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

Frederic Born
Matrikel-Nr. 2149235

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

1 Einführung

Zur Optimierung von Palladio-Komponentenmodellen (PCM) werden Maschinelle Lernverfahren, auf Grundlage von Neuronalen Netzen, eingesetzt. Ein existierendes PCM eines Business Reporting Systems[1] (BRS) soll für eine derartige Optimierung verwendet werden. Bei dem BRS handelt es sich um eine größere PCM Instanz, die acht Server und zwölf Komponenten umfasst. Es modelliert ein Business Reporting System, in dem Kunden Geschäftsberichte aus in einer Datenbank gespeicherten Daten generieren und herunterladen können [4]. Das Modell wurde 2010 von Martens et al. [7] als eine Fallstudie verwendet und basiert lose auf einem realistischen System.

Ziel des Praktikums ist die Ermöglichung der direkten Anbindung dieses PCM an das verwendete Framework für Maschinelles Lernen TensorFlow[2]. Das derzeitig für diese Zwecke verwendete Palladio-Plugin PerOpteryx basiert, wie auch Palladio selbst, auf der Eclipse Rich Client Plattform, die in Java implementiert ist und kann deswegen nicht ohne Weiteres von Maschinellen Lernverfahren verwendet werden, die auf TensorFlow basieren.

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

[1] `https://sdqweb.ipd.kit.edu/wiki/PerOpteryx#Business_Reporting_System`
[2] `https://www.tensorflow.org`

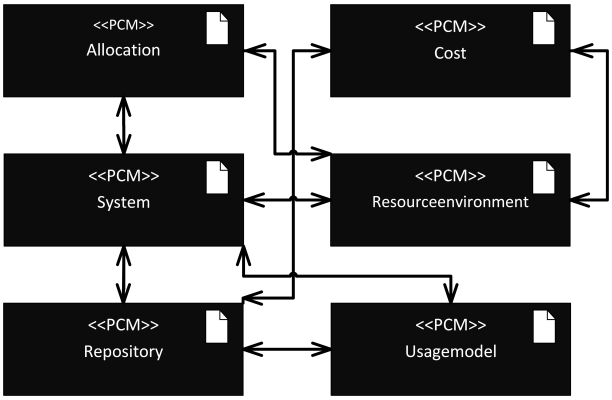

Abbildung 1: Abhängigkeiten zwischen den wichtigen PCM-Dateien.

2 Grundlagen

In diesem Kapitel werden die relevanten Grundlagen des Praktikums erläutert. Im Rahmen dieses Berichts werden nur diejenigen relevanten Aspekte der jeweiligen Grundlagen eingeführt, die im unmittelbaren Zusammenhang mit diesem Praktikum stehen. Für umfangreichere Informationen wird auf die entsprechende Literatur verwiesen.

2.1 Palladio Component Modell (PCM)

Palladio ist ein Ansatz zur Definition von Softwarearchitekturen mit besonderem Fokus auf Performance-Eigenschaften. Das PCM ist eine domänenspezifische Modellierungssprache, die entwickelt wurde um frühzeitige Leistungsprognosen für Softwarearchitekturen zu ermöglichen [1]. PCM ist ein Beispiel für ein konzeptionell klares Komponentenmodell [1]. Eine PCM-Datei ist im XML-Format gespeichert. Zum genauen Verständnis der Folgenden Sachverhalte ist es ratsam, die beschriebenen Dateien zu betrachten (siehe Git-Repository: pyopteryx/pcms). Jedes Element innerhalb einer Datei besitzt eine eindeutige ID, anhand derer eine Navigation zwischen einzelnen Dateien möglich ist. Die für dieses Praktikum essentiellen Komponenten und deren Verbindungen untereinander werden in Abbildung 1 dargestellt. Die Bedeutung der PCM-Dateien und deren wichtigste Verbindungen, zu sehen in Abbildung 1, werden mit Hilfe von Becker et al. [1] erläutert:

- ***.allocation** wird verwendet um den Allokations-Kontext einer Komponente anhand des Inputs zu erhalten. Ein Spaltenname des Inputs zeigt auf den Namen eines Allokations-Kontexts innerhalb der Allokations-Datei. Dieser Allokations-Kontext zeigt auf die ID eines Assembly-Kontexts (aus System-Datei).
- ***.system** beinhaltet Assembly-Kontexte, die auf die Id einer Komponente des Repositories verweist. Folgt man diesem Link und findet die passende Komponente aus dem Repository, wurde der linke Baum aus Abbildung 1 von oben nach unten durchlaufen.
- ***.repository** enthält Komponenten, Schnittstellen (Interfaces) und Datentypen. Das Repository wird verwendet um alle Komponenten, deren Interfaces, sowie dazu-

gehörige Aktionen (Actions) und Operationen zur Prozessorerstellung zu erhalten (siehe Abschnitt 3). Kind-Elemente einer Komponente zeigen auf ein Element in der Usagemodel-Datei (siehe Verbindung von Repository zu Usagemodel in Abbildung 1).

- ***.usagemodel** spezifiziert wie ein System benutzt wird. Informationen bezüglich der Wahrscheinlichkeit des Eintretens bestimmter Aktionen einer Repository-Komponente werden aus dieser Datei ausgelesen.
- ***.resourceenvironment** spezifiziert alle CPU-Server, sowie Ressourcenverknüpfungen des PCMs.
- ***.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 bezüglich PCM sind in der Arbeit von Becker et al. [1] oder unter auf der entsprechenden Wiki-Seite[3] nachzulesen.

2.2 PerOpteryx

PerOpteryx ist eine Palladio-Erweiterung zur Optimierung komponentenbasierter Softwarearchitekturen auf Basis modellbasierter Qualitätsvorhersagetechniken [4]. Derzeit wird PerOpteryx als eine Reihe von Eclipse-Plugins realisiert, mit dem Ziel PCM-Instanzen automatisch hinsichtlich Leistung, Zuverlässigkeit und Kosten zu verbessern [4]. Für die Leistungsanalyse unterstützt das PCM eine Transformation in Diskrete-Ereignis-Simulationen, z.B. durch Layered Queying Networks (LQN) (siehe Franks et al. [3]), um Reaktionszeiten, Durchsätze und Ressourcenauslastung zu berechnen [7]. Hierfür verwendet PerOpteryx PCM2LQN [5] und den LQN-Solver [2].

2.3 PCM2LQN Transformation

Wie in Unterabschnitt 2.2 beschrieben, wird PerOpteryx als Reihe von Eclipse-Plugins realisiert. Eines dieser Plugins, das auch für dieses Praktikum von besonderer Wichtigkeit ist, ist PCM2LQN [5]. PCM2LQN transformiert eine PCM-Instanz in eine LQN-Instanz und ermöglicht somit eine Leistungsuntersuchung des PCMs. Vereinfacht gesagt, wandelt PCM2LQN die Komponenten des PCM in eine LQN-Instanz um, die besser weiterzuverarbeiten ist. Diese LQN-Instanz wird als XML Datei, im sogenannten LQN XML Schema gespeichert und vom LQN-Solver weiterverarbeitet.

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

2.4 Layered Queueing Network XML Schema (LQN XML Schema)

Eine LQN-Instanz kann in einem speziellen XML Format gespeichert werden: LQXO. Diesem Format liegt das LQN XML Schema zu Grunde, welches auf der Arbeit von Wu [8] basiert und für die allgemeine Nutzung weiter verfeinert wurde [6]. Aufgrund der Baumstruktur einer XML Datei, werden LQN in einer Bottom-Up Reihenfolge angegeben, was eine Umkehrung der typischen Darstellung von LQN darstellt [6]. Der vereinfachte Aufbau eines LQN-Modells im LQN XML Schema ist in der, aus der Online-Dokumentation von Maly [6] stammenden, Abbildung 2 dargestellt. Die wichtigsten Elemente aus Abbildung 2 werden nun mit Hilfe des Papers von Franks et al. [3] erklärt und einige in diesem Praktikum verwendeten Parameter angegeben:

- **solver-params** werden verwendet, um verschiedene Betriebsparameter für den LQN-Solver einzustellen. In diesem Praktikum werden folgende Standard-Parameter (bis auf „comment“) aus PerOpteryx übernommen:
  - conv_val: „0.001“
  - print_int: „10“
  - underrelax_coeff: „0.5“
  - comment: „Generated by PyCM2LQN on DATUM“ (hat keinen Einfluss)
- **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** bezeichnen Aktivitäten, die zur Erfüllung einer „task“ durchgeführt werden müssen. Aktivitäten entstehen aus Actions der jeweiligen Service-Effect-Specification (SEFF) einer Komponente des Repositories. Actions haben unterschiedliche Typen.

```
<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].

In diesem Praktikum spielen: Start-, Stop-, Internal-, External-, Branch-, EntryLevelSystemCall- und Loop-Actions eine Rolle. 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.

2.5 Layered Queueing Network Solver (LQNS)

Der LQN-Solver bietet eine heuristische Leistungsanalyse und kann verwendet werden, um LQN-Modelle analytisch zu lösen [7]. Es handelt sich um ein Kommandozeilen-Werkzeug, dass seine Eingabe aus dem Dateinamen, der in der Kommandozeile angegeben ist, oder aus der Standardeingabe liest [3].

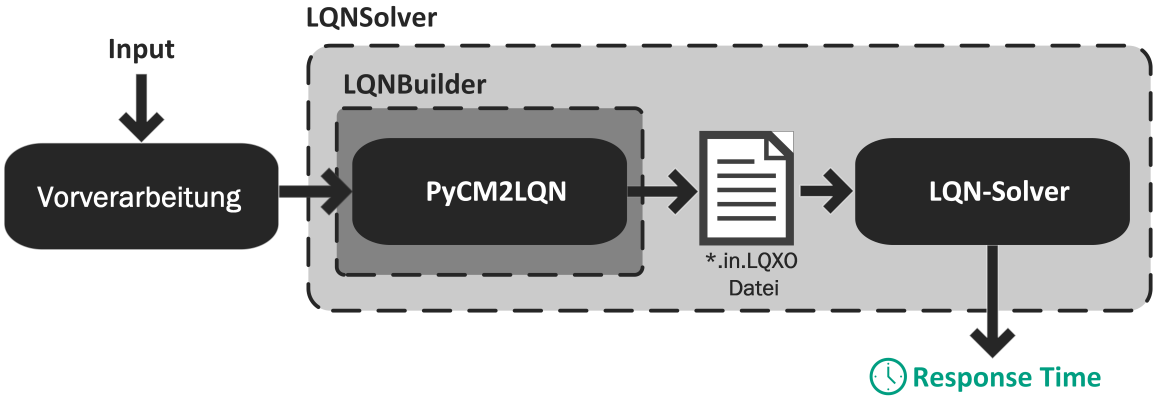

Abbildung 3: Ablauf des eigenen Ansatz: PyOpteryx.

3 Eigener Ansatz

Aufgrund der Komplexität und des Umfangs des BRS wurde entschieden, dass die Anbindung an TensorFlow mittels einer automatisierten Transformation von PCM in LQN und einer anschließenden Auswertung durch den LQN-Solver geschehen sollte. Die Herausforderung dieses Ansatzes besteht darin, dass PCM2LQN als Eclipse-Plugin realisiert ist und dadurch der Aufruf mit den erforderlichen Betriebsparametern (siehe „solverparams“ unter Unterabschnitt 2.4) aus Python heraus nicht ohne Weiteres möglich ist. Der in diesem Praktikum entwickelte Ansatz besteht darin, die für das PCM des BRS benötigten Funktionalitäten von PCM2LQN in Python zu implementieren. Um die neue Transformation 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 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.

[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

3.1 Vorverarbeitung

Da die Vorverarbeitung überwiegend von Ewald Rode bearbeitet wurde, werden in diesem Bericht nur einige essentiellen Schritte erläutert. Zum einen wird ein Cache-Objekt erstellt, in dem alle benötigten PCM Dateien gespeichert sind. Dieses Objekt wird der LQNSolver-Klasse und somit der LQNBuilder-Klasse übergeben. In der Vorverarbeitung wird außerdem ein Dictionary erstellt, das sämtliche Informationen enthält mit deren Hilfe die PyCM2LQN-Transformation durchgeführt werden kann. Neben der Allokation von verwendeten Komponenten zu einem CPU-Prozessor und deren spezifische CPU-Rate werden auch Informationen über Komponenten gespeichert, die nicht verwendet und somit aus PyCM2LQN ausgeschlossen werden.

3.2 PyCM2LQN Transformation

Die PyCM2LQN Transformation ist in der Klasse „LQNBuilder“ implementiert und besteht aus mehreren Erstellungsschritten und einem Finalisierungsschritt, die in den folgenden Unterabschnitten beschrieben werden. Eine ausführliche Dokumentation des Quellcodes ist der HTML-Dokumentation zu entnehmen, die entsprechenden PCM-Dateien sind im Repository[7] zu finden.

3.2.1 Erstellung der CPU-Prozessoren

Für jeden CPU-Prozessor wird ein Element vom Typ „processor“ erstellt, dessen Name aus dem „entityName“ des entsprechenden „resourceContainer_ResourceEnvironment“-Elements aus der „*.resourceenvironment“-Datei und dem String „_CPU_Processor“ zusammengesetzt wird. Wie in Abbildung 2 zu sehen, besitzt jeder CPU-Prozessor einen „task“ und eine „entry“.

3.2.2 Erstellung der Linking-Resource-Prozessoren

Für jede Verknüpfungsressource, die in „*.resourceenvironment“ als „linkingResources __ResourceEnvironment“-Element definiert ist, wird ein eigener Prozessor erstellt. Dieser beinhaltet einen Durchsatz- und zwei Latenz-Einträge (Synchron und Asynchron). Die Host-Demand-Mean des Durchsatz-Eintrags ist umgekehrt proportional zur Spezifikation, die im „throughput_CommunicationLinkResourceSpecification“ der „*.resource environment“-Datei angegeben wird. Die Host-Demand-Mean der beiden Latenz-Einträge ist der Spezifikation des „latency_CommunicationLinkResourceSpecification“-Elements zu entnehmen.

3.2.3 Erstellung der Usage-Delay-Prozessoren

Der Usage-Delay-Prozessor ist in den beiden LQN-Modellen, bzw. deren LQXO-Dateien, die wir betrachtet haben identisch. Er besteht aus einem Usage-Delay-Eintrag mit einer Host-Demand-Mean von „0.0“.

[7] `https://git.scc.kit.edu/udial/pyopteryx/tree/master/pcms`

3.2.4 Erstellung der Komponenten-Interface-Prozessoren

Für jede verwendete Komponente aus dem Repository werden ein oder mehrere Komponenten-Interface-Prozessoren erstellt. Eine Komponente, die im Repository definiert wird, besitzt ein oder mehrere Kinder vom Typ „serviceEffectSpecifications__BasicComponent“. Für jedes dieser Kinder wird ein eigener Komponenten-Interface-Prozessor erstellt. Ein solches Kind-Element besitzt einen Parameter „describedService__SEFF“. Der Wert dieses Parameters zeigt auf die ID eines Elements vom Typ „signatures__OperationInterface“. Dieses Element repräsentiert eine Operation eines bestimmten Interfaces. Anhand der „entityName“ von Komponente, Interface und Operation wird der Komponenten-Interface-Prozessorname zusammengesetzt (z.B. „Database_IDB_getSmallReport_Processor“).

Die Aktivitäten eines Komponenten-Interface-Prozessors werden aus den Kind-Elementen vom Typ „steps_Behaviour“ des „serviceEffectSpecifications__BasicComponent“-Elements erzeugt. Hierfür wird eine Action-Factory verwendet, die die entsprechende Aktivität und deren Precedences je nach Action-Typ erstellt (siehe HTML-Dokumentation für „pyopteryx/pyopteryx/factories“ für noch detailliertere Informationen).

3.2.5 Erstellung der Usage-Scenario-Prozessoren

Für jedes „usageScenario_UsageModel“-Element aus der „*.usagemodel“-Datei wird ein Usage-Scenario-Prozessor erstellt. Dieser Prozessor enthält alle Aktivitäten, die im „usageScenario_-UsageModel“-Element als „actions_ScenarioBehaviour“-Element definiert sind. Je nach Typ wird die Aktivität unterschiedlich erstellt. Die genaue Implementierung der unterschiedlichen Erstellungen ist in der „pyopteryx/usage_action_factories“-Dokumentation zu entnehmen.

3.2.6 Erstellung der Loop-Prozessoren

Für jedes „actions_ScenarioBehaviour“-Element vom Typ „usagemodel:Loop“ aus der „*.usagemodel“-Datei wird ein Loop-Prozessor erstellt. Diese Prozessoren enthalten wiederum alle Kind-Elemente vom Typ „actions_ScenarioBehaviour“ der entsprechenden Loop-Action als Aktivitäten.

3.2.7 Finalisierung

Im Finalisierungschritt werden nicht verwendete Prozessoren gelöscht. Ein Prozessor wird als nicht verwendet erachtet, wenn dessen Eintrag nie als „dest“-Parameter genutzt wird. Anschließend werden die Precedences auf Fehler untersucht. Im Besonderen wird überprüft, ob Precedences die eine LAN-Aktivität beinhalten, die einzigen Precedences sind, die diese Aktivität benutzten. Sobald eine Aktivität durch eine LAN-Verbindung genutzt wird, darf diese Aktivität nicht alleine (d.h. ohne „LAN_“-String) in einer anderen Precedence vorkommen. Als letzter Schritt werden die Internal-Actions eines Komponenten-Interface-Prozessors zu dem CPU-Prozessor hinzugefügt, der laut Input, der allokierte CPU-Prozessor der entsprechenden Komponente ist. Die Host-Demand-Mean für eine solche Action wird aus der Spezifikation des entsprechenden „specification_Parameteric

ResourceDemand"-Elements einer Internal-Action berechnet. Diese Spezifikationen können unter Umständen einen Stochastischen Ausdruck (engl. Stochastic Expression, kurz StoEx) beinhalten. Dabei handelt es sich um eine Spezifikationssprache, mit der u.a. Ressourcenanforderungen spezifiziert werden können. StoEx besteht aus einer Grammatik und einem Parser, die als Eclipse-Plugin implementiert ist. Die Übersetzung des Parsers in Python überschreitet den Rahmen des Praktikums und wurde für das BRS approximiert implementiert. Eine genauere Beschreibung wie dieses Problem umgangen wurde, ist im Bericht von Ewald Rode nachzulesen.

3.3 LQN-Solver

Der LQN-Solver ist als eigene Klasse realisiert, der den vorverarbeiteten Input entgegen nimmt und an den LQN-Builder weitergibt. 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.

4 Evaluation

Wie in Abschnitt 3 erwähnt, wurde der in diesem Praktikum entwickelte Ansatz 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 betreuenden Mitarbeiter als unzulässig erachtet. Die in Abschnitt 3 angesprochene Problematik mit StoEx führt in manchen wenigen Fällen mutmaßlich zu rundungsbedingten Fehlern. Diese Fehler werden in Tabelle 1 und Tabelle 2 zur Vollständigkeit aufgeführt, wobei die rundungsbedingten Fehler eine Teilmenge der fehlerhaften Testbeispiele sind. Die Größe der rundungsbedingten Fehler beträgt $\Delta Fehler : 0 < \Delta Fehler \leq 0.001$.

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% Fällen wird die korrekte Reaktionszeit berechnet.

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. Von den 108 fehlerhaften Ergebnissen

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

Tabelle 1: Testergebnisse für Simple Heuristic Example.

	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.

sind 102 nach der oben genannten Definition durch das StoEx-Problem potentiell rundungsbedingt. Werden diese minimalen Abweichungen als Fehler gewertet, ergibt sich eine Genauigkeit von 99,1%. Andernfalls werden sogar in 99,99% der Fälle die korrekte Reaktionszeit berechnet.

Der maximale Fehler zwischen Reaktionszeiten des entwickelten Ansatzes (PyOpteryx) und dem Golstandard (PerOpteryx) für das BRS beträgt $\Delta_{max} = 0.03$.

5 Ratschläge zur Fehlerbehebung und -vermeidung

In diesem Kapitel wird beschrieben, 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 ein Ansatz entwickelt, der eine direkte Anbindung des BRS PCMs an das verwendete Machine Learning Framework TensorFlow ermöglicht. Der entwickelte Ansatz besteht aus drei Komponenten: der Vorverarbeitung, PyCM2LQN-Transformation und dem LQN-Solver. PyCM2LQN 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 3.2.7 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[8] angepasst werden.
- Momentan können nur Aktivitäten vom Typ: Start, Stop, Internal, External, Branch, Loop und EntryLevelSystemCall erstellt werden. Actions vom Typ „SetVariableAction“ werden, wie bei Koziolek und Reussner [5], ebenfalls nicht unterstützt. Sollten außer den hier genannten Action-Typen weitere Typen auftauchen, muss die entsprechende Factory[9] erweitert werden.

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.

[8] siehe: pyopteryx/pyopteryx/factories/action_factories/branch_action_factory.py
[9] siehe: pyopteryx/pyopteryx/factories/