dargestellt. Er besteht aus drei Schritten: einer Vorverarbeitung, der PyCM2LQN-Transformation und der Verwendung des LQNS. Diese Schritte werden im Folgenden erläutert.

3.1 Vorverarbeitung

Die Vorverarbeitung der PCM Dateien geschieht, um die darauf folgende Transformation zu vereinfachen. Ziel ist es wichtige Informationen zu extrahieren, die während der Transformation mehrmals benötigt werden und nicht mehrmals extrahiert werden sollen. Hierfür wird ein Dictionary erstellt, das sämtliche Informationen enthält mit deren Hilfe die PyCM2LQN-Transformation durchgeführt werden kann. Die Vorverarbeitung lässt sich in die Schritte Auslesen, Speichern und Informationsbeschaffung unterteilen, die im Folgenden genauer behandelt werden.

[4] https://sdqweb.ipd.kit.edu/wiki/PerOpteryx#Minimal_Example
[5] https://git.scc.kit.edu/udial/pyopteryx
[6] https://git.scc.kit.edu/udial/pyopteryx/tree/master/Dokumentation/pyopteryx

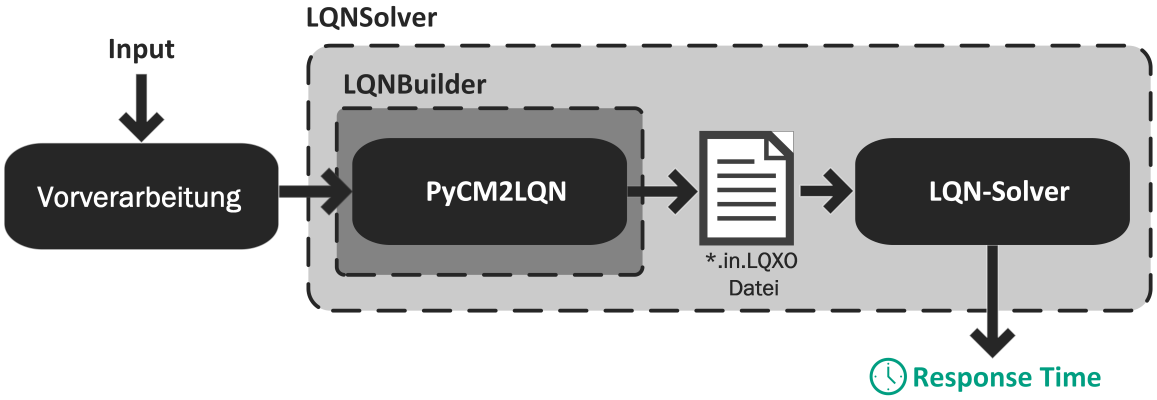

Abbildung 3: Ablauf des eigenen Ansatzes

3.1.1 Auslesen und Speichern

Bei den PCM Dateien handelt es sich ausschließlich um XML Dateien. Zum Auslesen, Verarbeiten und Schreiben von XML Dateien wurde im Praktikum die Python Bibliothek lxml[7] wegen seiner weiten Verbreitung, hohen Stabilität und einfachen Handhabung gewählt. Die mit lxml ausgelesenen und geparsten XML Dateien werden als Elementtree in einem Cache-Objekt gespeichert. Dieses Objekt wird der LQNSolver-Klasse und somit der LQNBuilder-Klasse übergeben, was verhindert das die Dateien mehrmals ausgewertet werden.

3.1.2 Informationsbeschaffung

Die Informationsbeschaffung soll dem LQNSolver Informationen zur Verfügung stellen, die von diesem mehrfach verwendet werden. Dabei wird zwischen CSV Informationen, szenariounabhängigen Informationen und szenarioabhängigen Informationen unterschieden. Diese werden im Folgenden jeweils erläutert:

CSV Informationen Bei der Entwicklung der Transformation nutzen wir von PerOpteryx generierte CSV Dateien, die unter anderem Komponentenallokationen, CPU Raten und Antwortzeiten spezifizieren. Wir benutzen diese CSV Dateien und die in ihnen spezifizierten Softwarearchitekturen als Eingabe für die Transformation. Wir benötigen daher Zuordnungen zwischen der CSV entnommenen Informationen und den Elementen der PCM Dateien. Im Folgenden werden die relevanten CSV Informationen erläutert.

keys_config Im keys_config Dictionary wird der Spaltenname der Antwortzeit der CSV Datei gespeichert. Der Zeilenname ist abhängig von den Elementen vom Typ „usageScenario_UsageModel“ aus dem Nutzungsmodell. Das Auslesen der Antwortzeit aus der CSV ist ausschließlich für die Evaluation des entwickelten Ansatzes relevant.

[7] https://pypi.org/project/lxml/

cpu_rates Die CPU Raten aller Server werden aus der CSV Datei ausgelesen und in einem Dictionary gespeichert. Die CPU Raten werden unter anderen für die Berechnung der Host-Demand-Means einiger Aktionen, sowie der Reaktionszeit des kompletten Systems gebraucht.

Szenariounabhängige Informationen Unter szenariounabhängigen Informationen verstehen wir Informationen die aus den PCM Dateien extrahiert werden können, ohne die Allokation von Komponenten und spezifische CPU-Raten zu kennen.

allocation_config Im allocation_config Dictionary wird die Zuordnung zwischen dem Name einer Allokation und seiner Id gespeichert. Benötigt wird dies, da wir unsere Szenarios aus einer CSV auslesen und wir von dem Namen einer Allokation auf deren Id schließen müssen.

assembled_component_config Ziel der assembled_component_config ist es, Informationen über alle assembled_components zu bekommen. Eine assembled_component beschreibt eine Softwarekomponente, die durch eine von mehreren Designalternativen umgesetzt werden kann. Ein Beispiel für eine assembled_component im BRS PCM ist ReportingAssembly_CoreGraphicEngine <CoreReportingEngine2> mit den beiden Designalternativen CoreOnlineEngine und CoreGraphicEngine.

server_config Ziel der server_config ist es ein Zuordnung zwischen der Id des Server-Elements und des Namens herzustellen. Benötigt wird diese Information an mehreren Stellen im Quelltext wenn nur die Id zur Verfügung steht, jedoch der Name des jeweiligen Servers benötigt wird.

Szenarioabhängige Informationen Unter szenarioabhängigen Informationen verstehen wir Informationen die aus den PCM Dateien mit Hilfe von Komponenten-Allokationen extrahiert werden können. In Rahmen unseres Praktikums werden die Komponenten-Allokationen, Informationen über zusammengesetzte Komponenten und CPU-Raten aus CSV Dateien entnommen. Diese Informationen können in Zukunft auch aus anderer Quelle, beispielsweise Tensorflow stammen.

graphical_mapping Die Komponente GraphicalReporting erfordert eine spezielle Behandlung (siehe Bericht von Frederic Born). Daher wird eine einzigartige Id abhängig der Allokation der Komponente generiert und im Graphical_mapping Dictionary gespeichert. Ziel ist es einen einzigartigen Namen für die Interface-Komponenten-Processor und alle Verweise auf diese korrekt erstellen zu können.

composite_components können auf einem bis mehreren Servern allokiert sein. Je nachdem auf wie vielen Servern eine composite_component allokiert ist, ist eine Fallunterscheidung zu unternehmen. Hierfür wird eine Zuordnung zwischen der Id einer composite_component und den Ids der Server auf denen die composite_component allokiert ist erstellt.

component_allocations Für die Transformation ist es wichtig zu wissen, auf welchem Server eine Komponente allokiert ist. Hierfür wird eine Zuordnung zwischen der Id einer Komponente und dem Server auf dem die Komponente allokiert ist erstellt. Einfache Komponenten werden in der Transformation anders behandelt als composite_components und assembled_components. Details sind im Bericht von Frederic Born im Kapitel PyCM2LQN Transformation zu finden.

cpu_to_seff_id_mapping Für die Transformation ist es wichtig zu wissen, auf welchem Server eine Komponente allokiert ist. Dieses Wissen wird der CSV entnommen. Jede Komponente besitzt Service Effect Spezifications (SEFF). Ziel ist es Zuordnung zwischen Server und auf ihm allokierten SEFFs zu erlangen.

assembled_components In assembled_components wird die Zuordnung zwischen der Id einer assembled_component und der Id ihrer Komponente gespeichert. In excluded_components werden die Ids von Komponenten gespeichert, die nicht verwendet werden. excluded_design_options stellt eine Zuordnung zwischen verwendeter Komponente und derer nicht verwendeter Designalternativer dar.

3.2 PyCM2LQN Transformation

Die Transformation wurde überwiegend von Frederic Born bearbeitet, daher werden die entsprechenden Punkte nur grob umschrieben. Details sind dem Bericht von Frederic Born zu entnehmen. Die PyCM2LQN Transformation ist in der Klasse „LQNBuilder“ implementiert und besteht aus mehreren Erstellungsschritten und einem Finalisierungsschritt.

1. Erstelle CPU-Prozessoren: Serverspezifsche Prozessoren.
2. Erstelle Linking-Resource-Prozessoren: Prozessor zur Spezifikation und Modellierung von Kommunikationressourcen und deren Verzögerungen.
3. Erstelle Nutzungsverzögerungs-Prozessoren: Prozessor zur Modellierung von Nutzungsverzögerungen.
4. Erstelle Komponenten-Interface-Prozessoren: Für jede Aktion eines Interfaces einer genutzten Komponente wird ein Prozessor erstellt. Somit lässt sich die unterschiedliche Ressourcennutzung jeder Aktion, sowie die Abhängigkeiten zwischen verschiedenen Aktionen modellieren.
5. Erstelle Nutzungsszenario-Prozessoren: Ermöglicht die Modellierung des Nutzerverhaltens.
6. Erstelle Schleifen-Prozessoren: Schleifen(engl. Loops) Aktionen setzen eine besondere Verarbeitung voraus und benötigen eigene Prozessoren.
7. Finalisierung: Es werden unnötige Elemente entfernt, die erstellten Elemente auf Fehler überprüft und abschließend die CPU Prozessoren mit Internal-Aktionen befüllt. Hier sind Stochastische Ausdrücke auszuwerten und als Host-Demand-Mean der Aktion zuzuweisen.

3.2.1 Stochastische Ausdrücke(StoEx)

Ressourcenspezifikationen können in PCM Dateien mit Hilfe Stochastischen Ausdrücken (StoEx)[8] spezifiziert werden. Code 1 zeigt exemplarisch wie solche Stochastischen Ausdrücke aussehen können.

```
(DoublePDF[(1.2;0.15)(1.3;0.4)(1.4;0.3)(1.5;0.15)]+1.2)/20/100
```

Code 1: Beispielstring einer StoEx Spezifikation, siehe BRS PCM [7]

Das StoEx Framework besteht aus Grammatik und Parser. Die Übersetzung das Parsers von Java in Python überschreitet den Rahmen des Praktikums und wurde daher nur vereinfacht umgesetzt. Hierfür wurde die Methode evaluate_stochastic_expression in „pyopteryx/utils/utils.py “ geschrieben. Diese Methode ist ausschließlich dazu fähig die im BRS PCM vorkommenden StoEx Spezifikationen annähernd auszuwerten. Diese Methode ist in der Zukunft durch eine Implementierung des StoEx Frameworks in Python zu ersetzen.

3.2.2 Genauigkeit von Berechnungen

Bei der Berechnung von Ausdrücken in Python unter Verwendung des Standard Datentyp float kommt es bei Gleitkommazahlen zur Approximation der letzten Nachkommastellen. Zur Vermeidung dieser Approximationen und zur Verwendung der korrekten Zahlen wurde an entsprechenden Stellen die Python Bibliothek Decimal[9] verwendet. Wie in Code 2 zu sehen, ermöglicht Decimal korrekt gerundete Gleitkommaarithmetik.

```
1.1 + 2.2 = 3.3000000000000003
Decimal(1.1) + Decimal(2.2) = Decimal(3.3)
```

Code 2: Beispiele für Gleitkommaarithmetik in Python

Decimal wurde bei der Berechnung von Ressourcenspezifikationen in „pyopteryx/utils/utils.py“ in der Methode calculate_expression_string sowie bei dem Vergleich zwischen berechneter Antwortzeit und tatsächlicher Antwortzeit in den Dateien „pyopteryx/usage_examples/test_lqn_builder.py “ und „pyopteryx/usage_examples/test_lqn_solver.py “ verwendet.

Zu beachten ist, dass mit Decimal Rundungsfehler vermieden werden können, diese jedoch nicht an anderer Stelle auszuschließen sind. Rundungsfehler sind weiterhin möglich, insbesondere wenn eine bereits Gleitkommazahlen bereits gerundet wurde, bevor diese mit Decimal weiterverwendet wird. Dementsprechend ist an allen Stellen des Quelltextes darauf zu achten, bei allen Berechnungen Decimal zu verwenden und damit ungewollte Rundungen zu vermeiden.

[8] https://sdqweb.ipd.kit.edu/wiki/StoEx_Stochastic_Expressions_in_PCM
[9] https://docs.python.org/3/library/decimal.html

3.3 LQN-Solver

Der LQN-Solver ist als Klasse realisiert, deren Aufgabe es ist, die Transformation via PyCM2LQN sowie die Auswertung via LQNSolver zu starten und im Anschluss die Ergebnisse zurückzugeben. Hierfür nimmt die Klasse bei der Initialisierung die vorverarbeitete Eingabe entgegen. Beim Aufruf der Methode evaluate_fitness() wird die Eingabe an den LQN-Builder weitergereicht. Mit der durch den LQN-Builder erstellten in.LQXO-Datei wird der LQN-Solver wie in Unterabschnitt 2.5 beschrieben durch einen Sub-Prozess ausgeführt und die Leistungsanalyse durchgeführt. Die Rückgabewerte der Methode sind Antwortzeit sowie Kosten der getesteten Softwarearchitektur.

4 Evaluation

Die Transformation durch PyCM2LQN wurde in diesem Praktikum anhand von zwei PCM Beispielen validiert. Für jedes PCM wurden Beispiele mit PerOpteryx generiert. Anschließend wurden diese Ergebnisse in Form von CSV-Dateien als Eingabedaten benutzt, um den eigenen Ansatz zu validieren. Dabei wurde die jeweilige durch PerOpteryx berechnete Reaktionszeit mit der berechneten Reaktionszeit des eigenen Ansatzes verglichen. Reaktionszeiten, die einen Wert von „Infinity“ annehmen, oder der direkt vorhergegangen Reaktionszeit entsprechen, werden nach Absprache mit dem Betreuer als unzulässig erachtet.

4.1 Simple Heuristics Example

Während der Entwicklung wurde PyCM2LQN gegen das Simple Heuristic Example validiert. Tabelle 1 veranschaulicht die Ergebnisse der Validierung. Von insgesamt 10000 Testbeispielen sind nach den oben genannten Annahmen 2800 Testbeispiele valide. Davon werden 2800 Ergebnisse korrekt berechnet. Es gibt somit keine fehlerhaften Ergebnisse und in 100% der Fälle wird die korrekte Reaktionszeit berechnet.

	Anzahl Testbeispiele
gesamt	10000
valide	2800
korrekt	2800
fehlerhaft	0
rundungsbedingt	0
Genauigkeit	100%

Tabelle 1: Testergebnisse für Simple Heuristic Example.

4.2 Business Reporting System

Tabelle 2 veranschaulicht die Ergebnisse der Validierung des BRS. Von insgesamt 12600 Testbeispielen sind nach den oben genannten Annahmen 12432 Testbeispiele valide. Davon

werden 12324 Ergebnisse korrekt berechnet. Die in Unterunterabschnitt 3.2.1 und Unterunterabschnitt 3.2.2 angesprochene Problematik führt in einigen Fällen mutmaßlich zu rundungsbedingten Fehlern. Diese Fehler sind eine Teilmenge der fehlerhaften Testbeispiele. Ein rundungsbedingter Fehler wird als solcher angesehen, wenn die absolute Abweichung zwischen dem berechneten Ergebnis und dem Referenzwert $\Delta_{\text{Fehler}} : 0 < \Delta_{\text{Fehler}} \leq 0.001$. Von den 108 fehlerhaften Ergebnissen sind 102 Fehler rundungsbedingte Fehler. Werden alle Abweichungen als Fehler gewertet, ergibt sich eine Genauigkeit von 99,13%. Werden die rundungsbedingten Abweichungen aus der Fehlermenge ausgeschlossen erreicht die Software eine Genauigkeit von 99,99%. Die maximale Differenz zwischen Reaktionszeiten des entwickelten Ansatzes (PyOpteryx) und dem Golstandard (PerOpteryx) für das BRS beträgt $\Delta_{max} = 0.03$.

	Anzahl Testbeispiele
gesamt	12600
valide	12432
korrekt	12324
fehlerhaft	108
rundungsbedingt	102
Genauigkeit	99,13% (abzgl. Rundungsfehler 99,99%)

Tabelle 2: Testergebnisse für Business Reporting System.

5 Ratschläge zur Fehlerbehebung und -vermeidung

In diesem Kapitel werden Ratschläge gegeben, wie Fehler bei der (Weiter-)Entwicklung des Projektes vermieden werden sollen. Des Weiteren werden häufige Fehlermeldungen und ihre Bedeutungen erklärt, sowie Hinweise gegeben, wie diese Fehler zu beheben sein könnten. Die hier gesammelten Ratschläge entstammen aus der Online-Dokumentation von Maly [6], sowie eigenen Erfahrungen, die während der Bearbeitung des Projektes gesammelt wurden. Um sich die hier genannten Fehlermeldungen anzeigen zu lassen, muss die „-w“-Flag aus dem Subprozess-Call in der LQNSolver-Klasse entfernt werden.

5.1 Richtlinien zur Fehlervermeidung

Um eine valide LQN Datei zu erstellen sind bei der Entwicklung der Transformation von PCM in LQN einige Richtlinien einzuhalten. Im Folgenden werden einige dieser Richtlinien aus der Onlinedokumentation von Maly [6] zitiert:

1. Alle Einträge müssen einen eindeutigen Namen haben.
2. Alle Aktivitäten müssen innerhalb einer bestimmten Aufgabe einen eindeutigen Namen haben.
3. Alle synchronen Anforderungen müssen ein gültiges Ziel haben.

4. Alle Weiterleitungsanforderungen müssen ein gültiges Ziel haben.
5. Alle Aktivitätsverbindungen (in Prioritätsblöcken) müssen sich auf gültige Aktivitäten beziehen.
6. Alle Antworten auf Aktivitäten müssen sich auf einen gültigen Eintrag beziehen.
7. Alle Aktivitätsschleifen müssen sich auf eine gültige Aktivität beziehen.
8. Jeder Eintrag hat nur eine Aktivität, die an ihn gebunden ist.

5.2 Häufige Fehlermeldungen und ihre Bedeutungen

Wie in der Online-Dokumentation von Maly [6] beschrieben, verwendet das gesamte LQN Toolset die Xerces XML-Parser-Bibliothek. Den Erfahrungen dieses Praktikums entsprechend, ist es schwierig die Folgenden Fehlermeldungen ohne entsprechende Vorkenntnisse zu beheben. Im Folgenden werden einige häufige Fehlermeldungen und ihre Bedeutungen genannt (vgl. [6]):

- **Duplicate unique value declared for identity constraint of element „element“.**
  Dies bedeutet, dass es zwei Elemente mit dem gleichen Namen gibt. Dies ist nicht erlaubt, da alle Elemente einen eindeutigen Namen haben müssen. Die entsprechende Zeile des Duplikates wird angegeben. Das erste Element, das den gleichen Namen hat, befindet sich in der Regel einige Zeilen über der des Duplikates.
- **Assertion failed! Program: C\...\lqns.exe File: model.cc Line 338 Expression: newEntry != NULL**
  Dies bedeutet, dass eine Weiterleitung, bzw. ein „dest“-Parameter auf ein ungültiges Ziel zeigt. Um diesen Fehler zu beheben müssen alle Weiterleitungen betrachtet und deren Ziel überprüft werden. Falls man gerade an einer Weiterleitung für bestimmte Elemente gearbeitet hat, sollte man zuerst die entsprechenden Elemente untersuchen.
- **C:/Output/FILE.lqxo:133: error: Symbol „NAME“ not previously defined.**
  Dies bedeutet, dass ein Element mit dem Namen „NAME“ nicht definiert wurde, bevor dieses verwendet wird. Eine Stelle in der dies häufig der Fall ist, sind Precedences, die Aktivitäten verwenden, welche zuvor noch nicht definiert wurden.
- **C:/Output/FILE.lqxo:130: error: Activity „NAME“ previously used in a join.**
  Dies bedeutet, dass eine Aktivität mit Namen „NAME“ zuvor bereits in einer Precedence im „pre“-Element verwendet wurde. Eine Aktivität darf jedoch nur einmal als „pre“-Element einer Precedence verwendet werden.
- **C:/Output/FILE.lqxo:133: error: Activity „NAME“ previously used in a fork.**
  Dies bedeutet, dass eine Aktivität mit Namen „NAME“ zuvor bereits in einer Precedence im „post“-Element verwendet wurde. Eine Aktivität darf jedoch nur einmal als „post“-Element einer Precedence verwendet werden.

6 Zusammenfassung

Im Rahmen dieses Praktikums wurde eine Transformation entwickelt, die eine direkte Anbindung des BRS PCMs an das verwendete Machine Learning Framework TensorFlow ermöglicht. Der entwickelte Ansatz transformiert ein gegebenes PCM-Modell in ein LQN-Modell. Das Ergebnis dieser Transformation wird im LQN XML Schema als LQXO-Datei gespeichert und vom LQNS analysiert. Der entwickelte Ansatz erzielt für das BRS eine Genauigkeit von 99,13% (abzgl. Rundungsfehler 99,99%, siehe Unterabschnitt 4.2). Darüber hinaus kann der Ansatz auf das PCM des Simple Heuristic Example angewendet werden und erzielt eine Genauigkeit von 100% (siehe Unterabschnitt 4.1).

7 Future Work

In diesem Kapitel werden Ausblicke gegeben, wie dieses Projekt weitergeführt werden kann.

7.1 StoEx

Wie bereits in Unterunterabschnitt 3.2.1 beschrieben, wurde im Rahmen dieses Praktikums StoEx nicht in Python übersetzt, sondern lediglich eine Approximation des Parsers implementiert. In der Zukunft könnte dieses Werkzeug in Gänze übersetzt oder eingebunden werden.

7.2 Generalisierung

Im Rahmen dieses Praktikums wurden die Weichen für eine generalisierte Transformation von PCM-Modell zu LQN-Modell in Python gelegt. An einigen Stellen wurde der Code jedoch an die beiden verwendeten PCM-Beispiele angepasst:

- Das Auslesen der Reaktionszeiten in der LQNSolver-Klasse geschieht über das Abfragen des „phase1-service-time“-Parameters des „entry“-Elements des Usage-Scenario-Prozessors. Der Name dieses Prozessors kann in Zukunft automatisch Erstellt werden und die Reaktionszeit automatisiert ausgelesen werden. Die entsprechende Stelle ist im Quellcode als „TODO“ markiert.
- Sollte in einem neuen PCM eine Action vom Typ Branch verwendet werden, die noch nicht in der Branch-Action-Factory abgedeckt wurde, müsste diese Factory[10] angepasst werden.
- Momentan können nur Aktivitäten vom Typ: Start, Stop, Internal, External, Branch, Loop und EntryLevelSystemCall erstellt werden. Actions vom Typ „SetVariable-Action“ werden, wie bei Koziolek und Reussner [5], ebenfalls nicht unterstützt. Sollten außer den hier genannten Action-Typen weitere Typen auftauchen, muss die entsprechende Factory[11] erweitert werden.

[10] siehe: pyopteryx/pyopteryx/factories/action_factories/branch_action_factory.py
[11] siehe: pyopteryx/pyopteryx/factories/

Literatur

[1] Steffen Becker, Heiko Koziolek und Ralf Reussner. "The Palladio Component Model for Model-driven Performance Prediction". In: *Journal of Systems and Software* 82 (2009), S. 3–22. DOI: 10.1016/j.jss.2008.03.066. URL: http://dx.doi.org/10.1016/j.jss.2008.03.066.

[2] Greg Franks u. a. "Enhanced modeling and solution of layered queueing networks". In: *IEEE Transactions on Software Engineering* 35.2 (2009), S. 148–161.

[3] Greg Franks u. a. "Layered queueing network solver and simulator user manual". In: *Dept. of Systems and Computer Engineering, Carleton University (December 2005)* (2005), S. 15–69.

[4] Anne Koziolek. *Peopteryx*. URL: https://sdqweb.ipd.kit.edu/wiki/PerOpteryx.

[5] Heiko Koziolek und Ralf Reussner. "A model transformation from the palladio component model to layered queueing networks". In: *SPEC International Performance Evaluation Workshop*. Springer. 2008, S. 58–78.

[6] Peter Maly. *Description of the LQN XML Schema*. Englisch. März 2004. URL: http://www.sce.carleton.ca/rads/lqns/lqn-documentation/schema/.

[7] Anne Martens u. a. "Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms". In: *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*. ACM. 2010, S. 105–116.

[8] Xiuping Wu. *An approach to predicting performance for component based systems*. Carleton University, 2003.