Ein generisches und hoch skalierbares Framework zur Automatisierung und Ausführung wissenschaftlicher Datenverarbeitungs- und Simulationsworkflows

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

vorgelegte

DISSERTATION

von

M. Sc. Jianlei Liu
geb. in Xi'an, China

Tag der mündlichen Prüfung:	03.07.2023
Hauptreferent:	Prof. Dr. Veit Hagenmeyer
Korreferentin:	Prof. Dr.-Ing. Anne Koziolek

Jianlei Liu
Ein generisches und hoch skalierbares Framework zur Automatisierung und Ausführung wissenschaftlicher Datenverarbeitungs- und Simulationsworkflows
Doktorarbeit
Revision: 28.02.2023
Datum der Prüfung: 03.07.2023
Gutachter: Prof. Dr. Veit Hagenmeyer und Prof. Dr.-Ing. Anne Koziolek

Karlsruher Institut für Technologie (KIT)
KIT-Fakultät für Informatik
Institut für Automation und angewandte Informatik (IAI)
Hermann-von-Helmholtz-Platz 1
76344 Eggenstein-Leopoldshafen

Abstract

Scientists and engineers designing and implementing complex system solutions use computational workflows for simulations, analysis, and evaluations. Along with growing system complexity, the complexity of these workflows also increases. However, without integration tools, scientists and engineers are more concerned with implementing additional interfaces to integrate software tools and model sets, which hinders their original research or engineering aims. Therefore, efficient automation and parallel computation of complex workflows are increasingly important in order to perform computational science in many scientific fields like energy and environmental informatics. When coupling heterogeneous models and other executables, a wide variety of software infrastructure requirements must be considered to ensure the compatibility of workflow components. The consistent utilization of advanced computing capabilities and the implementation of sustainable software development concepts that guarantee maximum efficiency and reusability are further issues that scientists within research organizations must regularly meet. This thesis addresses these challenges by presenting a new generic, modular, and highly scalable process operation framework for efficient coupling and automated execution of computational scientific workflows.

Based on a microservice architecture utilizing container virtualization and orchestration, the framework supports the flexible and efficient parallelization of computational tasks on distributed cluster nodes. Using distributed message-oriented middleware and different I/O adapters provides a scalable and high-performance communication infrastructure for data exchange between executables, allowing the computation of workflows without requiring the adjustment of executables or the implementation of interfaces or adapters. The convenient user interface based on Apache NiFi technology ensures the simplified specification, processing, controlling, and evaluation of computational scientific workflows. Due to the framework's high scalability and extended flexibility, use cases benefitting from parallel execution are parallelized, thereby significantly saving runtime and improving operational efficiency, especially during complex tasks like iterative grid optimization.

Danksagung

> *Somewhere, something incredible is waiting to be known.*
>
> **— Sharon Begley**

Die Arbeiten in der vorliegenden Dissertation wurden in den Arbeitsgruppen IT4ES (IT-Methoden und -Komponenten für Energiesysteme) und WebIS (Web-basierte Informationssysteme) am Institut für Automation und angewandte Informatik des Karlsruher Instituts für Technologie durchgeführt. Das Schreiben dieser Dissertation war eine spannende Reise und ich bin unglaublich dankbar für alle, die mich auf diesem Weg begleitet und unterstützt haben.

An erster Stelle möchte ich mich bei meinem Betreuer Prof. Dr. Veit Hagenmeyer, dem Direktor des Instituts für Automatisierung und angewandte Informatik, für die Möglichkeit bedanken, meine Forschung am Institut durchführen zu dürfen. Vielen Dank für Ihre Anleitung, Geduld, Inspiration, Unterstützung und wertvolle Beratung während der gesamten Forschungszeit. Ebenso danke ich meiner Zweitgutachterin Prof. Dr.-Ing. Anne Koziolek und meinem Prüfer Prof. Dr. Wolfgang Karl für ihre maßgebliche Unterstützung und ihr Feedback bei diversen Promotionsgesprächen.

Besonderer Dank gilt meinen Gruppenleitern Clemens Düpmeier (IT4ES) und Thorsten Schlachter (WebIS) für die Eröffnung der Forschungsmöglichkeiten innerhalb der Gruppen und für die Unterstützung und konstruktive Beratung während meiner Forschungsreise.

Ebenso möchte ich Herrn Eric Braun für die unschätzbare Unterstützung, Anleitung und Zusammenarbeit während des gesamten Forschungszeitraums herzlich danken. Die direkte Zusammenarbeit mit ihm hat mir sehr geholfen, mein Wissen im Forschungsbereich durch den kontinuierlichen Gedankenaustausch zu vertiefen.

Neben Herrn Düpmeier und Herrn Schlachter möchte ich mich noch bei Malte Chlosta, Jannik Sidler, Claudia Greceanu, Richard Lutz, Jakob Geiges und Guo Hong für ihre Hilfe bei der Überarbeitung meiner Dissertation herzlich bedanken.

Außerdem möchte ich allen meinen Co-Autoren meiner Veröffentlichungen danken (alphabetische Reihenfolge): Riccardo Remo Appino, Eric Braun, Hüseyin Çakmak,

Malte Chlosta, Clemens Düpmeier, Prof. Dr.-Ing. Timm Faulwasser, Kevin Förderer, Uwe Kühnapfel, Hatem Elias Khalloof, Richard Lutz, apl. Prof. Dr. Ralf Mikut, Tillmann Mühlpfordt, Rafael Poppenborg, Jannik Sidler, Simon Waczowicz aus dem IAI und Patrick Kuckertz, D. Severin Ryberg, Martin Robinius, Prof. Dr. Detlef Stolten vom IEK-3 (Institut für Energie- und Klimaforschung des Forschungszentrums Jülich). Ohne ihre Mitarbeit hätte ich diese Veröffentlichungen nicht publizieren können. Zusammenarbeit macht glücklich. Zusammenarbeit macht stark.

Vielen Dank auch an meine freundlichen Kolleginnen und Kollegen am IAI für die wunderbare Arbeitsatmosphäre und Unterstützung in verschiedenen Formen, die die ganze Forschungserfahrung lohnenswert gemacht haben. Ganz besonders (alphabetische Reihenfolge): Claudia Greceanu, Christina Griess, Andreas Hofmann, Bernadette Lehmann, Dominique Sauer, Nicolas Schaber, Christian Schmitt, Uwe Stucky, Wolfgang Süß. Sie tragen zu einer Kultur am IAI bei, die es zu einem angenehmen Arbeitsplatz macht.

Wegen der Unterstützung meiner Frau gebührt ihr hier mein voller und besonders herauszustellender Dank. Tief verbunden und dankbar bin ich meiner Frau Guo Hong, für ihre unglaublich hilfreiche Unterstützung und ihr Verständnis bei der Anfertigung dieser Dissertation.

Abschließend danke ich meinem Sohn Mufeng Liu, meiner Tochter Sofia Muxin Liu, den Eltern meiner Frau und meinen Eltern sowie meinen Freunden von ganzem Herzen für die Liebe, Ermutigung, Führung und unerschütterliche Unterstützung in den Jahren, die mich auf meiner Reise zu diesem Erfolg geführt haben. Sie haben immer an mich geglaubt, meine Erfolge gefeiert und mich aufgemuntert, wenn es mal schwierig wurde.

Karlsruhe, Februar 2023 *Jianlei Liu*

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation . . . 1

1.2 Wissenschaftliche offene Problembeschreibung . . . 6

1.3 Beiträge . . . 10

1.4 Struktur der Arbeit . . . 12

2 Technologische Grundlagen **15**

2.1 Co-Simulation . . . 15

2.2 Stand der Technik . . . 25

2.2.1 Microservices-basierte Architektur . . . 26

2.2.2 Containervirtualisierung Docker . . . 26

2.2.3 Container-Orchestrierung Kubernetes . . . 28

2.2.4 Kommunikationsinfrastruktur . . . 29

2.2.5 Dateningestion/-Transformation Apache NiFi . . . 31

2.2.6 REpresentational State Transfer (REST) . . . 32

3 Anforderungsanalyse und Stand der Forschung **35**

3.1 Anforderungsanalyse . . . 35

3.1.1 Beschreibung der Anforderungen . . . 35

3.1.2 Anforderungskatalog . . . 37

3.2 Stand der Forschung . . . 38

3.2.1 Kim et al.: CoSimulating Communication Networks and Electrical System for Performance Evaluation in Smart Grid . . . 38

3.2.2 Huang et al.: Open-source framework for power system transmission and distribution dynamics co-simulation . . . 40

3.2.3 Palmintier et al.: Design of the HELICS high-performance transmission-distribution-communication-market co-simulation framework . . . 42

3.2.4 Köster et al.: Snakemake-a scalable bioinformatics workflow engine . . . 44

3.2.5 Liberatore et al.: Smart grid communication and co-simulation 46

3.2.6 Schütte et al.: Mosaik: A framework for modular simulation of active components in Smart Grids 48
3.2.7 Dahmann et al.: High Level Architecture (HLA) 52
3.2.8 Analyse unberücksichtigter Forschungsarbeiten 55
3.3 Zusammenfassung und resultierender Handlungsbedarf 57

4 Framework-Architektur 61
4.1 Begrifflichkeiten 62
4.2 Multiplayer-Simulations-Architektur 63
4.2.1 Übersicht 65
4.2.2 Web-basierte Benutzeroberfläche 66
4.2.3 Beispielhafter Simulationsablauf 68
4.2.4 Rolle der Adapter 70
4.3 PROOF-Konzept 71
4.3.1 Übersicht 71
4.3.2 Überführung des PROOF-Konzepts in eine Microservices-basierte Architektur 74
4.4 PROOF-Prozess 76
4.4.1 Konzept 76
4.4.2 Wrapper 77
4.4.3 Aufbau eines PROOF-Prozesses 80
4.5 PROOF-Weboberfläche 83
4.5.1 Apache NiFi-Prozessor 83
4.5.2 Weboberfläche 87
4.6 Zusammenfassung 92

5 Konfigurierbare Kommunikation zwischen Framework-Prozessen 95
5.1 PROOF-Kommunikationsinfrastruktur 96
5.2 Datenfluss zwischen Framework-Prozessen 97
5.2.1 Übersicht 98
5.2.2 Parameter definieren 99
5.2.3 Parameterwert übertragen 100
5.2.4 Kopplung von Prozessoren 101
5.3 Volume – ein gemeinsamer Speicherbereich 102
5.3.1 Austausch großer Datenmengen mit Volumes 103
5.3.2 Bereitstellung von Applikationen 106
5.4 Konfigurierbare Services 108
5.4.1 MQTT-Service 108
5.4.2 Timeseries-Service 109

5.4.3 Dateioperation 110
5.4.4 Weitere Services 112
5.5 Zusammenfassung 112

6 Parallelisierung und Koordination von Workflows 115
6.1 Konzept der Parallelisierung und Koordination 116
6.1.1 Übersicht 116
6.1.2 Implementierung 118
6.2 Merge-Service zur Synchronisierung mehrerer Datenflüsse 120
6.2.1 Übersicht 120
6.2.2 Strategien 122
6.2.3 Deskriptive Sprache zur Definition der Strategien 125
6.3 Primary-Secondary-Architektur 130
6.3.1 Übersicht 132
6.3.2 Koordination 134
6.4 Zusammenfassung 134

7 Evaluierung 137
7.1 Methodik der Fallstudie 138
7.1.1 Fallstudienprozess 138
7.1.2 Gegenstand der Fallstudie 140
7.2 Anwendungsfälle 141
7.2.1 Welder et al.: Spatio-temporal optimization of a future energy system for power-to-hydrogen applications in Germany 141
7.2.2 González-Ordiano et al.: Probabilistic forecasts of the distribution grid state using data-driven forecasts and probabilistic power flow 145
7.2.3 Poppenborg et al.: Energy Hub Gas: A Multi-Domain System Modelling and Co-Simulation Approach 148
7.2.4 Weitere Anwendungsfälle 152
7.2.5 Zusammenfassung der PROOF-Prozesse in der Prozess-Bibliothek153
7.2.6 Zusammenfassung der Anwendungsfälle 155
7.3 Benchmarking 156
7.3.1 Use-Case-Workflow 156
7.3.2 Speichernutzung 157
7.3.3 Ausführungszeit 159
7.3.4 Parallelisierung 160
7.4 Analyse von Qualitätsmerkmalen 161
7.4.1 Funktionalität (Functional Suitablity) 161

7.4.2 Kompatibilität (Compatibility) 162
7.4.3 Portabilität (Portability) 163
7.4.4 Benutzbarkeit (Usability) 163
7.4.5 Verlässlichkeit (Reliability) 164
7.4.6 Wartbarkeit (Maintainability) 165
7.4.7 Effizienz (Performance Efficiency) 166
7.4.8 Sicherheit (Security) 167
7.5 Evaluation der Machbarkeit 167
7.5.1 A1: Hohe Konfigurierbarkeit 168
7.5.2 A2: Einfache Integration 168
7.5.3 A3: Performanter Datenaustausch 169
7.5.4 A4: Unterstützung zum Ausführen großer und komplexer Workflows 169
7.5.5 A5: Automatisierung 170
7.5.6 A6: Plattformunabhängige grafische Oberfläche 170
7.5.7 A7: Wiederverwendbarkeit 170
7.5.8 A8: Skalierung und Parallelisierung 171
7.5.9 A9: Generische Funktionalitäten 171
7.5.10 Zuordnungen der Anforderungen zu den Beiträgen 173
7.6 Zusammenfassung 173

8 Fazit und Ausblick **175**
8.1 Zusammenfassung 175
8.2 Beiträge dieser Arbeit 176
8.3 Ausblick 179

Literatur **181**

A Anhang **207**
A.1 Simulation einer Windkraftanlage 207
A.2 Erstellung von ausführbaren Matlab-Applikationen 208
A.3 Dockerfile 210
A.4 Implementierung eines NiFi-Prozessors 212
A.4.1 Initialisieren 212
A.4.2 Triggern 212
A.4.3 Löschen 214
A.5 Beispielhafter Workflow 216
A.6 XML-Vorlage 217
A.7 Parameter definieren 219
A.8 Konfigurierbare Services 219

A.8.1 MQTT-Service 219
A.8.2 Dateioperationen 221
A.8.3 Weitere Services 222
A.9 Anwendungsfälle 225
A.9.1 Welder et al.: Spatio-temporal optimization of a future energy system for power-to-hydrogen applications in Germany 225
A.9.2 González-Ordiano et al.: Probabilistic forecasts of the distribution grid state using data-driven forecasts and probabilistic power flow 227
A.9.3 Poppenborg et al.: Energy Hub Gas: A Multi-Domain System Modelling and Co-Simulation Approach 229

Publikationen **231**

Abkürzungsverzeichnis

adevs	**A** **D**iscrete **EV**ent **S**ystem
API	**A**pplication **P**rogramming **I**nterface
BHKW	**B**lock**h**eiz**k**raft**w**erk
CIM	**C**ommon **I**nformation **M**odel
CPU	**C**entral **P**rocessing **U**nit
CRUD	**C**reate, **R**ead, **U**pdate and **D**elete
CSV	**C**omma-**S**eparated **V**alues
DB	**D**aten**b**ank
E/A	**E**in-/**A**usgabe
eASiMOV	**E**lectrical Grid **A**nalysis **Si**mulation **M**odeling **O**ptimization and **V**isualization
EH	**E**nergy **H**ub
EL2.0	**E**nergy **L**ab 2.0
EMS	**E**nergy **M**anagement **S**ystem
ES2050	**E**nergy **S**ystem 2050
ESM	**E**nergy **S**ystem **M**odeling
ESO	**E**nergy **S**ystem **O**ptimization
FACTS	**F**lexible-**AC**-**T**ransmission-**S**ystems
FIFO	**F**irst **I**n **F**irst **O**ut
FMI	**F**unctional **M**ock-up **I**nterface
FMU	**F**unctional **M**ock-up **U**nit
FNCS	**F**ramework for **N**etwork **C**o-**S**imulation
FOM	**F**ederation **O**bject **M**odel

GAMS	General Algebraic Modeling System
GUI	Graphical User Interface
HELICS	the Hierarchical Engine for Large-scale Infrastructure Co-Simulation
HiL	Hardware-in-the-Loop
HLA	High Level Architecture
HPC	High Performance Computing
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IEEE	Institute of Electrical and Electronics Engineers
IKT	Informations- und Kommunikations-Technologie
IP	Internet Protocol
JSON	JavaScript Object Notation
MQTT	Message Queuing Telemetry Transport
nar	NiFi Archive
ns-2	Network Simulator version 2
ns-3	Network Simulator version 3
OMT	Object Model Template
PDF	Portable Document Format
PPF	Probabilistic Power Flow
PROOF	PROcess Operation Framework
PSLF	Positive Sequence Load Flow
QR	Quantile Regression
RAM	Random-Access Memory
Redis	Remote Dictionary Server
RES	Renewable Energy Source

REST REpresentational State Transfer

RTI Runtime Infrastructure

SOM Simulation Object Model

SSH Secure SHell

SSL Secure Sockets Layer

TCP Transmission Control Protocol

THYME Toolkit for HYbrid Modeling of Electric power systems

UUID Universally Unique Identifier

VM Virtuelle Maschine

WIMAX Worldwide Interoperability for Microwave Access

WPAN Wireless Personal Area Network

XML Extensible Markup Language

yaml Yet Another Markup Language

1 Einleitung

1.1 Motivation

Die Wissenschaft folgt traditionell häufig dem Weg, eine Problemstellung zu formulieren, ein wissenschaftliches Experiment als Verifizierungsumgebung zu erstellen und das Experiment auszuführen, um Daten zu sammeln. Danach werden die Daten analysiert, um realisierte Annahmen entweder zu verifizieren oder zu falsifizieren. Das Einrichten realer Umgebungen zur experimentellen Evaluation von Problemstellungen ist oft kostspielig, zeitaufwändig, oder gar unmöglich, da das reale System hierdurch verändert würde. Auch das parallele Testen von verschiedenen Szenarien ist in der realen Welt nicht umsetzbar. Eine Alternative bieten digitale Lösungen, die Softwaremodelle verwenden, um die experimentelle Umgebung zu ersetzen. Die Softwaremodelle werden in Simulatoren und anderen wissenschaftlichen Softwaretools ausgeführt, um Aufgaben zu erledigen und den realen experimentellen Workflow zum Sammeln von Daten durch einen digitalen Workflow in silico zu ersetzen. Ein wissenschaftlicher Workflow in der Forschung stellt eine formale Prozessbeschreibung zum Erreichen eines wissenschaftlichen Ziels dar. Die Beschreibung eines Workflows wird typischerweise in Form von Aufgaben und Datenabhängigkeiten ausgedrückt. Das Forschungsfeld der Computerwissenschaft begegnet solchen Herausforderungen mit der Erstellung der erforderlichen Werkzeuge und Methoden als Softwarelösungen. Die Softwarelösungen unterstützen die digitalen Workflows. Forschende aller Bereiche nutzen solche Softwarelösungen. Aspekte, die für die jeweilige Forschung relevant sind, werden formalisiert und in Computermodelle umgewandelt. Die Modelle benötigen effiziente Mittel zur Analyse großer Mengen datenintensiver Szenarien und zur Nutzung fortschrittlicher Rechenfunktionen. Disziplinübergreifend umfasst die Modelllandschaft eine Vielzahl von Modelltypen. Modelltypen fokussieren sich jeweils auf bestimmte Facetten eines Systems mit individuellen Detail- und Auflösungsebenen und wenden unterschiedliche Methoden für die Simulation, Optimierung und statistische Datenverarbeitung an. Um einen breiteren Überblick über ein System zu erhalten, müssen in der Regel mehrere Modelle, Simulatoren und andere Hilfswerkzeuge (z.B. Optimierungswerkzeuge) verwendet und deren Eingabe- und Ausgabedatenflüsse kombiniert werden, um einen komplexeren wissenschaftlichen Workflow für die Evaluation von Schlüs-

selsystemeigenschaften zu bilden. Ein solcher Workflow kann formal beschrieben werden, indem die Verbindung zwischen Modellausgabe- und Eingabeströmen und die koordinierte Ausführung der Modelllogik, die eine sequentielle, iterative oder simultane Ausführungslogik implementiert, definiert werden. Bei der Beteiligung von mehr als einem Simulator können andere komplexere Koordinationsmechanismen zur Modellierung von Co-Simulation-Workflows erforderlich sein. Präsente Beispiele hierfür sind die zeitschrittbasierte oder ereignisbasierte Synchronisation der Simulatorausführung. Zusätzliche Verarbeitungsaufgaben komplettieren die Modellaufgaben als Teil eines automatisierten Workflows, und erhöhen dadurch den Grad an Workflow-Automatisierung. Beispiele hierfür sind Datentransformation, Validierung, oder Visualisierung.

Der Ablauf softwareinstrumentierter Workflows im Kontext der Informatik stellt ein digitales Äquivalent zum herkömmlichen Versuchsaufbau dar. Eine Darstellung des Ablaufes ist in Abb. 1.1 zu sehen. In diesem Fall beginnt die wissenschaftliche Forschung mit der Formulierung einer beliebig konkreten Forschungsfrage. Die Vorteile der Erforschung dieses Themas und die Auswirkungen potenzieller Ergebnisse werden beschrieben. Nachdem die Frage gestellt wurde, wird typischerweise eine gründliche Untersuchung durchgeführt, z.B. basierend auf einer Literaturrecherche mit thematischem Hintergrund. Dabei werden die fortschrittlichsten Modelle, Verfahren und Prozesse identifiziert, die bei der Beantwortung der Frage hilfreich sein könnten. Die gefundenen Modelle, Verfahren und Prozesse müssen dann in das digitale Äquivalent des klassischen Versuchsaufbaus kompiliert werden. Dann können die digitalen Versuche durch Ausführen des digitalen Workflows durchgeführt werden. Der Forschungsansatz wird abhängig von der Verfügbarkeit der bei dieser Untersuchung gefundenen Voraussetzungen festgelegt. Die gefundenen Modelle und Methoden können direkt integriert und verwendet werden. Voraussetzung hierfür ist, dass die gefundenen Modelle und Methoden auf dem neuesten Stand der Technik bereits in einer Form implementiert sind, die zur Software-Infrastruktur des Forschenden passt. Wenn dies nicht möglich ist, werden neue Modellimplementierungen oder die Implementierungen dedizierter Software, z.B. das Implementieren eines bestimmten Steueralgorithmus, notwendig. Nachdem alle Modelle und weitere Softwaretools implementiert wurden, kann der vorgesehene Workflow für digitale Experimente (manuell) ausgeführt werden. Hierzu werden geeignete Testdaten eingegeben und Ergebnisdaten gesammelt, bis alle jeweiligen Szenarien ausreichend untersucht sind. Die endgültigen Modellergebnisse, die in der Regel bereits am Ende eines Workflows nachbearbeitet und für die menschliche Lesbarkeit visualisiert werden, werden dann in den Kontext des Forschungsthemas und der Ergebnisse früherer wissenschaftlicher Arbeiten gestellt. Die Ergebnisse werden sorgfältig interpretiert

und Schlussfolgerungen innerhalb der jeweiligen wissenschaftlichen Gemeinschaft gezogen. Idealerweise wird eine zufriedenstellende Antwort auf die Forschungsfrage gefunden. In der Regel führen Forschungsergebnisse zu neuen Forschungsfragen, die Erweiterungen und Anpassungen des vorherigen Workflows erfordern. Dies ist der Beginn der nächsten Iteration innerhalb des Workflow-Entwicklungszykluses.

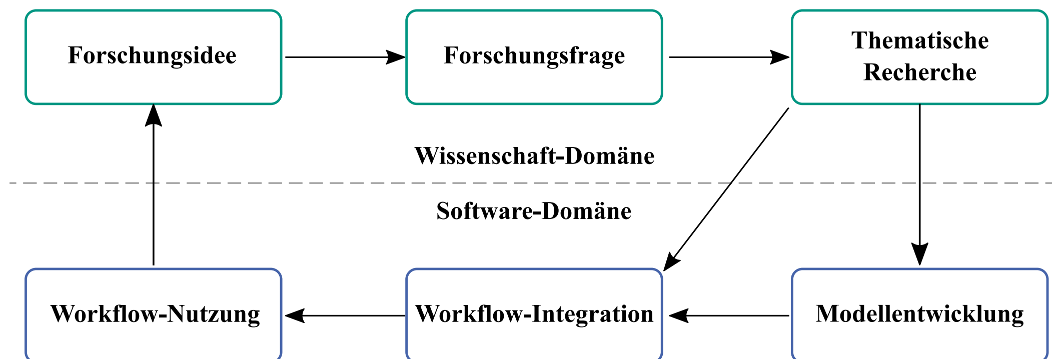

Abb. 1.1.: Entwicklungszyklus von Workflows im Kontext von Computational Science [Liu+19; Ove09]

Wissenschaftliche Workflows bestehen aus mehreren Teilkomponenten (Applikationen), die voneinander abhängen und miteinander kommunizieren müssen. Zur Implementierung der Koordination und Interaktion zwischen Applikationen können verschiedene Ansätze verwendet werden. In einem engen Kopplungsansatz verweisen konkrete Applikationsimplementierungen in ihren Quellcodes direkt aufeinander. Dieser Ansatz beschränkt sich auf die Verwendung geeigneter Programmierframeworks für die Implementierung der internen Prozess- oder verteilten Kommunikationslogik und erfordert eine genaue Kenntnis der internen Abläufe der einzelnen Modelle (White Box). Die unmittelbaren Abhängigkeiten zwischen Modellfunktionalitäten führen zu einer geringen Modularität und die Wiederverwendbarkeit von Modellcode in anderen Kontexten, sowie das interne Refactoring, werden eingeschränkt. Der Austausch einzelner Modellkomponenten oder -versionen erfordert in diesem Ansatz einen hohen Aufwand an Neuprogrammierung, Tests und Dokumentation. Die lose Kopplung [HP17] hingegen zielt auf eine Abstraktion von Modellkomponenten ab, bei der jede Komponente eine generische Schnittstelle implementiert. Beispiele hierfür sind die Definition von Eingabe- und Ausgabefunktionen, die flexible Konfiguration für die Anpassung des Moduls an verschiedene Anwendungskontexte und die Kapselung der internen Modelllogik (Black Box). Mithilfe dieses modularen Ansatzes zur Implementierung eines Workflows können Modelle in verschiedenen Anwendungskontexten verwendet werden, und es sind nur Kenntnisse über die jeweiligen Schnittstellen erforderlich. Das Ersetzen von

Komponenten innerhalb eines Workflows, der eine kompatible externe Schnittstelle implementiert, erfordert keine Anpassung des Quellcodes bei einer Änderung der Aufgaben. Nur eine Anpassung der Laufzeitkonfiguration, die für die Ausführung der Aufgabenkomponente verantwortlich ist, wird vorgenommen. Je häufiger die jeweiligen Aufgaben gemeinsame Schnittstellenstandards verwenden, umso weniger Anpassungen sind erforderlich, um diese Komponenten in einen Workflow zu integrieren. Schnittstellenstandards beschreiben beispielsweise den Austausch von Daten oder Konfigurationsinformationen. Obwohl ein loser Kopplungsansatz einen höheren Freiheitsgrad beim Kombinieren von Programmiersprachen und -umgebungen ermöglicht, muss der Entwurf einer stabilen standardisierten Schnittstellen in Kopplung mit anderen Softwaretools bei der Erstellung eines Workflows explizit berücksichtigt werden.

Im Zusammenhang mit der Erforschung und Bewertung des Verhaltens komplexer Systemkonfigurationen wird die manuelle Ausführung digitaler wissenschaftlicher Workflows immer mühsamer und umständlicher. Der Prozess eines manuellen Ausführens beinhaltet oft viele Schritte über verschiedene Softwaretools hinweg. Die Verarbeitung von Daten, die Ausführung von Simulationen und Modellen, sowie die Präsentation der ermittelten Ergebnisse sind essenzielle Teile der wissenschaftlichen Arbeit. Die tägliche Realität der Forschenden ist dabei häufig geprägt von manuellen, teils repetitiven Schritten, vom Einsatz verschiedener spezifischer Werkzeuge, Brüchen im Datenfluss und von der Nutzung von implizitem Expertenwissen. Oft ist eine automatisierte koordinierte Ausführung eines oder mehrerer Softwaretools erforderlich, um beispielsweise eine gemeinsame Simulation von lose gekoppelten eigenständigen Simulatoren (Co-Simulation [KS10; Sad+18])) durchzuführen. Forschende setzen die Koordination und benötigten Klebstoff-Code häufig eigenständig mithilfe von Programmcode um. Existierende Frameworks (z.B. ein Co-Simulations-Framework), oder vordefinierte Strategien werden hier oft als Hilfsmittel herangezogen. Dies alles trägt zu einem großen Aufwand bei, mit dem Forschende beim Aufbau ihres digitalen Äquivalents eines Versuchsaufbaus konfrontiert sind. Dieser Aufwand hindert die Forschenden daran, sich auf die Durchführung der eigentlichen Experimente und die Auswertung der Ergebnisse zu konzentrieren. Daher besteht heutzutage ein zentraler Forschungsbedarf darin, zu bestimmen, ob generische Ausführungsplattformen für wissenschaftliche Workflows erstellt werden können. Ziel der generischen Ausführungsplattformen ist es, die Ausführung komplexer wissenschaftlicher Workflows vollständig zu automatisieren und die Forschenden von der manuellen Implementierung verschiedener Arten von Werkzeugen zu befreien, um beispielsweise Koordination oder Datentransformation zu implementieren.

Angesichts der schnell wachsenden Komplexität moderner technischer Systeme und des Bedarfs, viele miteinander verbundene wissenschaftliche Aufgaben effizient auszuführen, ist das orchestrierte Ausführen von Tools und die Einrichtung einer Softwareumgebung, die benutzerfreundlichen Definition und Ausführung von Workflows ermöglicht, ein wissenschaftliches, strategisches Problem mit erheblichen Auswirkungen auf die Forschungseffizienz. Vor dem Hintergrund der von der jeweiligen wissenschaftlichen Community verwendeten Software hat die Auswahl von Programmiersprachen und -umgebungen, Betriebssystemen, Optimierern und anderen unterstützenden Tools und Bibliotheken, Schnittstellenstandards, Dateiformaten usw. direkten Einfluss auf das Potenzial zur Kopplung des Modellsatzes mit anderen Forschenden. Da die Untersuchung von komplexen und interdisziplinären Forschungsfragen in der Regel nicht von einer einzigen Organisation behandelt wird, sondern durch die Zusammenarbeit innerhalb der Community ermöglicht wird, ist dieses Potenzial von besonderer Bedeutung. Gleichzeitig ist es keine leichte Aufgabe, den Softwareentwicklungsprozess einer Organisation zu ändern. Modelle und andere ausführbare Dateien sind bereits implementiert, sodass Neuentwicklungen mit anderen Programmiersprachen und -umgebungen selten sinnvoll, zumindest aber mit erheblichem Aufwand und Einarbeitungszeiten verbunden sind. Darüber hinaus sind die Umschulung von Mitarbeitern und der Wiederaufbau von Programmiererfahrungen komplizierte Aufgaben. Dies macht Kopplungsumgebungen, die die Integration bestehender Werkzeuge und Modelle mit möglichst geringem Aufwand ermöglichen, sehr wertvoll.

Wenn eine ausreichende Flexibilität bei der Modellkopplung gewährleistet werden soll, stellt die effiziente Verarbeitung der immer komplexer werdenden Workflows, die für die Erzielung aussagekräftiger und zuverlässiger Ergebnisse unabdingbar sind, eine wachsende Performance-Herausforderung dar [Bar+22]). Im Gegensatz zu integrierten White-Box-Anwendungen, die häufig intern für paralleles Rechnen unter Verwendung eines Parallel-Computing-Frameworks entwickelt werden, werden lose gekoppelte Workflows normalerweise durch Verketten vorhandener Modellkomponenten ohne die effiziente parallele Ausführung der Komponenten erstellt. Daraus folgt, dass die Logik zum parallelen Ausführen von Unterprozessen und gesamten Workflows sowie die Kommunikationssteuerung und Konsistenzsicherungsmechanismen für jeden neu entworfenen Workflow entwickelt, implementiert und getestet werden müssen. Das Workflow-Engineering und die Laufzeitumgebung sollten es ermöglichen, Funktionen als wesentliche Teile eines wissenschaftlichen Computer-Workflows so einfach wie möglich zu realisieren.

1.2 Wissenschaftliche offene Problembeschreibung

Die Beantwortung vieler Fragestellungen durch den Aufbau realer Experimentalumgebungen mit echten Anlagen ist heutzutage oftmals zu teuer, zu ineffizient oder nicht möglich. Zur effizienten Durchführung wissenschaftlicher Aufgaben benötigen Forscher bzw. ganze Forschungsteams heutzutage eine Vielzahl von Softwarewerkzeugen als Bestandteile einer digitalen Experimentierumgebung. Selbst bei der Nutzung realer Experimentierumgebungen mit technischen Anlagen/Komponenten müssen viele wissenschaftliche Arbeitsschritte ebenfalls unter Nutzung verschiedenster Softwarewerkzeuge digital durchgeführt werden. Zum Beispiel im Business-Bereich sind schon sehr lange Softwarelösungen bekannt, welche die unterschiedlichen Arbeitsschritte eines komplexen Geschäftsprozesses zu effizienten digital unterstützten Arbeitsabläufen integrieren, an denen eine Vielzahl von Mitarbeitern beteiligt sein können. Im Gegensatz dazu findet man bei der wissenschaftlichen Zusammenarbeit typischerweise häufig isolierte Umgebungen vor, bei denen jedes Teammitglied seine Arbeitsschritte weitgehend auf eigenen Rechnern mit unterschiedlichen Werkzeugen isoliert, und bei denen oftmals bereits der Austausch von Daten über die verschiedenen Tools hinweg – geschweige denn mit Partnern in anderen Organisationseinheiten – ein größeres Problem darstellt. Dies gilt besonders für fächerübergreifenden Themengebiete wie der Erforschung von komplexen Energiesystemen. Komplexe Energiesysteme bringen Spezialisten aus vielen verschiedenen wissenschaftlichen Bereichen zusammen, hierzu gelten vor allem Elektroingenieure, Informatiker, Wirtschaftswissenschaftler, Physiker, und Weitere. Dies gilt insbesondere im Bereich der Forschung zu komplexen Energiesystemen, die die Communities von Spezialisten aus vielen verschiedenen wissenschaftlichen Gemeinschaften erfordern, z.B. Elektroingenieure, Informatiker, Wirtschaftswissenschaftler, Physiker, Chemiker, Biologen. Experten dieser Disziplinen entwickeln Modelle und Simulatoren als Komponenten von Subsystemen für verschiedene Teile des Energiesystems. Alle Komponenten werden mithilfe von **I**nformations- und **K**ommunikations-**T**echnologie (IKT) integriert, um intelligente Geräte im Netz miteinander zu verbinden, und dann automatisch als ein einziger koordinierter Simulation-Workflow für zukünftige intelligente Energielösungen (z.B. Smart Grid[1] [Sia14]) ausgeführt. Mehrere Forschungsprojekte untersuchen die Anforderungen

[1] Gemäß der Definition der International Electrotechnical Commission sind Smart Grids elektrische Energieversorgungssysteme. Solche Systeme nutzen den Informationsaustausch, Kontrollstrategien, verteilte Datenverarbeitung, sowie Aktuatoren und Sensoren zur

- Integration des Verhaltens der Netzwerknutzer, sowie anderer Teilnehmer und
- effizienten Bereitstellung einer nachhaltigen, ökonomischen und sicheren elektrischen Energieversorgung.

an den zukünftigen Smart-Grid-Betrieb und sollen mögliche Lösungen für die Herausforderungen der Energiewende zu erarbeiten. Zu diesen Projekten gehören das **E**nergy **L**ab 2.0 (EL2.0)-Projekt [Hag+16; Düp+16]) und das **E**nergy **S**ystem 2050 (ES2050)-Forschungsprojekt[2] [HEL22]) der Helmholtz-Gemeinschaft. Abb. 1.2 illustriert die Rolle des Smart Grids schemenhaft. In der Abbildung kennzeichnen alle Symbole links den Verbraucher und die Symbole rechts den Erzeuger. Die Erzeuger werden unterteilt in erneuerbare Energie (grün markiert) aus nachhaltigen Quellen wie Wasserkraft, Wind und Sonne und nicht-regenerativer Energie (rot markiert) aus Quellen wie Kohle, Öl, Gas und Kernkraft. Basierend auf digitaler Technologie dient ein Smart Grid als Elektrizitätsnetz dazu, die Stromflüsse von einigen wenigen zentralen Stromerzeugern an eine große Anzahl von Verbrauchern zu übertragen. Dabei ermöglicht das Smart-Grid-System die Überwachung, Analyse, Steuerung und Kommunikation innerhalb der Versorgungskette. Ein Smart Grid kann so die Effizienz verbessern, den Energieverbrauch sowie die Kosten senken, und die Transparenz und Zuverlässigkeit der Energieversorgungskette erhöhen [kh22]).

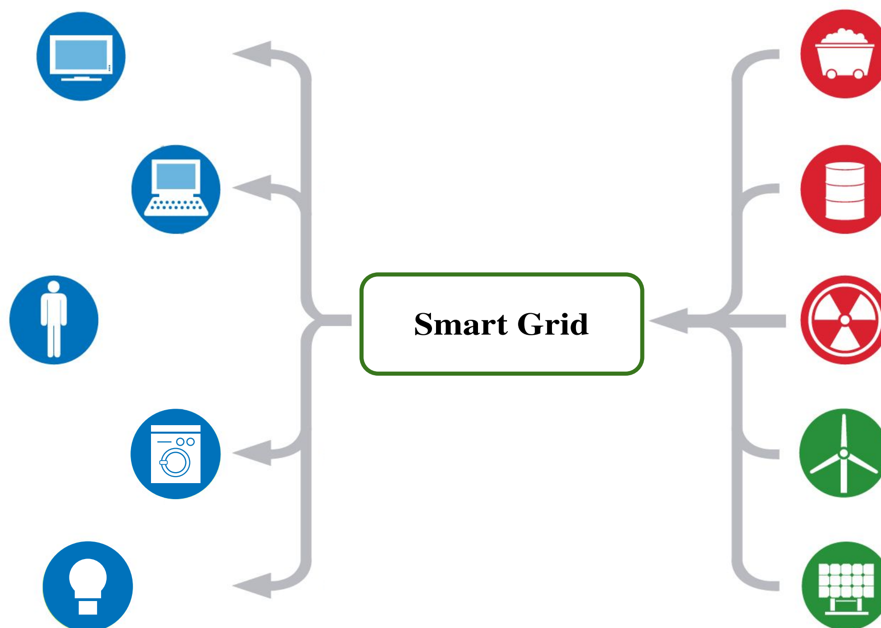

Abb. 1.2.: Smart Grid [Pau+14; Col16]

Die vorliegende Arbeit befasst sich daher mit einer Lösung des Problems, wie sich die Arbeit von wissenschaftlichen/technischen Teams über Nutzung integrierter digitaler Umgebungen effizienter gestalten lässt. Ein Schlüsselelement einer solchen

[2] https://www.helmholtz.de/forschung/forschungsbereiche/energie/energie-system-2050/

Forschungsumgebung könnte eine einheitliche, durchgängige Softwareplattform sein. Auf solch einer Softwareplattform können die beteiligten Forschenden Daten auf einfache Weise austauschen. Darauf lassen sich auch komplexe wissenschaftliche Arbeitsvorgänge, für deren Durchführung verschiedene Softwaretools genutzt werden müssen, zu digitalisierten automatisierten wissenschaftlichen Bearbeitungsworkflows zusammensetzen. Die aufgebauten Workflows können dann auf einer leistungsfähigen wissenschaftlichen Rechnerinfrastruktur (Computing Cluster) hochperformant ausgeführt werden. Die folgenden Unterabschnitte zeigen eine Übersicht und Diskussion der Probleme, die eine solche Plattform lösen könnte. Im Rest der Arbeit werden Lösungen für die Probleme in Form verschiedener Beiträge vorgestellt. Die mit P1 bis P4 bezeichneten Probleme lassen sich wie folgt zusammenfassen:

P1: Die manuelle Ausführung von Softwarewerkzeugen zur Durchführung wissenschaftlicher Bearbeitungsvorgänge ist häufig fehlerbehaftet und zeitaufwändig für den Nutzer.

Bei der systemischen Forschung zu Energiesystemen müssen z.B. durch unterschiedliche Partner eine Vielzahl von Einzelmodellen für Teilsysteme erstellt und dann zu einem Systemmodell (z.B. unter Nutzung eines Co-Simulationsansatzes) zusammengesetzt werden. Möchte man in das Systemmodell intelligente Betriebsführung integrieren, um z.B. neue Formen der Steuerung des Systems evaluieren zu können, müssen häufig weitere Werkzeuge zur Prognose, Datenanalyse, Optimierung usw. mit in die Bearbeitungskette integriert werden. Zum Aufbau solcher Modelle oder Simulationen werden verschiedene Programmiersprachen und verschiedene Werkzeuge eingesetzt, die auf unterschiedlicher Hardware installiert werden. Um ein integriertes systemisches Modell zur Simulation und anschließenden Analyse einer solchen komplexen systemischen Lösung zu implementieren, müssen die Teilkomponenten zum Datenaustausch miteinander gekoppelt und simuliert werden. Jeder Teilaspekt eines solchen Projektablaufs wird heutzutage in der Regel von den beteiligten Personen ohne Nutzung einer integrativen Umgebung unter manuellem Einsatz verschiedenster Softwarewerkzeuge auf dem Arbeitsplatzrechner eines Wissenschaftlers durchgeführt, was sehr ineffizient ist.

P2: Der Datenaustausch mit nicht-direkt-kompatiblen Schnittstellen ist fehlerbehaftet

Der Austausch von Zwischenergebnissen bei komplexen wissenschaftlichen Workflows kann sich oft als sehr kompliziert gestalten. Nicht digital unterstützte Workflows werden im Vorfeld oft nicht präzise genug definiert. So entstehen häufig

Schnittstellenfehler in Zwischenergebnissen, wie Daten, die sowohl in Bezug auf die Syntax als auch Semantik in einem nachfolgenden Schritt als Eingabe in ein weiteres Tool eingehen. Bestenfalls führt dies zu der Notwendigkeit einer manuellen Nachbearbeitung der Zwischenergebnisse, aber schlimmstenfalls dazu, dass mit dem Ersteller des Zwischenergebnisses mehrere Iterationen notwendig sind, bis die Daten in der benötigten Form vorliegen.

P3: Mangelhafte Portabilität aufgrund von fehlender plattformunabhängiger Bedien- und Ausführbarkeit

Jedes Softwarewerkzeug zur Durchführung eines wissenschaftlichen Bearbeitungsschrittes benötigt dedizierte Laufzeitumgebungen und hat wiederum Abhängigkeiten, beispielsweise zum Betriebs-Ökosystem. Rechner müssen oftmals für die Ausführung von wissenschaftlichen Bearbeitungsvorgängen dediziert für die jeweilige wissenschaftliche Person konfiguriert werden. Benötigte Daten als Eingabe für die Softwarewerkzeuge werden häufig auf den Bearbeitungsrechnern gespeichert, sind nicht mit Metadaten für eine einfache Suche versehen, und oftmals nicht plattformunabhängig von anderen Rechnern erreichbar. Das Auffinden, Übertragen und die systematische Aufbewahrung der Ein-/Ausgabedaten kostet Zeit und ist oft nicht plattformunabhängig realisiert.

P4: Mangelnde Skalierbarkeit durch fehlende Parallelisierbarkeit von Vorgängen

Die Durchführung wissenschaftlicher Arbeitsschritte auf Einzelrechnern z.B. am Arbeitsplatz reduziert in vielen Situationen die Möglichkeiten, softwaregestützte Arbeitsvorgänge oder komplexe Workflows als Abfolge einzelner Arbeitsschritte performant und skalierbar auszuführen. Die manuelle Bearbeitung wissenschaftlicher Arbeitsschritte ist nicht nur schwer zu parallelisieren, sondern auch die Erhöhung der Skalierbarkeit bedeutet zusätzlichen Aufwand.

Die vorgestellten Probleme finden in der Literatur besonders unter den Sammelbegriffen „Co-Simulation“ und „Automatisierungsframework“ Beachtung. Die Arbeiten können Teilaspekte der gewünschten Lösung adressieren. So stellen [Ama+15b; Sun+14; Bia+15; BAS14; Shu+18a] verschiedene Co-Simulation-Frameworks vor, die speziell für Smart-Grid-Anwendungsszenarien entwickelt wurden. Alle diese Frameworks sind nicht generisch oder skalierbar. Sie koppeln jeweils nur zwei Simulatoren und sind auf bestimmte Szenarien beschränkt. Die Frameworks bieten keine Möglichkeit, verschiedene Arten von Modellen oder Simulatoren zu integrieren. Somit bieten sie keine Basis zum Aufbau und zur Simulation realistischer

Multidoman-Energiesystemszenarien. In Kapitel 3 zeigt das Ergebnis einer detaillierten wissenschaftlichen Recherche und Analyse aber auch, dass diese existierenden Arbeiten bei weitem nicht alle geforderten Anforderungen aus dem wissenschaftlich/-technischen Bereich erfüllen, sondern dass weitere, auf technisch/wissenschaftliche Anwendungen zugeschnittene Lösungsbausteine notwendig sind.

1.3 Beiträge

Die vorliegende Arbeit befasst sich mit der Lösung der oben genannten offenen Probleme, die im Rahmen der Analyse des Stands der Softwarearchitekturen und Ansätze zur Co-Simulation, Datenverarbeitung und Automation identifiziert wurden. Das Ziel dieser Dissertation ist es, ein neues Framework zur Automatisierung wissenschaftlicher Workflows zu konzipieren und zu entwickeln. Hierdurch soll sich die Arbeit von wissenschaftlichen/technischen Teams effizienter gestalten lassen. Das neue Framework wird in dieser Arbeit **PRO**cess **O**peration **F**ramework (PROOF) genannt. Das PROOF-Konzept nutzt bereits vorhandene Ansätze zur Co-Simulation, Datenverarbeitung und Automation und ergänzt diese mit eigenen Lösungsbausteinen zu einer integrierten Gesamtlösung, die auf den Bereich der technisch/wissenschaftlichen Forschung und Entwicklung zugeschnitten ist.

Nach der Diskussion der im vorherigen Abschnitt identifizierten wissenschaftlichen Probleme werden in den folgenden Abschnitten die wichtigsten Beiträge zur Gesamtlösung dieser Probleme beschrieben. Die Beiträge bilden den Kern dieser Arbeit und sind jeweils mit B1-B3 gekennzeichnet.

B1: Konzeption und Umsetzung eines Grundkonzeptes für die Beschreibung und Ausführung wissenschaftlicher Workflows und Co-Simulationen

Für die Umsetzung des Konzeptes wird eine Microservice-Architektur verwendet. Diese Technologie ist aus dem Cloud-Computing-Bereich bekannt und wird zur Containervirtualisierung und Container-Orchestrierung verwendet. Unter der Verwendung modularer Framework-Bausteine können die Applikationen voneinander isoliert werden. Auf der Basis großer Computing-Umgebungen (z.B. Rechnercluster) werden die Framework-Bausteine als verteilte Laufzeitinfrastruktur automatisiert betrieben und skaliert. Hierbei gilt es die Laufzeit- und Arbeitsumgebungen der wissenschaftlichen Applikationen in den jeweiligen Bausteinen zu beachten. Die Applikation, inklusive ihrer Laufzeitumgebung (Bibliotheken, Löser), wird in ein

Containerimage verpackt. In das Containerimage kann „zur Laufzeit“ auch eine benötigte Arbeitsumgebung (bei einem Simulator z.B. das benötigte Modell und die **E**in-/**A**usgabe (E/A)-Dateien) eingehängt werden. Ebenfalls können die bereitgestellten E/A-Adapter in das Containerimage integriert werden, um die Kommunikation der eingepackten Applikation mit anderen Werkzeugen eines Workflows zu konfigurieren und zu realisieren, ohne dass die eingepackte Applikation selbst dafür Mechanismen bereitstellen muss.

Basierend auf Apache NiFi wird eine benutzerfreundliche Weboberfläche bereitgestellt und um fehlende Funktionalitäten für wissenschaftliche Workflows erweitert. Damit können wissenschaftliche Workflows grafisch erstellt, betrieben und verwaltet werden. Jede so verpackte Softwareapplikation kann dann mit zusätzlichen Metadaten in einer Softwaremodul-Bibliothek gespeichert werden. Alle integrierten Applikation sind über die Bibliothek verfügbar und können als sogenannte Framework-Bausteine auf die webbasierte Arbeitsoberfläche gezogen werden, um hiermit wissenschaftliche Workflows visuell aufzubauen. Die einzelnen Framework-Bausteine können dann miteinander verknüpft werden, wobei Parameter, Eingaben und Ausgaben dafür definiert und konfiguriert werden können, wie Daten zwischen Instanzen der Softwareapplikationen fließen sollen bzw. wie die Bearbeitungsschritte miteinander koordiniert werden.

Beitrag B1 löst das erste Problem P1 durch Verwendung der Containervirtualisierungsplattform und des -Orchestrierungswerkzeugs. Außerdem wird das dritte Problem P3 durch Integration der Weboberfläche zur plattformunabhängigen Bedien- und Ausführbarkeit abgedeckt.

B2: Konfigurierbare Kommunikation zwischen Framework-Prozessen

Unter Nutzung von einer verteilten nachrichtenorientierten Middleware und verteilten virtuellen Laufwerken, die in Laufzeitcontainer dynamisch eingehängt werden können, sowie verschiedenen E/A-Adaptern erfolgt der Austausch von Daten zwischen den einzelnen laufenden Softwaresystemen, ohne dass deren interne Logik hierfür an das Framework angepasst werden muss. Das notwendige Wissen zur Umsetzung der Verknüpfung wird dabei über die Metadaten der Bausteine in der Softwaremodul-Bibliothek definiert, was wiederum die Integration der E/A-Adapter und Hilfstools im Containerimage berücksichtigt.

Beitrag B2 gilt als Lösung für das zweite Problem P2 durch Verwendung einer zentralen Kommunikationsinfrastruktur und verschiedenen E/A-Adaptern.

B3: Parallelisierung und Koordination von Workflows

Ein Koordinationsservice startet die Framework-Bausteine eines Workflows in parallelen und unabhängig voneinander lauffähigen Containern auf einer Laufzeitumgebung und führt sie als Containerinstanzen auf dem Rechnercluster aus. Der Koordinator kann dabei mehr als eine Instanz eines Bearbeitungsvorganges parallel starten. Dann synchronisiert er die Ausgaben von parallelen Containern nach vorgegebenen Koordinierungsregeln, z.B. einer Append-Operation über eine **F**irst **I**n **F**irst **O**ut (FIFO)-Warteschlange oder einer anderen vordefinierten Regel, um die Ausgaben zusammenzuführen und/oder an andere Applikationen weiterzuleiten. Die koordinierte Containervirtualisierung ermöglicht das parallele Ausführen von sonst nur seriell anwendbaren Softwareapplikationen. Die Parallelisierung bietet einen Ansatzpunkt zur Performancesteigerung.

Beitrag B3 wird betrachtet als Lösung für das letzte Problem P4 durch Verwendung des Koordinationsservices, um wissenschaftlichen Applikationen in einer entsprechenden Anzahl separater Container parallel zu starten und skalieren.

1.4 Struktur der Arbeit

Die vorliegende Arbeit ist in die folgenden acht Kapitel gegliedert:

Kapitel 1 - Einleitung motiviert und präsentiert das Thema der Arbeit. Anschließend werden die Probleme, auf die in der Motivation hingewiesen wird, klar dargestellt. Jedes Problem wird durch einen Beitrag angesprochen und die Ziele der Arbeit formuliert.

Kapitel 2 - Technologische Grundlagen legt das Fundament für die nachfolgenden Kapitel. Die technologischen Grundlagen und Konzepte werden dargestellt, um die Grundlagen für die PROOF-Architektur zu erläutern.

Kapitel 3 - Anforderungsanalyse und Stand der Forschung skizziert den Forschungsstand. Relevante Arbeiten hinsichtlich eines aufgestellten Anforderungskatalogs werden analysiert und ausgewertet. Die abgeleiteten offenen Probleme der Literatur werden herausgearbeitet und durch die drei folgenden Beiträge gelöst.

Kapitel 4 - Framework Architektur bildet den ersten Beitrag und konzentriert sich auf die PROOF-Architektur. Zunächst wird eine Multiplayer-Simulations-Architektur (Co-Simulations-Architektur) zur Vorbereitung des Entwurfs der PROOF-Architektur vorgestellt. Darüber hinaus wird ein Überblick über die Multiplayer-Simulations-Architektur mit Aspekten für Web-Benutzeroberfläche, Adapter und einem Multiplayer-Simulations-Beispiel gegeben. Anschließend wird eine Übersicht der PROOF-Architektur vorgestellt. Danach wird eine technische Ansicht der Hauptkonzepte von Framework-Prozess (Baustein), Prozessmanagement, Apache NiFi-Prozessoren fertiggestellt, um den ersten Beitrag zur Automatisierung und Ausführung des wissenschaftlichen Workflows und Wiederverwendbarkeit integrierter Applikationen mit ihren spezifischen Abhängigkeiten und Laufzeitumgebungen zu erklären. Außerdem wird die Web-Benutzeroberfläche der Referenzarchitektur vorgestellt, welche Funktionen zum Erstellen, Betreiben und Verwalten wissenschaftlicher Workflows bietet.

Kapitel 5 - Konfigurierbare Kommunikation zwischen Framework-Prozessen fokussiert sich auf die Darstellung und Diskussion des zweiten Beitrags und beschreibt, wie die Kommunikationsinfrastruktur innerhalb von PROOF aufgebaut wird.

Kapitel 6 - Parallelisierung und Koordination von Workflows umfasst das Konzept zum dritten Beitrag, das die gesamte PROOF-Architektur vervollständigt. Hier wird ein Ansatz zum Parallelisieren, Koordinieren und Synchronisieren des Framework-Prozesses vorgestellt.

Kapitel 7 - Evaluation erbringt die Validierung der einzelnen Beiträge der PROOF-Architektur mithilfe von drei Anwendungsfällen. Zusätzlich wird ein Benchmarking bezüglich Memory, Ausführungszeit und Parallelisierung diskutiert. Anschließend widmet sich dieses Kapitel der Qualitätsanalyse. Schließlich wird die Evaluation der Machbarkeit zur Prüfung auf die vollständige Erfüllung der Anforderungen gegeben.

Kapitel 8 - Zusammenfassung schließt die Dissertation mit einer Zusammenfassung der wichtigsten Lösungen für die wissenschaftlichen Herausforderungen und gibt einen Ausblick auf mögliche weiterführende wissenschaftliche Untersuchungen in diesem Feld.

Technologische Grundlagen

2

Um ein kohärentes Verständnis der weiteren Arbeit zu erlangen, werden in diesem Kapitel zuerst die grundlegenden theoretischen Konzepte und Technologien beschrieben. Zunächst werden die wichtigsten Konzepte und Eigenschaften von Co-Simulation vorgestellt. Darauf folgend werden Technologien vorgestellt, die die unterschiedlichen Beiträge der vorliegenden Arbeit abdecken.

2.1 Co-Simulation

Wissenschaftliche Workflows bestehen aus verschiedenen Applikationen zur Modellierung, Simulation, Datenerfassung und -verarbeitung sowie zur Visualisierung von Ergebnissen (vergleiche Kapitel 1). Zur Simulation eines wissenschaftlichen Workflows müssen alle relevanten Applikationen zum Datenaustausch gekoppelt werden. Beispielsweise im Energieversorgungssystem bietet das Smart Grid hinsichtlich der Integration von Energieressourcen, wie z.B. erneuerbaren Energien, große Chancen. Gleichzeitig stellen aber auch die zunehmende Dezentralisierung der Energieressourcen und die gewünschte Flexibilität die Teilnehmer des Energieversorgungssystems vor große Herausforderungen. Dieses Energieversorgungssystem besitzt aufgrund der hohen Teilnehmerzahl in einem dezentralen System höhere Komplexität, aber die Regeln des Zusammenwirkens sollen Stabilität und Flexibilität des Energieversorgungssystems garantieren. So wird die Komplexität der ohnehin komplexen Steuerung und Entwicklung neuer Komponenten weiter erhöht. Die Vernetzung der verschiedenen Komponenten des Energieversorgungssystems wird dabei durch den Einsatz von IKT ermöglicht. Da das Energieversorgungssystem aber eine kritische Infrastruktur darstellt, kommen Trial-and-Error-Methoden zum Testen neuer Kontrollstrategien oder Komponenten im laufenden Betrieb nicht infrage. Folglich müssen sämtliche Änderungen, die am Energieversorgungssystem vorgenommen werden, vorher z.B. modelliert, simuliert und deren Auswirkungen analysiert werden. Durch einen Simulator kann das Verhalten eines Energieversorgungssystems nachgebildet werden. Der Simulator umfasst ein Modell und einen Solver [Pal+17a]. Ein Modell abstrahiert die Wirklichkeit auf die für die Simulation relevanten Aspekte. Die Aspekte ergeben sich aus dem Anwendungsfall [Sch+15],

und werden durch den Solver interpretiert. Dabei gibt es schon eine Vielzahl an hoch spezialisierten Simulatoren für das elektrische Energienetz und Kommunikationssysteme [MOD14]. Hierbei werden z.B. wirtschaftliche Aspekte oder Wettermodelle, die für die Simulation von erneuerbaren Energien besonders wichtig sind, noch gar nicht betrachtet. Zur Simulation eines komplexen Systems als Ganzes ergeben sich vier verschiedene Strategien, die in Abb. 2.1 verglichen werden:

- „**Klassische**“ **Simulation**: Simulation eines Modells durch einen Solver.
- **Parallel-Simulation**: Simulation eines Modells auf verschiedenen Solvern. Diese Variante wird häufig zur Beschleunigung verwendet.
- **Hybrid-Simulation**: Modelle verschiedener Repräsentationen werden in einer Laufzeitumgebung durch einen Solver gelöst. Die individuellen Teile des Modells können so durch spezialisierte Modellierungsmethoden repräsentiert werden.
- **Co-Simulation**: Modelle unterschiedlicher Repräsentationen werden durch individuelle Solver ausgeführt.

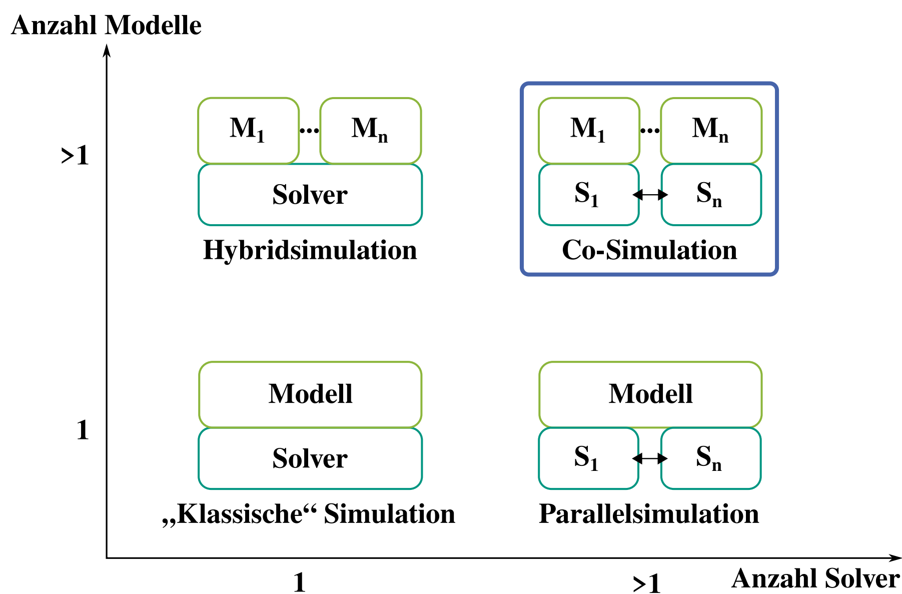

Abb. 2.1.: Simulationskategorien [Pal+17a; HP17]

Die Co-Simulation ermöglicht den Einsatz spezialisierter Modelle und deren Wiederverwendung. Dadurch wird eine aufwändige und fehleranfällige Reimplementierung verschiedener heterogener Simulationen in einer gemeinsamen Umgebung vermieden. Gleichzeitig können sich die Experten jedes Fachgebiets auf ihre spezialisierten Simulationen konzentrieren. Im Folgenden werden die wesentlichen Eigenschaften von Co-Simulationen zu deren leichteren Einordnung vorgestellt.

Nutzer

Eine (Co-)Simulation dient in der Regel dem Erreichen eines wissenschaftlichen Ziels. Im wissenschaftlichen Kontext ist dieses Ziel die Beantwortung wissenschaftlicher Fragen. Um wissenschaftliche Frage beantworten zu können, müssen gewisse Anforderungen der Nutzer erfüllt werden. Je nach Nutzer können sich diese Anforderungen aber stark unterscheiden, sodass es sinnvoll ist, zwischen verschiedenen Nutzergruppen zu unterscheiden. Wie bei einer Anforderungsanalyse in der Softwaretechnik werden hier die Co-Simulationen eingeordnet. Dabei werden die Anforderungen der Akteure erhoben und erfasst. Des Weiteren wird ermittelt, wie die Anforderungen durch eine Simulation bedient werden können. Aufgrund der Vielzahl von Anwendungsfällen, die oftmals Bestandteil aktiver Forschung sind, werden hier beispielhafte Kategorien von Nutzergruppen und Anwendungsfällen vorgestellt [Sch+15]:

- **Netzbetreiber**: Wie beeinflusst die Integration neuer Netzkomponenten die Netzstabilität sowie die Versorgungssicherheit? Wie kann die Steuerung optimiert werden? Welche Einflussgrößen können optimiert werden?
  Fokus: Simulation verschiedener Szenarien der Netzerweiterung und Analyse der Auswirkungen.
- **Produzenten von IKT-Komponenten und -Systemen**: Welche Anforderungen gibt es seitens der anderen Akteure im Smart Grid hinsichtlich der Kommunikation (Bandbreite, Latenz, Zuverlässigkeit, Sicherheit etc.) an die Komponenten?
  Fokus: Anforderungsanalyse und Optimierung der Informationstechnikarchitektur.
- **Produzenten von Netzkomponenten und verteilten Energieressourcen**: Wie müssen die Produkte für das Smart Grid angepasst werden?
  Fokus: Simulation von Anwendungsfällen, um die Anforderungen an die Produkte herauszufinden.
- **Betreiber virtueller Kraftwerke und Energielieferanten**: Wie kann die Steuerung des Smart Grids für maximalen wirtschaftlichen Erfolg optimiert werden?
  Fokus: Entwicklung und Validierung von Geschäftsfällen in Simulationen.
- **Politik und Regulation**: Wie müssen rechtliche und wirtschaftliche Rahmenbedingungen (Grid Codes, Marktdesign, Regulierung usw.) gestaltet werden, um volkswirtschaftliche und/oder technische Ziele zu erreichen?
  Fokus: Auswertung der Auswirkungen durch geänderte Regelungen und neue Gesetze.

- **Bildungssektor und Wissenschaft**: Welche Komponenten gibt es im Smart Grid und wie interagieren diese miteinander?
  Fokus: Verständnis des Smart Grids, Erweiterung bestehender und Aufbau neuer Szenarien.

Zeitliches Verhältnis

Modelle bilden bestimmte Aspekte des realen Smart Grids in Simulationen ab. Da das reale Smart Grid immer in Echtzeit läuft, spielen zeitliche Eigenschaften eine zentrale Rolle bei Co-Simulationen. Für bestimmte Anwendungen wie **H**ardware-**i**n-the-**L**oop (HiL), also die Einbindung von Hardware-Komponenten des Smart Grids in die Simulation, ist eine Ausführung in Echtzeit unabdingbar. Zu unterscheiden sind deshalb folgende Fälle[Sch+15; MOD14]:

- **Langsamer als Echtzeit**: Bei Ausführung auf langsamer Hardware oder wenn bestimmte Phänomene im Detail analysiert werden müssen.
- **Echtzeit**: Bei Einbindung von HiL oder ähnlichen Anwendungen.
- **Schneller als Echtzeit**: Der Normalfall bei der Ausführung von Simulationen. Dies ermöglicht es, viele Experimente im Vorhinein laufen zu lassen, um so beispielsweise die Netzstabilität bei verschiedenen Aktionen analysieren zu können.

Zeitliche Auflösung

Wie in Abschnitt Zeitliches Verhältnis gezeigt, spielen die zeitlichen Aspekte eine entscheidende Rolle bei Co-Simulationen. Dazu gehört auch die zeitliche Auflösung, da bestimmte Phänomene im Smart Grid nur bei entsprechenden zeitlichen Auflösungen analysiert werden können. Die Grenzen der Kategorien sind dabei fließend [Sch+15; MOD14]:

- **Gleichgewichtszustand** (Sekunden und darüber): Das System ist im Gleichgewicht und Phänomene wie der optimale Lastfluss können analysiert werden. Auch Daten von Wettersimulationen (Globalstrahlung oder ähnliches) werden in diesem Bereich simuliert.
- **Elektromechanischer Bereich** (unter Sekunden): Das Verhalten mechanischer Generatoren hinsichtlich Stabilität und Kurzschlussverhalten wird analysiert.
- **Elektromagnetischer Bereich** (Millisekunden und darunter): Elektromagnetische Wellen sind der dominierende Mechanismus. Beispielsweise wird das Verhalten von **F**lexible-**AC**-**T**ransmission-**S**ystems (FACTS) analysiert.

Kommunikationsreihenfolge

Ein weiterer wichtiger Aspekt bei der Zusammenarbeit verschiedener Simulationen ist die Kommunikation. Die Kommunikationsreihenfolge legt fest, in welcher Reihenfolge die Simulationen Daten austauschen. Zu unterscheiden sind:

Nicht iterativ

- **Parallele Sequenz**: Wie in Abb. 2.2 links zu sehen ist, werden die Simulationen parallel ausgeführt (hier für den einfachsten Fall mit zwei Simulationen gezeigt). Im ersten Schritt zum Zeitpunkt t_k werden die Daten ausgetauscht, im zweiten Schritt extrapolieren die Simulationen dann jeweils die Daten der anderen Simulationen, die sie zur Berechnung des nächsten Zeitpunkts t_{k+1} (dritter Schritt) benötigen. Im einfachsten Fall wird für die benötigten Daten der anderen Simulationen zu t_{k+1} der gleiche Wert wie bei t_k angenommen. Bei der Extrapolation entstehen also Fehler. Der vierte Schritt entspricht dann wieder dem ersten.

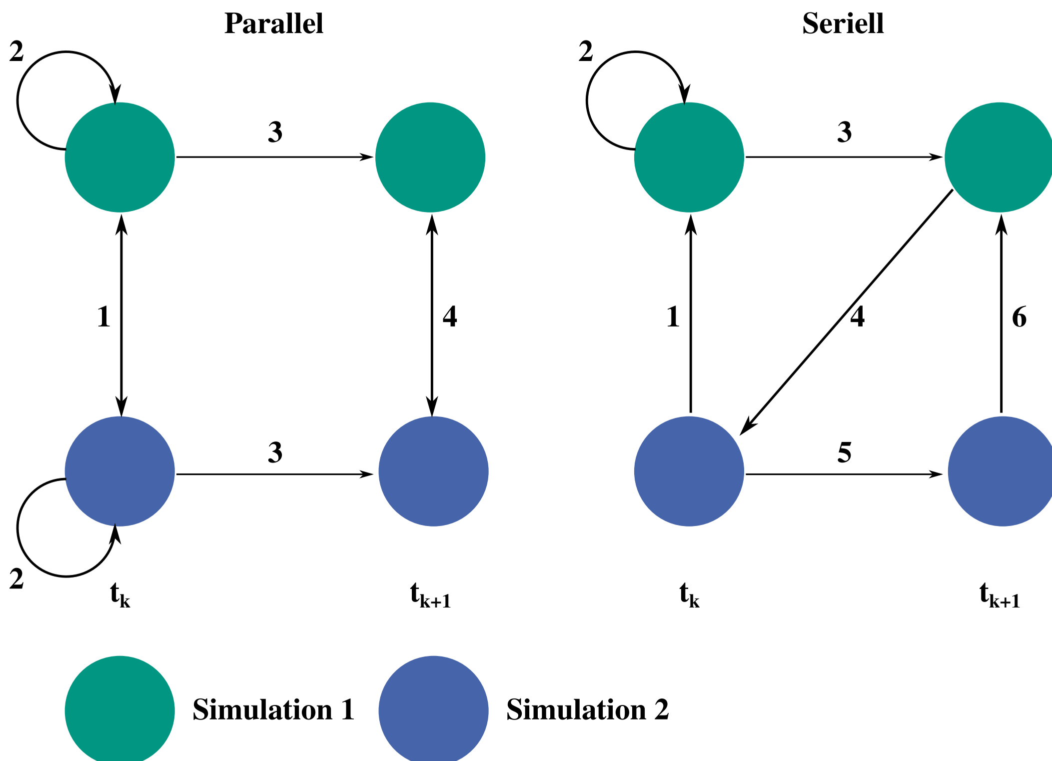

Abb. 2.2.: Kommunikationsreihenfolge – nicht iterativ [Pal+17a]

- **Serielle Sequenz**: Wie in Abb. 2.2 rechts zu sehen ist, werden die Simulationen seriell ausgeführt (auch hier wieder für den einfachsten Fall mit zwei Simulationen gezeigt). Im ersten Schritt übergeben die Simulationen ihre

Daten an die „erste“ Simulation. Diese extrapoliert die für die Berechnung der eigenen Simulationsdaten benötigten Daten der anderen Simulationen im zweiten Schritt (wie schon bei der parallelen Sequenz). Diese Extrapolation findet aber nur bei einer der Simulationen statt, nämlich der „ersten“. Danach startet die „erste“ Simulation den Übergang vom Zeitpunkt t_k zu t_{k+1}. Anschließend werden im vierten Schritt die Daten an die nächste Simulation übergeben, die dann zur Berechnung startet. Im fünften Schritt berechnet diese dann den Übergang von t_k zu t_{k+1}. So berechnen nach und nach alle Simulationen den Übergang von t_k zu t_{k+1}. Der sechste Schritt entspricht dann wieder der Übergabe der Werte an die „erste“ Simulation. Durch das Ausführen der Extrapolation bei nur einer Simulation wird der Fehler durch die Extrapolation geringer. Gleichzeitig wird aber die Ausführungszeit größer, da die Simulationen seriell ausgeführt werden.

Iterativ

Um den Fehler durch Extrapolation bei der nicht iterativen Variante zu minimieren, können sowohl die parallele Sequenz als auch die serielle Sequenz iterativ ausgeführt werden. Der Übergang von Zeitpunkt t_k zu t_{k+1} wird mehrfach nacheinander ausgeführt, um den Fehler durch mehrfachen Austausch der Simulationsdaten zu verringern. Meist wird so oft iteriert, bis ein bestimmtes Konvergenzkriterium erfüllt ist. In Abb. 2.3 werden beispielhaft zwei Iterationen der parallelen Sequenz gezeigt.

Die Schritte 1-3 werden wie bei der nicht iterativen Variante ausgeführt. Der vierte Schritt unterscheidet sich dann aber insofern, dass die Simulationen wiederum ihre Daten austauschen, aber wieder von t_{k+1} zu t_k zurückgehen. Anschließend wird im fünften Schritt wieder extrapoliert. Im sechsten Schritt wird analog zum dritten Schritt der Übergang von t_k zu t_{k+1} berechnet. Anschließend könnte eine weitere Iteration stattfinden, wobei das Beispiel hier auf zwei Iterationen beschränkt ist. Der siebte Schritt gehört dann bereits zur ersten Iteration des Übergangs von t_{k+1} zu t_{k+2}. Die folgenden Schritte erfolgen dann wie beim Übergang t_k zu t_{k+1}. Die iterative Variante bei der seriellen Sequenz funktioniert analog. Eine wichtige Voraussetzung für die iterative Variante der parallelen und seriellen Sequenz ist die Fähigkeit eines Simulators einen Zeitschritt wiederholt zu berechnen. Da viele Simulatoren dies nicht unterstützen, ist die iterative Kommunikation hier nicht möglich [Ste+18].

Synchronisation

Bei einer parallelen Ausführung von Simulatoren und dem gemeinsamen Ziel einer Co-Simulation ist eine Synchronisation unabdingbar, da die Simulatoren zu

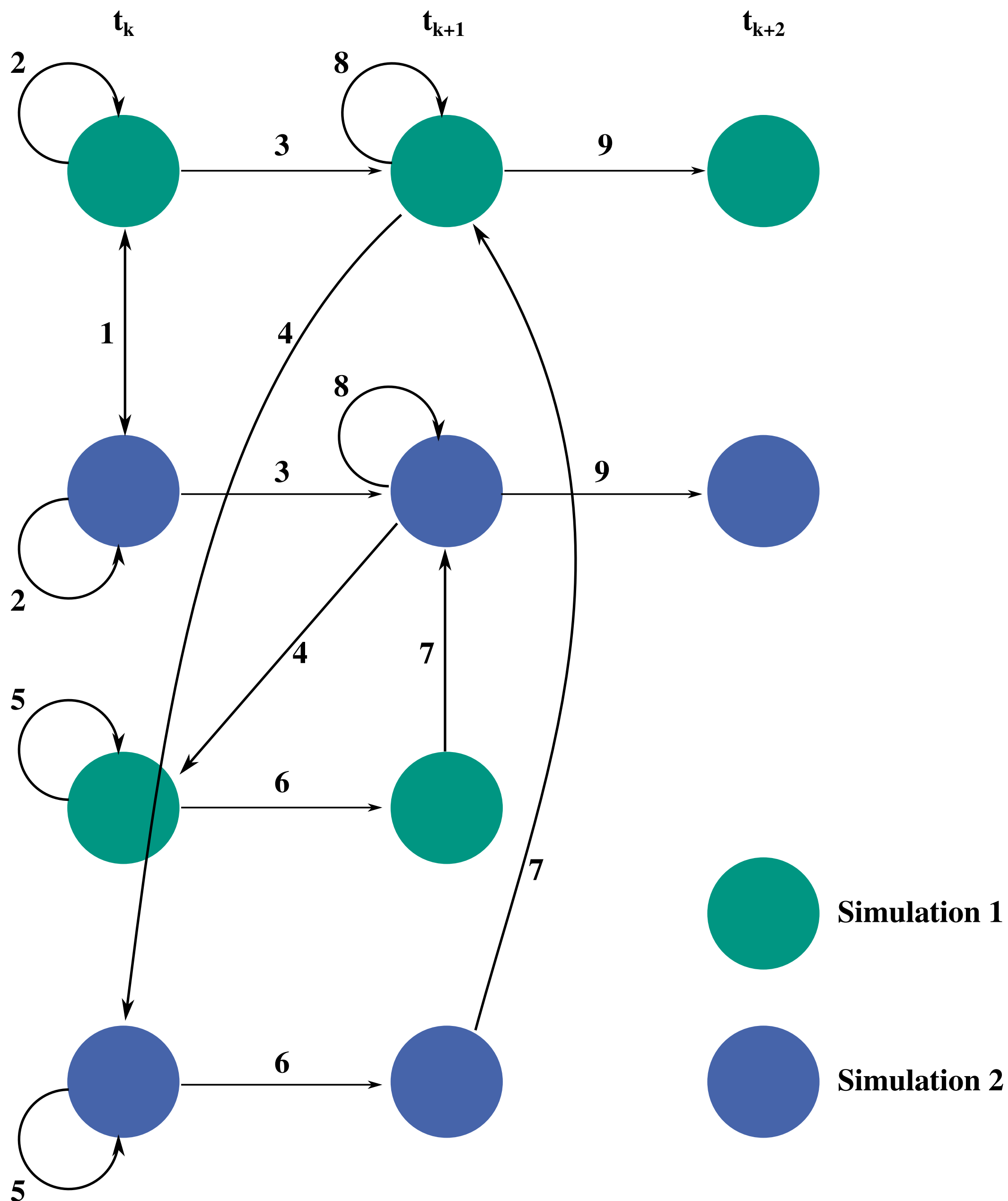

Abb. 2.3.: Kommunikationsreihenfolge – iterativ, parallele Sequenz [Pal+17a]

bestimmten Zeitpunkten Daten austauschen müssen. Zur Synchronisation der verschiedenen Typen von Simulationen werden in der Literatur folgende drei Varianten unterschieden [Pal+17a; Lin+11]:

1. **Feste Zeitpunkte**: Bei dieser Methode werden im Vorhinein feste Synchronisationszeitpunkte festgelegt. Die Simulatoren laufen parallel und halten immer an den Synchronisationszeitpunkten, um dann Informationen auszutauschen. In Abb. 2.4 tauschen die Simulation des Energiesystems (hier mit fester Schritt-

länge simuliert, durch vertikale Pfeile auf der Zeitachse t symbolisiert) und die Simulation des Kommunikationsnetzes (Ereignisse 1-6) an den Zeitpunkten t_0, t_1 und t_2 Daten aus.

- Vorteile: Das Verfahren ist intuitiv.
- Nachteile: Es entstehen Ungenauigkeiten durch verpasste Ereignisse. Wenn beispielsweise die Simulation des Kommunikationsnetzes zum Ereignis 2 die Daten der Energiesystemsimulation benötigt, muss diese bis zum Synchronisationszeitpunkt t_1 warten. Um die Fehler, die daraus entstehen, zu verkleinern, kann die Zeit zwischen zwei aufeinanderfolgenden Synchronisationszeitpunkten t_i und t_{i+1} verkürzt werden. Dadurch steigt aber wiederum der Kommunikationsaufwand und somit die Laufzeit insgesamt.

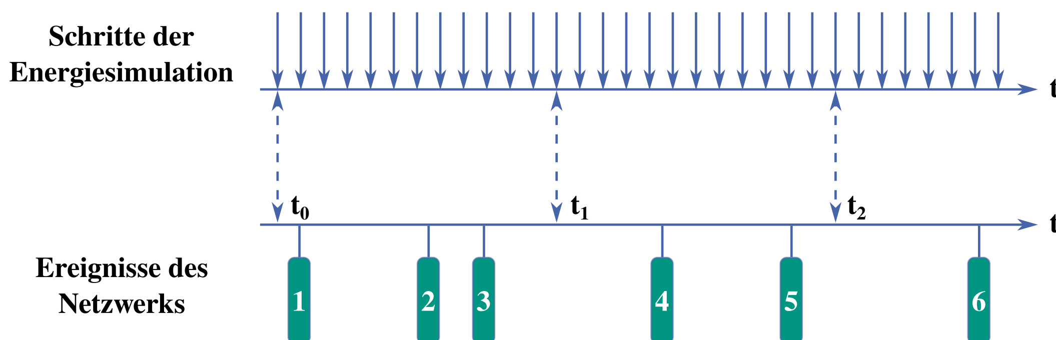

Abb. 2.4.: Synchronisation durch feste Zeitpunkte [Pal+17a]

2. **Ereignisgetrieben**: Bei der ereignisgetriebenen Synchronisation werden alle Zeitschritte bei der Simulation des Energiesystems als Ereignisse aufgefasst. Sie werden also genauso behandelt wie die Ereignisse des Kommunikationsnetzes. Sämtliche Ereignisse werden in einer globalen Ereigniswarteschlange gesammelt. In Abb. 2.5 werden die Schritte der Energiesystemsimulation sowie die Ereignisse des Kommunikationsnetzes auf eine gemeinsame Zeitachse aufgetragen und dann gemäß ihrer zeitlichen Reihenfolge in einer globalen Ereigniswarteschlange von einem Scheduler abgearbeitet. Es wird immer dann synchronisiert, wenn eines der Ereignisse dies erfordert.
   - Vorteile: Es entstehen keine Ungenauigkeiten durch verpasste Ereignisse.
   - Nachteile: Die Co-Simulationszeit ist direkt von der Schrittgröße der Energiesystemsimulation abhängig. Durch kleine Schrittlängen bei der Energiesystemsimulation muss potenziell viel kommuniziert werden, wodurch die Laufzeit steigt. Die Schnittstellen der Simulationen können so zum Flaschenhals werden.

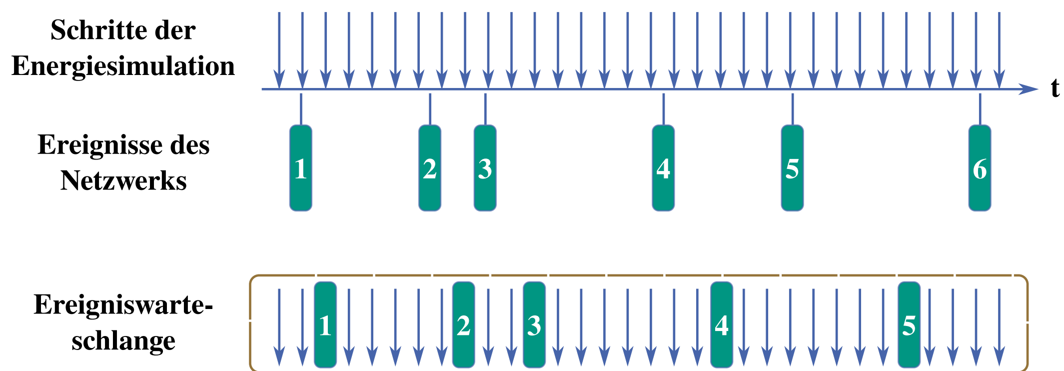

Abb. 2.5.: Ereignisgetriebene Synchronisation [Pal+17a]

3. **Primary-Secondary**: Bei der Primary-Secondary-Synchronisation steuert ein Primary-Simulator die Co-Simulation und somit alle anderen Secondary-Simulationen. In Abb. 2.6 agiert der Simulator des Kommunikationsnetzes als Primary-Simulator und die Energiesystemsimulation als Secondary-Simulator. Der Primary-Simulator simuliert so lange (Makroschritt), bis er die Daten eines Secondary-Simulators braucht, der dann aufgerufen wird. Im Beispiel benötigt der Primary-Simulator erst beim zweiten Ereignis zum Zeitpunkt t_1 die Daten des Secondary-Simulators. Dieser wird dann aufgerufen und berechnet ebenso bis t_1 und gibt die Daten dann an den Primary-Simulator. Der Primary-Simulator berechnet dann das Ereignis 3, bei dem er keine Daten vom Secondary-Simulator braucht. Erst beim 4. Ereignis (Zeitpunkt t_2) hält der Primary-Simulator wieder an, ruft den Secondary-Simulator auf, dieser berechnet bis t_2 und übergibt die Daten an den Primary-Simulator. Zwischen Ereignis 4 und 5 vergeht etwas mehr Zeit, aber der Primary-Simulator benötigt die Daten des Secondary-Simulators erst wieder beim 6. Ereignis. Es könnte auch sein, dass der Primary-Simulator erst die Daten beim 7. oder schon beim 5. Ereignis vom Secondary-Simulator braucht, wodurch sich der Synchronisationszeitpunkt verschieben würde.
   - Vorteile: Die Implementierung gestaltet sich einfach.
   - Nachteile: Die Ereignisse beim Secondary-Simulator zwischen zwei Synchronisationszeitpunkten werden ignoriert und führen somit zu Fehlern. Die Ausführung erfolgt zumeist sequentiell, wodurch eine Beschleunigung durch Parallelität entfällt. Da der Primary-Simulator die gesamte Co-Simulation steuert, kann dieser sich als Flaschenhals bei vielen Simulationen herausstellen. Dies kann aber umgangen werden, indem ein separater Primary-Simulator (keiner der Simulatoren) installiert wird.

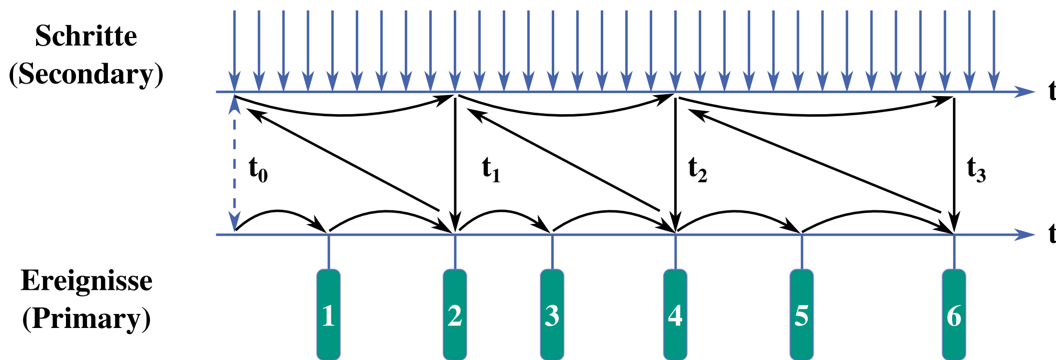

Abb. 2.6.: Primary-Secondary-Synchronisation [Pal+17a]

Simulationstyp

Beim Simulationstyp soll nochmals eine Unterscheidung der Co-Simulationen hinsichtlich der Abstraktion des Verhaltens des simulierten Systems vorgenommen werden. Zu unterscheiden sind hier zwei Typen [Pal+17a; Sch+15]:

- **Hardware-Unterstützung**: Unterstützung durch spezialisierte Hardware, um die Performanz zu steigern.
- **HiL**: Zur Einbindung realer Elemente des Smart Grid in die Simulation. Dies umfasst physische Hardware, Softwareprogramme und Menschen.

Elemente der Modellierung

Bei der Modellierung des Smart Grids können folgende Subkategorien der simulierten Elemente unterschieden werden [Sch+15]:

- **Kontinuierliche Prozesse**: Energiesysteme können durch kontinuierliche Prozesse modelliert werden.
- **Diskrete Prozesse und Ereignisse**: In Kommunikationsnetzen treten Ereignisse diskret auf. Komplexe Simulationen werden zumeist numerisch in diskreten Zeitschritten gelöst.
- **Rollen**: Das Verhalten von Individuen, wie beispielsweise von Energiehändlern, kann simuliert werden.
- **Statistische Elemente**: Wettermodelle sind insbesondere für die Einbindung erneuerbarer Energien in eine Simulation wichtig, um die eingespeiste Energie berechnen zu können.

Zugänglichkeit interner Modellstrukturen

Je nachdem, wie viel über die internen Modellstrukturen des Systems, das simuliert werden soll, bekannt ist, können diese Strukturen durch das Modell der Simulation nachgebildet werden [Sch+15].

- **Black Box**: Es wird nur das Verhalten beobachtet und imitiert.
- **Grey Box**: Es sind Teile der internen Modellstrukturen bekannt.
- **White Box**: Abbildung aller physikalischen und funktionalen Teile sowie Wechselwirkungen zwischen diesen auf ein Modell.

Nutzbarkeit

Bei der Nutzbarkeit sind generell die Nutzung durch **A**pplication **P**rogramming **I**nterface (API) oder eine **G**raphical **U**ser **I**nterface (GUI) zu unterscheiden. Da die Simulationen von Smart Grids immer komplexer werden und die Entwickler der Werkzeuge nicht mehr unbedingt auch deren Nutzer sind, ist die Bereitstellung einer GUI ein großer Vorteil bei der Nutzbarkeit. Gleichzeitig ist eine GUI aber auch deutlich beschränkter hinsichtlich der Funktionalität und ein größerer Entwicklungsaufwand als eine API [Sch+15].

Standards

Die Co-Simulation von Smart Grids wird durch die Unterstützung von bestehenden Standards zur Kontrolle, Steuerung und Datenübertragung deutlich vereinfacht. Exemplarisch werden hier, obwohl es noch viele andere gibt, folgende vier Standards aufgeführt [Pal+17a; MOD14]:

- **IEC 61850**: Steuerung und Übertragung von Daten
- **Common Information Model (CIM)**: Repräsentation von Szenarien
- **OPC UA**: Maschine-zu-Maschine-Kommunikationsprotokoll
- **Functional Mock-up Interface (FMI)**: Einfacher Austausch von Modellen

2.2 Stand der Technik

In diesem Abschnitt werden die wichtigsten Softwaretools und -technologien erläutert, die in Abb. 2.7 dargestellt sind und zur Implementierung des PROOF-Konzepts verwendet werden.

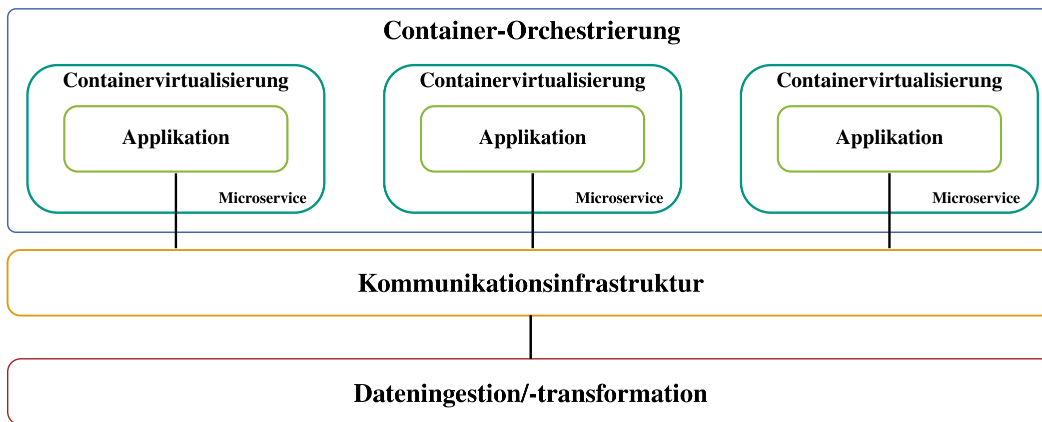

Abb. 2.7.: Stand der Technik zur Implementierung des PROOF-Konzepts

2.2.1 Microservices-basierte Architektur

Eine Microservices-basierte Architektur ist ein Architekturstil, der seinen Schwerpunkt auf die Aufteilung und Gestaltung einer gesamten Anwendung als eine Reihe von lose gekoppelten, leichtgewichtigen, unabhängigen Diensten legt [HS17; Dra+16; Ric22]. Jeder unabhängige Dienst kann unabhängig entwickelt, getestet, bereitgestellt, überwacht, skaliert und sogar in verschiedenen Programmiersprachen implementiert werden [HS17; Dra+16; SP17]. Als Zusammenfassung aus [Kha+16; Vil+16; Ama+15a; Bak17] zeigt Tab. 2.1 einen Vergleich zwischen einer Microservices-basierten und einer monolithischen Architektur.

Gemäß dem Vergleich von sieben verschiedenen Kategorien in Tab. 2.1 kann eine Microservices-basierte Architektur ein sehr überzeugender Architekturstil zum Aufbau einer Workflow-Umgebung mit dem Ziel sein, unabhängig voneinander lauffähige Aufgaben (z.B. beliebige Ausführungsumgebungen) lose zu koppeln. Darüber hinaus kann durch die Verwendung einer Microservice-basierten Architektur das gesamte vorgestellte PROOF-Konzept, das als Satz separater Microservices aufgebaut ist, einfach, schnell und flexibel entwickelt, getestet, versioniert, bereitgestellt und skaliert werden.

2.2.2 Containervirtualisierung Docker

Die containerbasierte Virtualisierung unterscheidet sich geringfügig von der Virtualisierung dadurch, dass eine unabhängige Umgebung (Container) einer **V**irtuelle **M**aschine (VM) ähnelt, jedoch ohne virtuelle Hardwareemulation [Nai16].

Tab. 2.1.: Vergleich von monolithischen und Microservices-Architekturen (Erweiterung von [Bak17]

Kategorie	Monolithische Architektur	Microservices-Architektur
Code	Eine einzige Codebasis für die gesamte Software	Jeder Microservice hat seine eigene Codebasis
Verständlichkeit	Oft verwirrend und schwer zu warten	Viel bessere Lesbarkeit und viel einfacher zu warten
Entwicklung	Komplexe Entwicklungen mit Wartungsschnittstelle	Einfache Entwicklung, da jeder Microservice einzeln schnell entwickelt werden kann
Programmier-sprache	Typischerweise vollständig in einer Programmiersprache entwickelt	Jeder Microservice kann in einer anderen Programmiersprache entwickelt werden
Skalierung	Die gesamte Anwendung muss skaliert werden, auch wenn Engpässe lokalisiert sind	Ermöglicht die Skalierung von Services mit Engpässen, ohne die gesamte Anwendung zu skalieren
Dokumentation	Gewachsene Dokumentation, die schwieriger zu lesen ist, je älter der Monolith ist.	Einfache Dokumentation durch kleine Codebasis und Feature-Set. Standardisierte Dokumentation der API-Schnittstelle
Testen	Bei großen Softwaresystemen wird das Testen sehr schwierig. Codeabdeckung ist nahezu unmöglich	Microservices können mithilfe einer DevOps-Pipeline automatisch getestet werden. Tests sind einfacher zu warten, da die Codebasis klein ist

Nach [Liu+18; Ama+15a; Sol+07; Fel+15] können sich virtuelle Container das Host-Betriebssystem teilen und den Verwaltungsaufwand (Probleme mit Speicherplatzmangel, Betriebssystem-Update usw.) reduzieren. Im Vergleich zu VMs können Container als Virtualisierung auf Softwareebene angesehen werden, da Container im Benutzerbereich über dem Betriebssystemkernel erstellt werden [Tur14].

Docker ist eine Open-Source-Container-Virtualisierungsplattform auf Betriebssystemebene für eine einfache und automatisierte Installation und Ausführung von Applikationen innerhalb von Docker-Containern mithilfe der Betriebssystemvirtualisierung [Inc22]. Docker nutzt Ressourcenisolationsfunktionen wie Cgroups und Linux-Kernel, um isolierte Container für laufende Applikationen zu erstellen. Daher

können die Applikationen von dem Betriebssystem, auf dem sie ausgeführt werden, getrennt und isoliert werden [Liu+18; Nai17; Mer14; Inc22]. Im Unterschied zu einer VM bietet Docker die Möglichkeit, mehrere Applikationen innerhalb desselben Host-Betriebssystems auszuführen und zugrunde liegende Ressourcen (**C**entral **P**rocessing **U**nit (CPU), **R**andom-**A**ccess **M**emory (RAM) usw.) gemeinsam zu nutzen, wie in Abb. 2.8 gezeigt [Tur14]. Da außerdem jedes Docker-Image eine oder mehrere Anwendungen und alle notwendigen Laufzeitabhängigkeiten enthält, z.B. Betriebssystem- und Framework-Bibliotheken sowie die Applikation-Binärdateien (siehe oberer Teil von Abb. 2.8), können Docker-Images von Benutzern einfach geteilt und in Form verschiedener Docker-Container ausgeführt werden. Darüber hinaus kann die Bereitstellung von Applikationen einfach automatisiert werden [Tur14; Nai17; Inc22; LX16].

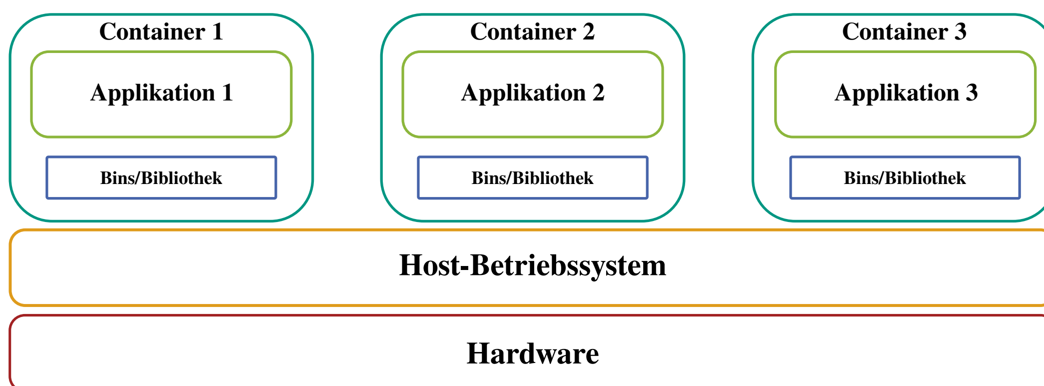

Abb. 2.8.: Docker Container [Tur14]

Durch die Verwendung von Docker kann das vorgestellte PROOF-Konzept die Laufzeitumgebung von Workflows auf Docker-Images abstrahieren. Jedes Docker-Image beinhaltet die Bausteine für eine oder mehrere Applikationen, welche als synchronisierte Prozesse in einem Docker-Container mit anderen Containern gleichzeitig auf dem zugrunde liegenden Rechencluster laufen können. Dies trägt zu einem modularen und lose gekoppelten Ansatz bei. Viele der Vorteile von Docker-Containern und Microservices-Technologies führen direkt zu einer erhöhten Skalierbarkeit und erweiterten Flexibilität von PROOF.

2.2.3 Container-Orchestrierung Kubernetes

Docker bietet Tools zum Erstellen von Container-Images und zum lokalen Testen der Anwendungscontainer. Doch wie lässt sich ein Workflow, der aus vielen Microservices besteht, verwalten und steuern? Dazu kommen produktionsbereite Container-Orchestrierungsplattformen ins Spiel, die Container auf mehreren Knoten

eines Computing-Clusters bereitstellen können. Ein Beispiel für eine solche Plattform zur Erfüllung dieser Anforderungen ist die Kubernetes-Plattform [Ber14]. Laut [Fou22d] ist Kubernetes eine Plattform zur Automatisierung der Bereitstellung, Skalierung und Verwaltung von containerisierten Applikationen. Zusammen mit Docker wird Kubernetes verwendet, um Applikationen (und damit auch Microservices) in Containern über Cluster von Rechenknoten hinweg bereitzustellen. Es enthält Tools für die Orchestrierung, z.B. die Handhabung einer Reihe von ausführbaren Dateien als eine Applikation, die Diensterkennung (z.B. die Identifizierung von Instanzen eines bestimmten Typs und die Überprüfung ihres Zustands auf Fehlertoleranz) und den Lastausgleich.

Durch die Verwendung von Kubernetes können ein oder mehrere Container logisch einem Pod zugeordnet werden. Ein Pod ist eine grundlegende Planungseinheit für Kubernetes, um einen Anwendungsteil (z.B. bestehend aus einem oder mehreren Diensten mit enger Beziehung) als eine Reihe von Docker-Images bereitzustellen und zu verwalten. Pods können Ressourcen, die von Anwendungsteilen benötigt werden, über die zugeordneten Container hinweg gemeinsam nutzen, z.B. einen bestimmten Datenspeicher, der von einem verteilten Dateisystem oder einer bestimmten Netzwerkadresse bereitgestellt wird. Kubernetes kann automatisch eine Maschine finden, die über genügend freie Rechenkapazität für einen bestimmten Pod verfügt, und die zugehörigen Container starten. Um Konflikte zu vermeiden, wird jedem Pod eine eindeutige **I**nternet **P**rotocol (IP)-Adresse zugewiesen, sodass Anwendungen verschiedene Ports unter derselben IP-Adresse verwenden können. Kubernetes bietet Verfügbarkeits- und Qualitätsprüfungen für Container, um ausgefallene Container in Pods durch seine automatischen Fehlerwiederherstellungsaktionen zu reparieren [Abd+18]. Wie in [Mod+18] dargestellt, ist Kubernetes im Vergleich zu Docker Swarm, einer weiteren Container-Orchestrierungsplattform [Fre22], leistungsfähiger und bietet gleichzeitig hohe Skalierbarkeit und Automatisierung. Die Leistung von Kubernetes übertrifft derzeit die von Docker Swarm. Durch die Verwendung von Kubernetes für PROOF können containerisierte Applikationen, die in Pods ausgeführt werden, auf dem Rechencluster einfach und effizient bereitgestellt, verwaltet und gesteuert werden.

2.2.4 Kommunikationsinfrastruktur

Ein weiterer wichtiger Teil eines wissenschaftlichen Workflow-Frameworks hängt mit der Frage zusammen, wie der Informationsfluss zwischen Workflow-Applikationen organisiert und koordiniert werden kann. Da Applikationen in verteilten Containern ausgeführt werden, wird eine adäquate verteilte Technologie für die Koordination

und den Nachrichtenaustausch dazwischen benötigt. Solche Kommunikations- und Koordinierungsdienste basieren häufig auf Schlüsselwertdatenbanken als Speicher (z.B. etcd [Fou22c], Apache Zookeeper [GM15] usw.) und verwenden schließlich ein Nachrichtenprotokoll als Kommunikationsschnittstelle (z.B. **Re**mote **Di**ctionary **S**erver (Redis), RabbitMQ [VMW23; OAS23] oder Apache Kafka).

Apache Zookeeper

Apache ZooKeeper ist ein Open-Source-Server für hochzuverlässige verteilte Koordination von Cloud-Anwendungen [Fou23]. ZooKeeper ist im Wesentlichen ein Dienst für verteilte Systeme, der einen hierarchischen Schlüsselwertspeicher bietet, der verwendet wird, um einen verteilten Konfigurationsdienst, einen Synchronisationsdienst und eine Namensregistrierung für große verteilte Systeme bereitzustellen. Es wird im Falle eines Ausfalls automatisch wiederhergestellt [McD23].

Apache Kafka

Apache Kafka ist eine nachrichtenorientierte, verteilte Daten-Streaming-Plattform, die über eine Java-API verwendet wird. Als Nachrichtenserverumgebung generalisiert Apache Kafka unter Verwendung des Consumer-Gruppe-Konzepts die beiden traditionellen Messaging-Modelle Queuing und Publish-Subscribe. Beim Queuing wird die Verarbeitung von Daten durch die Consumer-Gruppe auf mehrere Consumerinstanzen (die verschiedenen Simulationsmitglieder der Consumer-Gruppe) verteilt, um die Verarbeitung zu skalieren. Durch die Instrumentierung von Publish-Subscribe kann Apache Kafka Nachrichten an mehrere Consumer-Gruppen senden, die Daten parallel für unterschiedliche Zwecke verarbeiten können. Für den Aufbau von Echtzeit-Datenpipelines und Datenverarbeitungsanwendungen sind die vier Kern-APIs definiert [Fou22a; LDH17]:

- Producer-API zum Veröffentlichen eines Nachrichtenstroms
- Consumer-API zum Abonnieren eines oder mehrerer Topics und zum Verarbeiten des Nachrichtenstroms
- Streams-API zum Konsumieren eines Eingabestreams und zum Produzieren eines Ausgabestreams
- Connector-API zum Erstellen und Ausführen von Producern oder Consumern, die Kafka-Topics mit vorhandenen Anwendungen oder Datensystemen verbinden

Redis

Redis ist ein In-Memory-Datenstrukturspeicher, der als nicht-relationale Datenbank, Cache und Nachrichtenbroker verwendet werden kann [Ltd22a]. Aufgrund seines In-Memory-Konzepts setzt Redis nicht auf Plattenspeicher, sondern auf Hauptspeicher. Daher bietet Redis eine hohe Leistung für Lese- und Schreibvorgänge im Gegensatz zu klassischen Datenbankmanagementsystemen. Um High-Level-Datenstrukturen als Werte zu bilden, bietet Redis außerdem fünf abstrakte Datentypen für Werte: Strings, Lists, Sets, Hashes und sortierte Sets. Redis ist in ANSI C geschrieben und leichtgewichtig, ohne externe Abhängigkeiten. Darüber hinaus bietet Redis atomare Operationen für Lese-/Schreibaktionen zur Verwendung von Datenstrukturen zur Koordination. Für die Cluster-Umgebung bietet Redis Cluster Hochverfügbarkeit und Skalierbarkeit für verteilte Systeme.

Um auf den Entwurf der PROOF-Architektur vorzubereiten, wird eine Multiplayer-Simulations-Architektur (Co-Simulations-Architektur) in der vorliegenden Arbeit konzipiert und implementiert. Zur Realisierung von Datenaustausch zwischen Applikationen wird Apache Kafka eingesetzt. Für das PROOF-Konzept wird Redis statt Apache Kafka als Kommunikationsschnittstelle integriert, da sich der Datenaustausch durch die Nutzung seiner praktischen Publish/Subscribe-Funktionalität im Gegensatz zu Apache Kafka effizient und fehlertolerant umsetzen lässt.

Basierend auf den Anforderungen der vorliegenden Arbeit wird ein weiterer ausführlicher Vergleich verschiedener Kommunikationsarchitekturen in Kapitel 5 dargestellt.

2.2.5 Dateningestion/-Transformation Apache NiFi

Apache NiFi (Abkürzung: NiFi) ist eine Workflow-Anwendung zur Datenaufnahme und bietet eine gute Reihe von Basistechnologien zum Einrichten einer wissenschaftlichen Workflow-Umgebung. Es ist ein Tool zur Automatisierung der Datenverarbeitung und ermöglicht das Senden, Empfangen, Weiterleiten, Transformieren und Sortieren von Daten. Es bietet Echtzeitkontrolle zum Verwalten der Bewegung von Daten zwischen jeder Quelle und jedem Ziel. Das Kernfluss-basierte Programmierkonzept von Apache NiFi enthält die folgenden grundlegenden Komponenten [Fou22b; SBS17]:

- *FlowFile*: Ein Dateiobjekt, das das zwischen Prozessoren übertragene Datenobjekt darstellt.

- *Processor*: Eine logische Komponente zum Erstellen, Empfangen, Senden, Konvertieren, Weiterleiten, Teilen, Zusammenführen und Verarbeiten eines *FlowFile*.
- *Connection*: Eine Zuordnung zwischen Prozessoren in Form von Java-basierten Warteschlangen, um die Ausführungsbeziehung zwischen Prozessoren zu definieren.
- *Flow Controller*: Ein Controller hält die Verbindung zwischen den Prozessoren aufrecht, verwaltet und weist Threads zu, die von allen Prozessoren verwendet werden.
- *Process Group*: Eine Reihe von Prozessen, die Daten über den Eingangsport empfangen und Daten über den Ausgangsport senden können.

Apache NiFi enthält mehrere Funktionen, um die Herausforderungen des Internet of Things anzugehen [Liu+18; SBS17]:

- Eine webbasierte grafische Benutzeroberfläche bietet Benutzern eine intuitive visuelle Darstellung des Datenflusses und der Bearbeitungsfunktionen.
- Konfigurierbare Workflows können vor dem Start gespeichert, geladen und geändert werden.
- Es bietet eine Erweiterungsschnittstelle zum Erstellen und Hinzufügen eigener Prozessoren.
- Erweiterte Sicherheit mit **S**ecure **S**ockets **L**ayer (SSL), **S**ecure **SH**ell (SSH), **H**yper**t**ext **T**ransfer **P**rotocol **S**ecure (HTTPS) usw. ist integriert.

Durch die Integration von Apache NiFi und die Erweiterung für Koordinations- und Co-Simulationsinteraktionen ermöglicht PROOF Benutzern die einfache Erstellung, Ausführung und Verwaltung von Datenverarbeitungs- oder Simulationsworkflows über die webbasierte NiFi-Benutzeroberfläche. Mit der abstrakten Java-Prozessorklasse von Apache NiFi können eigene Prozessoren für die Modellierung benutzerdefinierter Workflows entworfen werden. Darüber hinaus können mehrere Prozessoren (z.B. ListenHTTP- und InvokeHTTP-Prozessoren) einfach und direkt für die Datenintegration, -transformation, -aufnahme und -übertragung verwendet werden, ohne etwas zu ändern.

2.2.6 REpresentational State Transfer (REST)

Als ein wichtiges Kommunikationskonzept wird REST zur Realisierung vieler Schnittstellen moderner Microservice-Architekturen eingesetzt. Der Begriff **RE**presentational **S**tate **T**ransfer wurde von Roy Fielding in seiner Dissertation [FT00] eingeführt.

Heutzutage wird es in einer Vielzahl von Anwendungen verwendet, da es einen leichtgewichtigen Kommunikationsmechanismus für Webanwendungen und -dienste definiert. REST wurde entwickelt, um Interoperabilität zwischen Computersoftwareanwendungen oder -programmen über ein Netzwerk bereitzustellen, wie z.B. Dienste und Webanwendungen, die im Internet bereitgestellt werden.

REST-APIs basieren auf dem HTTP. Darüber hinaus sind REST-APIs einfach zu dokumentieren. Ein RESTful-Dienst stellt die grundlegenden **C**reate, **R**ead, **U**pdate and **D**elete (CRUD)-Funktionen bereit, indem er die HTTP-Methoden wie folgt instrumentiert. Die GET-Methode wird verwendet, um eine oder mehrere Ressourcen anzufordern. Die PATCH-Methode wird für teilweise Aktualisierungen einer Ressource verwendet, und die PUT-Methode wird verwendet, um eine Ressource vollständig zu ersetzen. Die POST-Methode wird verwendet, um eine neue Ressource zu erstellen, und gibt den Speicherort der neuen Ressource als Header-Feld zurück. Schließlich löscht die DELETE-Methode eine Ressource vollständig. Außerdem kann das POST-Verfahren für komplexere Operationen verwendet werden, beispielsweise als Trigger zum Starten der Berechnung einer Datenanalyse.

Unter Verwendung von REST bietet PROOF Interaktionsschnittstellen zum Datenaustausch mit fremden Diensten und Anwendungen im Internet.

3 Anforderungsanalyse und Stand der Forschung

Aus der Analyse der in Kapitel 1.2 identifizierten Probleme werden mehrere Anforderungen abgeleitet und im ersten Teil dieses Kapitels erörtert. Sie dienen dazu, relevante Co-Simulationsplattformen/Automatisierungsframeworks und ebenso die in dieser Arbeit erzielten Ergebnisse zu bewerten. Die Anforderungen werden im Folgenden erläutert und in einem Anforderungskatalog zusammengefasst. Der Anforderungskatalog bildet die Basis für den zweiten Teil des Kapitels, den aktuellen Stand der Forschung. Der Forschungsstand wird präsentiert, analysiert und evaluiert. Schließlich werden daraus resultierende Lücken, die auch die Motivation für die eigenen Beiträge liefern, identifiziert.

3.1 Anforderungsanalyse

Die Anforderungen an das Framework zur Erfüllung der Automatisierung wissenschaftlicher Workflows wurden vorrangig aus den zu bearbeitenden Problemstellungen (P1-P4) sowie den Beiträgen (B1-B3) der vorliegenden Arbeit abgeleitet. Am Ende des Abschnitts werden die Anforderungen in einen Anforderungskatalog überführt.

3.1.1 Beschreibung der Anforderungen

Im Hinblick auf den Entwurf zur Co-Simulation und Automatisierung wissenschaftlicher Workflows wurden neun konkrete Anforderungen (A1 bis A9) formuliert. Diese Anforderungen lassen sich in Beziehung zu den Beiträgen dieser Arbeit und den zugrundeliegenden Problemstellungen setzen.

A1: Hohe Konfigurierbarkeit

Ein wissenschaftlicher Workflow besteht aus heterogenen Softwareapplikationen, z.B. in einem Energiesystem: Simulatoren, Prognosewerkzeuge, Datenverarbeitungstools, Optimierer, Datenvisualisierung, Zwischenspeicher usw. Das Framework soll Schnittstellen (z.B. grafische Oberfläche) zur Konfigurierbarkeit bereitstellen, um Softwareapplikationen zum Aufbau eines wissenschaftlichen Workflows konfigurieren und verknüpfen zu können, ohne dass die Konfigurationen dafür in einer Programmiersprache fest codiert und angelegt werden müssen.

A2: Einfache Integration

Verschiedene Applikationen und (Mess)Datenquellen aus technischen Komponenten/Reallaboren können direkt zur Weiterverarbeitung in Workflows im Framework integriert werden, ohne dass die Applikationen selbst dafür erweitert werden müssen.

A3: Performanter Datenaustausch

Zwischenergebnisse können zwischen Applikationen, Datenquellen und angebundenen Anlagen performant ausgetauscht werden.

A4: Unterstützung zum Ausführen großer und komplexer Workflows

Sehr große und komplexe wissenschaftliche Workflows und Simulationen mit einer Vielzahl von Komponentenmodellen können mithilfe des Frameworks auf einer leistungsfähigen Berechnungsinfrastruktur (Computing-Cluster) hochperformant ausgeführt werden.

A5: Automatisierung

Die Ausführung integrierter Softwareapplikationen wird vollautomatisch gemanagt. Das Framework kümmert sich um die Laufzeitinfrastrukturen sowie Laufzeitbibliotheken für die Applikationen.

A6: Plattformunabhängige grafische Oberfläche

Das Framework bietet plattformunabhängige grafische Benutzerschnittstellen (z.B. über Web oder Apps) zur Definition und Steuerung von Workflows: Webanwendungen oder browserbasierte Apps können via Browser plattform- und betriebssystemunabhängig genutzt werden.

A7: Wiederverwendbarkeit

Die integrierten Applikationen werden als Framework-Prozesse in einer Bibliothek abgespeichert, um sie in anderen Simulationskontexten wiederverwenden zu können. Darüber hinaus können aufgebaute Workflows zur Verwendung für andere Szenarien gespeichert werden.

A8: Skalierung und Parallelisierung

Workflows sind mithilfe des Frameworks leicht zu skalieren und parallelisieren, um die Performance zu steigern.

A9: Generische Funktionalitäten

Das Framework bietet Hilfsbausteine für generische Funktionalitäten zur Datentransformation und zur Koordinierung von Simulationen (Primary-Baustein für Primary-Secondary-Simulationen, Ereignis-orientierte Koordinierung über Message-Bus usw.).

3.1.2 Anforderungskatalog

Der folgende Anforderungskatalog fasst die neun genannten Anforderungen (siehe Abschnitt 3.1.1) zur Bewertung des relevanten Frameworks und der eigenen Beiträge (B1-B3) in der vorliegenden Arbeit in der Tabelle 3.1 zusammen.

Tab. 3.1.: Anforderungskatalog für die Bewertung relevanter Arbeiten sowie der eigenen Beiträge

Nummer	Bezeichnung
A1	Hohe Konfigurierbarkeit
A2	Einfache Integration
A3	Performanter Datenaustausch
A4	Unterstützung zum Ausführen großer und komplexer Workflows
A5	Automatisierung
A6	Plattformunabhängige grafische Oberfläche
A7	Wiederverwendbarkeit
A8	Skalierung und Parallelisierung
A9	Generische Funktionalitäten

3.2 Stand der Forschung

Im Folgenden werden relevante Forschungsarbeiten, die einen ähnlichen Ansatz verfolgen oder Schwerpunkte auf einzelne Anforderungen des Anforderungskatalogs in Abschnitt 3.1.2 setzen, zur Übersicht vorgestellt und analysiert. Anschließend werden die Arbeiten anhand der verschiedenen Anforderungen, die im vorherigen Abschnitt definiert wurden, bewertet.

3.2.1 Kim et al.: CoSimulating Communication Networks and Electrical System for Performance Evaluation in Smart Grid

In [KPK13; Kim+18] wird eine Co-Simulationsarbeit namens *OOCoSim*, die aus *OPNET* (Netzwerksimulationstool) [Sad+15] und *OpenDSS* (Energiesystem Simulationstool) [Sar14] besteht, beschrieben.

Abb. 3.1 gibt einen Überblick über die Architektur von *OOCoSim*. Das Anwendungsmodell definiert die Steuerungsrichtlinien, Algorithmen und Funktionen, die aktuelle Zustände des Energiesystems. Die Übertragung der Ereignissignale und Steuerinformationen kann durch das *OPNET* simuliert werden. Wenn das *OPNET* aufgerufen wird, werden die *OPNET-Informationen* durch das *Anwendungsmodell* im Kontext der Co-Simulation abgerufen. In ähnlicher Weise werden Steuerungsrichtlinien auf

die *Energiesystem-Simulation* angewendet, und die neuen Zustandsinformationen werden an das *Anwendungsmodell* gemeldet.

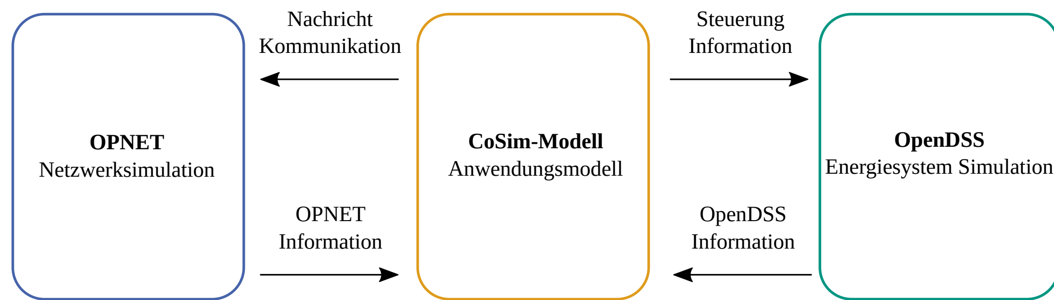

Abb. 3.1.: Architektur von OOCoSim [Kim+18]

Bewertung

A1 Die Anforderung wird nicht erfüllt. Um das Energiesystem zu konfigurieren, müssen alle Parameter in einem *DSS-Skript* dafür definiert werden. *OOCoSim* unterstützt keine Oberfläche zur leichten Konfigurierbarkeit, um Co-Simulationen oder Workflows aufzubauen.

A2 Die Anforderung wird nicht erfüllt. Das Energiesystem wird nur in *OpenSDD* simuliert und in einer Co-Simulation integriert. Deshalb ist es unmöglich, verschiedener Arten von Modellen oder Simulatoren in *OOCoSim* zu integrieren.

A3 Die Anforderung wird vollständig erfüllt. Das *CoSim-Modell* verwaltet die Operationen beider Simulatoren mithilfe des globalen Schedulers und tauscht Daten zwischen *OPNET* und *OpenDSS* aus. Das heißt, das *CoSim-Modell* ist als eine Middleware zur Steuerung, Synchronisation und zum performanten Datenaustausch eingesetzt.

A4 Die Anforderung wird nicht erfüllt. *OOCoSim* kann ausschließlich *OPNET* und *OpenDSS* als Simulatoren zum Aufbau einer Co-Simulation koppeln. Die Ausführung großer und komplizierte Workflows, wird also von *OOCoSim* nicht unterstützt.

A5 Die Anforderung wird nicht erfüllt, denn in *OOCoSim* wird keine Automatisierung von Simulatoren erwähnt.

A6 Die Anforderung wird nicht erfüllt, weil *OOCoSim* keine grafische Oberfläche bereitstellt.

A7 Die Anforderung wird teilweise erfüllt. Die *OpenDSS-Simulation* ist wiederverwendbar für verschiedene Szenarien auf einer Plattform. Aber sie ist schwer auf anderen Plattformen umzusetzen.

A8 Die Anforderung wird teilweise erfüllt. Mehrere *OpenDSS-Simulationen* können parallel ausgeführt werden und mithilfe des Schedulers mit dem *CoSim-Modell* kommunizieren. Jedoch wird nicht erwähnt, wie die *OpenDSS-Simulationen* zur Parallelisierung konfiguriert und koordiniert werden. Daher wird diese Anforderung nur teilweise erfüllt.

A9 Die Anforderung wird teilweise erfüllt. Co-Simulationen werden in *OOCoSim* nur ereignisbasiert gesteuert und synchronisiert. Es werden keine weiteren Funktionalitäten, wie z.B. Datenverarbeitung oder Koordination (Primary-Secondary) vorgestellt.

3.2.2 Huang et al.: Open-source framework for power system transmission and distribution dynamics co-simulation

Ein Open-Source „***F**ramework for **N**etwork **C**o-**S**imulation (FNCS)*“ wird für die Co-Simulation von Energiesystemen in [Hua+17] beschrieben. *FNCS* ist nicht nur eine Middleware-Schnittstelle, sondern auch ein Framework zur Interaktion und Synchronisierung der Übertragungs- und Verteilungssimulatoren. Die Funktionen von *FNCS* umfassen:

- Die Bibliotheken unterstützt `C++`-, Java-, Matlab-, Python- und FORTRAN-Schnittstellen für eine flexible Integration von Simulatoren.
- *FNCS* verwendet einen dezentralen Ansatz, um vorhandene Stromnetz- und Netzwerksimulatoren zu integrieren, und die Anzahl der teilnehmenden Simulatoren ist unbegrenzt [Cir+14] [Han+17].
- Eine definierte API abstrahiert die Low-Level-Interaktion für Nachrichtenkommunikation und Zeitsynchronisierung und lässt sich leicht in die bestehenden dynamischen Übertragungs- und Verteilungssimulatoren integrieren.

Abb. 3.2 zeigt die Architektur von *FNCS* mit dem Datenfluss zwischen den Simulatoren und dem Framework. Die Kommunikation zwischen dem *Broker* und dem *Simulator* wird durch die *Inter-Simulation-Communicator-Komponente* gehandhabt. Diese Komponente wird innerhalb der Simulatorprozesse ausgeführt. Der *Broker* und die *Communicator-Komponente* werden über die ZeroMQ-Bibliothek implementiert. Die *Communication-Management-Komponente* stellt Schnittstellen zum Senden und Empfangen von Nachrichten bereit. Die *Time-Management-Komponente* von *FNCS* bietet die Schnittstellen für einen Simulator, um seine Zeit mit anderen zu synchronisieren.

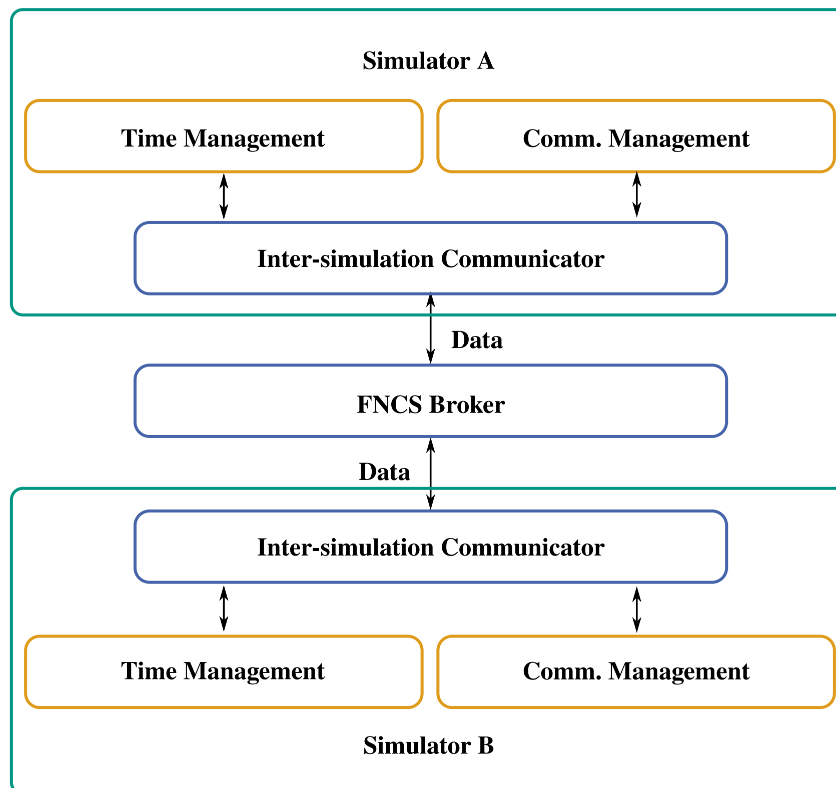

Abb. 3.2.: Architektur von *FNCS* [Hua+17]

Bewertung

A1 Die Anforderung wird teilweise erfüllt. Durch die Verwendung und Erweiterung der *FNCS*-Schnittstellen sind Co-Simulationen leicht zu konfigurieren. *FNCS* unterstützt jedoch keine grafische Oberfläche dafür.

A2 Die Anforderung wird nicht erfüllt. Die Bibliotheken unterstützt die Schnittstellen für verschiedene Programmiersprachen zur flexiblen Integration von Simulatoren. Aber ohne Programmierkenntnisse ist es schwierig, Applikationen in *FNCS* zu integrieren.

A3 Die Anforderung wird vollständig erfüllt. Mithilfe der *Broker-Komponente*, die auf der ZeroMQ-Bibliothek basiert, ist der Datenaustausch zwischen Simulatoren wirkungsvoll.

A4 Die Anforderung wird vollständig erfüllt, da *FNCS* keine Begrenzung der Anzahl der teilnehmenden Simulatoren hat.

A5 Die Anforderung wird nicht erfüllt. In *FNCS* wird keine Automatisierung von Simulatoren erwähnt.

A6 Die Anforderung wird nicht erfüllt. *FNCS* unterstützt keine grafische Oberfläche.

A7 Die Anforderung wird teilweise erfüllt. Die integrierten und geprüften Modelle sind wiederverwendbar in *FNCS*, aber schwer umzusetzen.

A8 Die Anforderung wird nicht erfüllt. In *FNCS* wird Parallelismus der Simulationen nicht erwähnt.

A9 Die Anforderung wird teilweise erfüllt. Nur die zeitbasierte Synchronisation wird in *FNCS* bereitgestellt. Keine weitere Funktionalität, z.B. zur Datenverarbeitung oder Koordination (ereignisbasiert), wird vorgestellt.

3.2.3 Palmintier et al.: Design of the HELICS high-performance transmission-distribution-communication-market co-simulation framework

[Pal+17b] stellt den Entwurf für „the hierarchical engine for large-scale Infrastructure co-simulation“ (*HELICS*) vor, ein neues Co-Simulations-Framework für elektrische Energiesysteme. Durch die Verwendung von *HELICS* können groß angelegte Co-Simulationen mit Energiesystem-, Kommunikations-, Markt- und Endverbraucher-Tools erstellt werden.

Wie in Abb. 3.3 gezeigt, verwendet *HELICS* eine mehrschichtige Architektur:

- Die *Plattform*-Schicht stellt sicher, dass *HELICS* auf allen Betriebssystemen (Windows, Linux und Macs) und auch auf **H**igh **P**erformance **C**omputing (HPC)-Clustern funktionieren kann.
- Die *Kommunikation*-Schicht bietet Zeitmanagement und Datenaustausch für kombinierte diskrete Ereignissimulationen und Zeitreihensimulationen.
- Die *Applikation*-Schicht stellt die Schnittstellen zwischen Simulatoren und Kommunikationsfunktionen bereit.
- Die *Simulatoren*-Schicht definiert zwei Klassen zur Integration verschiedener Simulatoren, z.B. FMI [Ass22a], GridDyn [Kel+15], CYMDIST [TMA12], ns-3 [RH10], FESTIV [EO12].
- Die *Schnittstelle*-Schicht unterstützt die Operationen zum Aufbau und zur Ausführung der Simulationen.

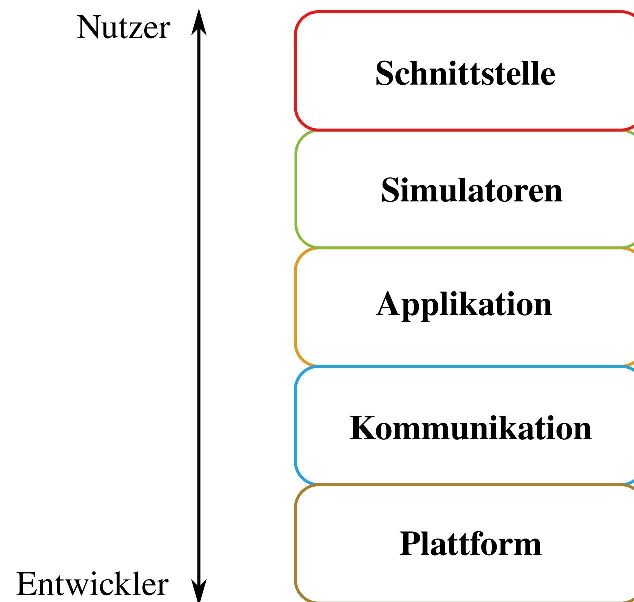

Abb. 3.3.: Architektur von *HELICS* [Pal+17b]

Bewertung

A1 Die Anforderung wird nicht erfüllt, da eine JSON-Datei von Nutzern zur Konfiguration einer Simulation angelegt werden muss.

A2 Die Anforderung wird nicht erfüllt, denn Benutzer müssen Programmierschnittstellen implementieren, um ihre Simulatoren mit *HELICS* zu kombinieren. Beispielsweise kann *HELICS* durch die Verwendung der Matlab-API mit Matlab-Modellen interagieren. Jedoch ist eine kommerzielle Matlab-Lizenz zur Ausführung der Modelle in der Matlab-Umgebung notwendig.

A3 Die Anforderung wird vollständig erfüllt, denn in *HELICS* wird *ZeroMQ* als Kommunikationsinfrastruktur zum effizienten Datenaustausch eingesetzt.

A4 Die Anforderung wird vollständig erfüllt, denn *HELICS* unterstützt Co-Simulationen im sehr großen Maßstab (100000+ Simulatoren).

A5 Die Anforderung wird vollständig erfüllt, denn in *HELICS* werden Simulationen automatisiert ausgeführt.

A6 Die Anforderung wird nicht erfüllt. Eine Oberfläche von *HELICS* kann von Nutzern installiert werden. Jedoch werden keine grafischen symbolischen Simulatoren bereitgestellt, die miteinander gekoppelt werden können, um Co-Simulationen oder Workflows zu erstellen. Der Aufbau einer Co-Simulation kann nur mit einer Konfigurationsdatei, die von Nutzern geschrieben wird, realisiert werden.

A7 Die Anforderung wird teilweise erfüllt. Die integrierten und geprüften Modelle sind wiederverwendbar in *HELICS*, aber schwer umzusetzen.

A8 Die Anforderung wird nicht erfüllt. In *HELICS* können verschiedene Simulatoren parallel ausgeführt werden. Jedoch wird Daten-Parallelismus (ein einzelner Simulator wird durch Parallelisierung auf viele Daten gleichzeitig angewendet) nicht erwähnt.

A9 Die Anforderung wird teilweise erfüllt, denn die zeit- und ereignisbasierte sowie hybride Synchronisation wird in *HELICS* bereitgestellt. Aber keine weitere Funktionalität, z.B. zur Datenverarbeitung, wird vorgestellt.

3.2.4 Köster et al.: Snakemake-a scalable bioinformatics workflow engine

Snakemake [KR12] ist ein Workflow-Management-System, welches auf Python aufbaut und vor allem in der Bioinformatik eingesetzt wird. Es wird gut geeignet benutzt, um reproduzierbare und skalierbare Datenanalysen zu erstellen. Es stellt eine lesbare Python-basierte Workflow-Definitionssprache und eine leistungsstarke Ausführungsumgebung bereit. Workflows können nahtlos auf Server-, Cluster-, Grid- und Cloud-Umgebungen skaliert werden, ohne dass die Workflow-Definition geändert werden muss. Schließlich können *Snakemake*-Workflows eine Beschreibung der erforderlichen Software beinhalten, die automatisch in jeder Ausführungsumgebung bereitgestellt wird.

Es funktioniert nach folgenden Prinzipien [BDL21]:

- Jeder Schritt im Ablauf ist eine *rule* (Regel).
- Die Eingaben, Ausgaben und die Kommandozeilenbefehle für jeden Schritt sind in einem sogenannten *Snakefile* (s. Auflistung 3.1) beschrieben.
- Jede Regel verfügt über eine Anzahl Parameter, die in einer *yaml*-Datei (s. Auflistung 3.2) beschrieben sind.
- Jede Regel erzeugt eine Ausgabe, die wiederum die Eingabe für andere Regeln darstellen kann.

```
1 rule NAME:
2     input: "path/to/inputfile", "path/to/other/inputfile"
3     output: "path/to/outputfile", "path/to/another/outputfile"
4     shell: "somecommand {input} {output}"
```

Auflistung 3.1: Beispiel von Snakefile

```
samples:
    Sample1: fastq_files/1.fastq
    Sample2: fastq_files/2.fastq
```

Auflistung 3.2: Beispiel von config.yaml

Bewertung

A1 Die Anforderung wird teilweise erfüllt. Obwohl die Workflow-Definitionssprache von *Snakemake* auf eine hohe Lesbarkeit ausgelegt ist, was für Transparenz und Anpassbarkeit entscheidend ist, sind Workflows nicht einfach zu konfigurieren. Denn Nutzer brauchen immer noch Zeit zu lernen, wie die Sprache formuliert und wie das Format eines *Snakefiles* definiert wird. Das manuelle Entwerfen von *Snakefiles* ist fehleranfällig.

A2 Die Anforderung wird nicht erfüllt. Die Verwaltung von Software-Stacks, die für einzelne Regeln benötigt werden, ist über zwei komplementäre Mechanismen (*Conda* [Ana17] und *Container*) direkt in *Snakemake* selbst integriert [Möl+21]. Dazu muss die Installation von *Conda* und *Singularity-Container*[Shu+22] zuerst durchgeführt werden. Anschließend wird z.B. eine *environment.yml* zur Definition der benötigten Software von Nutzern angelegt. Danach wird sie zur Installation der Software direkt in der *Conda-Umgebung* oder in einem *Singularity-Container* mit der *Conda-Umgebung* ausgeführt.

A3 Die Anforderung wird teilweise erfüllt. Der Datenaustausch zwischen Simulatoren ist in *Snakemake* durch das Dateisystem realisiert. Wenn ein Job für das Caching geeignet ist, können nachfolgende Jobs im Workflow (mit denselben Abhängigkeiten) die Ausgabedateien direkt aus dem Cache entnehmen [Möl+21].

A4 Die Anforderung wird vollständig erfüllt. *Snakemake* hat keine Beschränkung für die Anzahl der teilnehmenden Jobs/Simulatoren.

A5 Die Anforderung wird vollständig erfüllt. Mithilfe von *Snakefile* kann ein Workflow als ein *Snakemake*-Objekt automatisiert ausgeführt werden.

A6 Die Anforderung wird nicht erfüllt, denn *Snakemake* ist in erster Linie ein Befehlszeilentool und unterstützt daher keine grafische Oberfläche.

A7 Die Anforderung wird teilweise erfüllt. Jedes *Snakefile* ist wiederverwendbar, aber nur innerhalb der *Snakemake*-Umgebung.

A8 Die Anforderung wird teilweise erfüllt. In der *Snakefile* kann von Nutzern fest codiert werden, welche Teile des Workflows parallel ausgeführt werden.

A9 Die Anforderung wird teilweise erfüllt. Nur ein *Job-Scheduling* wird in *Snakemake* zur Job-Steuerung bereitgestellt. Keine weitere Funktionalität, z.B. zur Datenerfassung, -verarbeitung oder Koordination (Primary-Secondary, ereignisbasiert), wird vorgestellt.

3.2.5 Liberatore et al.: Smart grid communication and co-simulation

In [LA11] wird das System „*PowerNet*“ zur Kommunikation in Smard Grids vorgestellt. In PowerNet werden *ns-2* und *Modelica* als Co-Simulation ausgeführt. Wie in Abb. 3.4 dargestellt werden *ns-2* und *Modelica* direkt gekoppelt, um Interaktion dazwischen zu realisieren. Die Simulation von *ns-2* übernimmt den Teil des Kommunikationsnetzes. In *Modelica* wird das Energiesystem simuliert. Die Synchronisation erfolgt durch den Primary-Secondary-Ansatz. Der Primary ist *ns-2* und beinhaltet die Kontrolllogik.

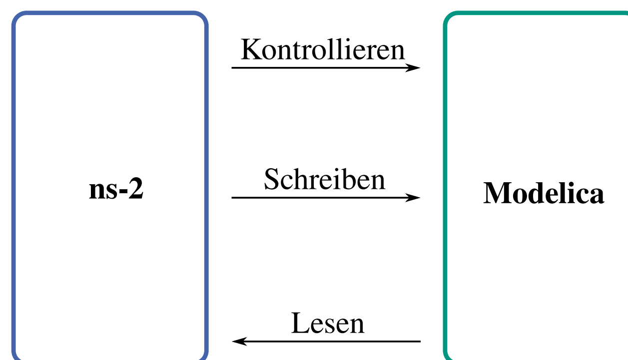

Abb. 3.4.: Die Lese- und Schreiboperationen zwischen *Modelica* und *ns-2* [LA11]

Zur Einordnung von *PowerNet* in die vorgestellten Eigenschaften (Kapitel 2.1) ergeben sich die folgenden Aspekte, wobei eine kompakte Zusammenfassung in Abb. 3.5 dargestellt ist.

- Als Nutzer kommen vor allem IKT-Produzenten und Netzkomponentenhersteller infrage. Diese haben vor allem ein Interesse daran, das Zusammenspiel zwischen ihren Produkten und dem Rest des Smart Grid zu analysieren. Dafür ist eine relativ einfache Simulation wie *PowerNet* schon ausreichend. Ebenso gilt dies für den Bildungsbereich, um einen Einblick in die Funktionsweise eines Smart Grid zu bekommen.
- Bei den zeitlichen Aspekten ist *PowerNet* auf eine Simulation schneller als in Echtzeit im Gleichgewichtszustand beschränkt. Die Kommunikationsnetzsimulation ns-2 bietet Echtzeitausführung an, jedoch kann es dabei zu Problemen kommen [MI04].

- Die Synchronisation erfolgt durch einen Primary-Secondary-Ansatz. Die Kommunikationsreihenfolge wird dabei nicht iterativ und seriell durch *ns-2* durchgeführt.
- *PowerNet* bietet keine Hardware-Unterstützung und simuliert nur diskrete und kontinuierliche Prozesse.
- Grundsätzlich werden nur White-Box-Modelle unterstützt, da man exaktes Wissen über interne Modellstrukturen benötigt.
- Die Simulationen werden über eine API angesprochen und unterstützen keine der vorgestellten Standards.

<table>
<tr><td>Nutzer</td><td colspan="2">Netzbetreiber</td><td colspan="2">IKT-Produzent</td><td colspan="2">Netzkomp.-hersteller</td><td colspan="2">VKW Betreiber & E.-Vers.</td><td colspan="2">Politik & Regulation</td><td colspan="2">Bildungsbereich</td></tr>
<tr><td>Zeitliches Verhältnis</td><td colspan="4">Langsamer als Echtzeit</td><td colspan="4">Echtzeit</td><td colspan="4">Schneller als Echtzeit</td></tr>
<tr><td>Zeitliche Auflösung</td><td colspan="4">Gleichgewichtszustand</td><td colspan="4">Elektro-mechanisch</td><td colspan="4">Elektro-magnetisch</td></tr>
<tr><td rowspan="2">Kommunikationsreihenfolge</td><td colspan="6">Nicht iterativ</td><td colspan="6">Iterativ</td></tr>
<tr><td colspan="3">Parallel</td><td colspan="3">Seriell</td><td colspan="3">Parallel</td><td colspan="3">Seriell</td></tr>
<tr><td>Synchronisation</td><td colspan="4">Feste Zeitpunkte</td><td colspan="4">Ereignisgetrieben</td><td colspan="4">Master-Slave</td></tr>
<tr><td>Simulationstyp</td><td colspan="4">Co-Simulation</td><td colspan="4">Hardware-Unterstützung</td><td colspan="4">HiL</td></tr>
<tr><td>Elemente der Modellierung</td><td colspan="3">Kontinuierliche Prozesse</td><td colspan="3">Diskrete Prozesse</td><td colspan="3">Rollen</td><td colspan="3">Statistische Elemente</td></tr>
<tr><td>Zugänglichkeit</td><td colspan="4">Black Box</td><td colspan="4">White Box</td><td colspan="4">Grey Box</td></tr>
<tr><td>Nutzbarkeit</td><td colspan="6">API</td><td colspan="6">GUI</td></tr>
<tr><td>Standards</td><td colspan="3">IEC61850</td><td colspan="3">CIM</td><td colspan="3">OPC UA</td><td colspan="3">FMI</td></tr>
</table>

Abb. 3.5.: Einordnung von PowerNet

Bewertung

A1 Die Anforderung wird nicht erfüllt, denn *PowerNet* unterstützt keine Konfigurierbarkeit zum Aufbau von Co-Simulationen oder Workflows.

A2 Die Anforderung wird nicht erfüllt. Das Energiesystem wird nur in *Modelica* simuliert und in einer Co-Simulation integriert. Daher wird keine Möglichkeit für die Integration vieler verschiedener Arten von Modellen oder Simulatoren bereitgestellt.

A3 Die Anforderung wird nicht erfüllt. Bei der direkten Kopplung von *ns-2* und *Modelica* zur Ausführung der Co-Simulation interagieren die Simulatoren

direkt miteinander. Das heißt, es gibt keine Middleware zur Steuerung, Synchronisation oder zum Datenaustausch.

A4 Die Anforderung wird nicht erfüllt, denn *PowerNet* koppelt nur zwei Simulatoren und ist auf bestimmte Szenarien beschränkt.

A5 Die Anforderung wird nicht erfüllt, da keine Automatisierung von Simulatoren in *PowerNet* erwähnt wird.

A6 Die Anforderung wird nicht erfüllt, denn *PowerNet* unterstützt keine grafische Oberfläche.

A7 Die Anforderung wird nicht erfüllt, denn durch die fehlende Middleware ergibt sich eine schlechte Wiederverwendbarkeit und Erweiterbarkeit.

A8 Die Anforderung wird nicht erfüllt. Die Architektur ermöglicht keinen Parallelismus und keine Verteilung von Simulationen.

A9 Die Anforderung wird teilweise erfüllt. Nur der Primary-Secondary-Ansatz wird in *PowerNet* zur Synchronisation bereitgestellt. Keine weitere Funktionalität, z.B. zur Datenverarbeitung oder Koordination, wird vorgestellt.

3.2.6 Schütte et al.: Mosaik: A framework for modular simulation of active components in Smart Grids

Bei *Mosaik* handelt es sich um ein modulares Smart Grid Co-Simulations-Framework zur Erstellung gemeinsamer Smart Grid Szenarien. *Mosaik* bietet eine Simulationsschnittstelle zur Integration verschiedener Simulationen, die auf unterschiedlichen Plattformen laufen können. Zur Beschreibung der Modelle gibt es eine formale, semantische Beschreibungssprache [SST11].

Mosaik löst das Problem der Komposition der Simulationen auf verschiedenen Ebenen [SST11; SSS12]:

- Syntaktische Ebene: Auf der untersten Ebene, der syntaktischen Ebene, werden die Interaktionen mit den Simulationsmodellen festgelegt. *Mosaik* beschränkt sich hier auf Entitäten und Attribute, die zu bestimmten Zeitpunkten ausgetauscht werden. Der Austausch erfolgt über eine Schnittstelle (*Component-API*), die die Simulation implementiert.
- Semantische Ebene: Zur semantischen Beschreibung der Simulationen und Modelle wird eine domänenspezifische Sprache eingesetzt. Dies ermöglicht den automatischen Transfer von Daten von einem zum anderen Modell.

- Szenarioebene: Bei Mosaik werden die Szenariodefinitionen außerhalb der Simulationsmodelle platziert. Dazu kommt zur Definition der Szenarien eine domänenspezifische Sprache zum Einsatz, die sich der *Scenario-API* bedient.
- Kontrollebene: Auf Kontrollebene können verschiedene Kontrollstrategien basierend auf domänenspezifischen Standards eingebunden werden.

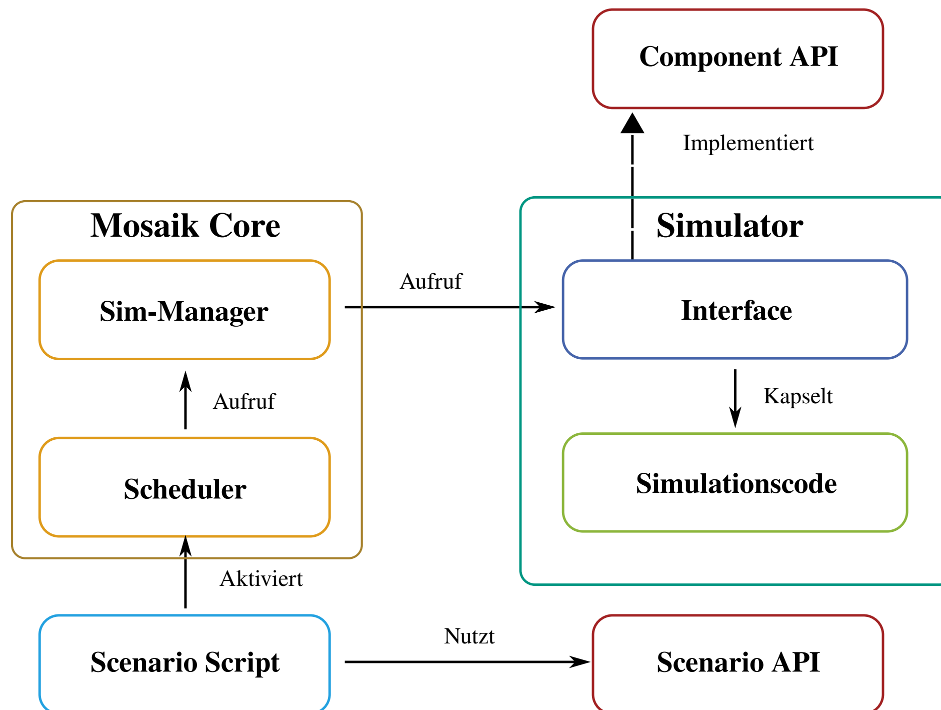

Abb. 3.6.: Architektur von Mosaik [Ste+18]

Die Architektur von *Mosaik* wird in Abb. 3.6 gezeigt. Zentrale Instanz zur Steuerung und Synchronisation bei *Mosaik* ist der *Mosaik-Core*. Dieser besteht im Wesentlichen aus dem *Sim-Manager* und *Scheduler*. Der *Sim-Manager* ruft über eine Schnittstelle die einzelnen gekapselten Simulationen auf. Dazu implementieren die Simulationen die *Component-API* und werden über TCP-Verbindungen angesprochen. Gleichzeitig stellen die Simulationen eine Metabeschreibung bereit. Der *Scheduler* koordiniert den Austausch von Daten zwischen den Simulationen basierend auf einer gemeinsamen Simulationsuhr. Zum Aufbau einer Co-Simulation wird ein *Scenario-Script* benötigt, das die *Scenario-API* nutzt. Das Skript spezifiziert die verwendeten Simulationen, Anzahl der Modellentitäten und wie diese verbunden sind. Mosaik ordnet sich in die vorgestellten Eigenschaften (Kapitel 2.1) wie folgt ein (siehe Abb. 3.7):

- Als Nutzer kommen potenziell alle vorgestellten Nutzergruppen (s. Abschnitt 2.1) infrage. *Mosaik* ermöglicht es, mit wenig Aufwand komplexe Szenarien aufzubauen.

- Bei den zeitlichen Aspekten ist *Mosaik* hinsichtlich des zeitlichen Verhältnisses sehr flexibel und bei der zeitlichen Auflösung auf den Gleichgewichtszustand beschränkt.
- Der *Scheduler* ermöglicht nur eine nicht-iterative Synchronisationsreihenfolge in paralleler und serieller Form. Die Synchronisation erfolgt durch einen Primary-Secondary-Ansatz und einen ereignisgesteuerten Ansatz, der mit der API *SimPy* implementiert werden kann.
- *Mosaik* bietet keine Hardware-Unterstützung.
- *Mosaik* ist auf die Simulation von Smart Grids spezialisiert. Es können nahezu alle Elemente eines Smart Grid modelliert werden. Gleichzeitig werden domänenspezifische Standards zum leichteren Aufbau von Szenarien unterstützt.
- Bei der Zugänglichkeit der Modelle macht *Mosaik* keine Beschränkungen.
- *Mosaik* hat eine Menge von APIs zur Kommunikation und Steuerung, wovon die wichtigsten vorgestellt wurden. Eine GUI wird zwar angeboten, diese beschränkt sich aber auf rudimentäre Funktionen.

<table>
<tr><td>Nutzer</td><td colspan="2">Netzbetreiber</td><td colspan="2">IKT-Produzent</td><td colspan="2">Netzkomp.-hersteller</td><td colspan="2">VKW Betreiber & E.-Vers.</td><td colspan="2">Politik & Regulation</td><td colspan="2">Bildungsbereich</td></tr>
<tr><td>Zeitliches Verhältnis</td><td colspan="4">Langsamer als Echtzeit</td><td colspan="4">Echtzeit</td><td colspan="4">Schneller als Echtzeit</td></tr>
<tr><td>Zeitliche Auflösung</td><td colspan="4">Gleichgewichtszustand</td><td colspan="4">Elektro-mechanisch</td><td colspan="4">Elektro-magnetisch</td></tr>
<tr><td rowspan="2">Kommunikationsreihenfolge</td><td colspan="6">Nicht iterativ</td><td colspan="6">Iterativ</td></tr>
<tr><td colspan="3">Parallel</td><td colspan="3">Seriell</td><td colspan="3">Parallel</td><td colspan="3">Seriell</td></tr>
<tr><td>Synchronisation</td><td colspan="4">Feste Zeitpunkte</td><td colspan="4">Ereignisgetrieben</td><td colspan="4">Master-Slave</td></tr>
<tr><td>Simulationstyp</td><td colspan="4">Co-Simulation</td><td colspan="4">Hardware-Unterstützung</td><td colspan="4">HiL</td></tr>
<tr><td>Elemente der Modellierung</td><td colspan="3">Kontinuierliche Prozesse</td><td colspan="3">Diskrete Prozesse</td><td colspan="3">Rollen</td><td colspan="3">Statistische Elemente</td></tr>
<tr><td>Zugänglichkeit</td><td colspan="4">Black Box</td><td colspan="4">White Box</td><td colspan="4">Grey Box</td></tr>
<tr><td>Nutzbarkeit</td><td colspan="6">API</td><td colspan="6">GUI</td></tr>
<tr><td>Standards</td><td colspan="3">IEC61850</td><td colspan="3">CIM</td><td colspan="3">OPC UA</td><td colspan="3">FMI</td></tr>
</table>

Abb. 3.7.: Einordnung von *Mosaik*

Bewertung

A1 Die Anforderung wird teilweise erfüllt. *Mosaik* bietet APIs zur Kommunikation zwischen Komponenten, z.B. *Component-API* und *Scenairo-API* sowie *Scenario-Script*. Jedoch müssen solche APIs zum Aufbau verschiedener Simulationsszenarien immer neu erweitert werden.

A2 Die Anforderung wird nicht erfüllt. Um ein Simulationsmodell in das *Mosaik-Ökosystem* zu integrieren, muss *Mosaiks SimAPI* Schritt für Schritt implementiert werden, z.B. die Metadaten eines Modells inkl. Parameter und Attribute zu definieren, die *Simulator-Klasse* inkl. der Methoden *init()*, *create()*, *step()* usw. zu implementieren. D.h. ohne Programmierkenntnisse ist es nur sehr eingeschränkt möglich, Applikationen in *Mosaik* zu integrieren.

A3 Die Anforderung wird vollständig erfüllt. *ZeroMQ* ist eine Nachrichtenaustauschbibliothek für High-throughput computing, die speziell auf verteilte Systeme oder gleichzeitige Ausführung in verschiedenen Systemen entwickelt wurde [Gou+17; OFF22b]. Mithilfe von *ZeroMQ* ist der Datenaustausch in *Mosaik* performant realisierbar.

A4 Die Anforderung wird nicht erfüllt. Da keine Schnittstelle zur Operation des Computing-Clusters in *Mosaik* bereitgestellt wird, ist die Anzahl der Komponentenmodelle einer Simulation beschränkt. Wie die Autoren von [Ste+18] ausgeführt haben, ist *Mosaik* eine gute Wahl für prototypische Co-Simulationen auf Einstiegsniveau, nicht aber für komplexe Studien.

A5 Die Anforderung wird vollständig erfüllt. Nach der Integration einer Applikation in das Mosaik-Ökosystem ist die Applikation als ein Simulator in *Mosaik* automatisiert auszuführen.

A6 Die Anforderung wird nicht erfüllt. *Mosaik* hat keine einfach zu bedienende Web-Benutzeroberfläche, sondern eine Desktop-Benutzeroberfläche [OFF22a], die Benutzern nur eine Ansicht von Simulationsergebnissen, aber keine Diagramme von Datenrouting, Transformationen und Systemvermittlungslogik bietet.

A7 Die Anforderung wird teilweise erfüllt. Die in *Mosaik* integrierten Applikationen können als Simulatoren für verschiedene Simulationsszenarien wiederverwendet werden. Aber *Mosaik* bietet keine Containerisierungsschnittstelle zum Einkapseln der Simulatoren. Dies kann dazu führen, dass die Umsetzung einer Simulation fehlerbehaftet und zeitaufwändig ist, da der gesamte Sourcecode dabei immer mitkopiert und transplantiert werden muss.

A8 Die Anforderung wird teilweise erfüllt. Das *Master Control Programm* verwaltet die Zusammenstellung der Simulationsszenarien und steuert die Ausführung der Simulationen in *Mosaik*. Damit kann eine Experiment-Instanz für jeden Simulationslauf in einem eigenen Thread laufen, sodass mehrere Experimente parallel ausgeführt werden können. Der *Simulatormanager* (oder einfach *Sim-Manager*) ist für das Starten und Steuern sowie die Parallelisierung der an einer Simulation beteiligten externen Simulatorprozesse sowie für die Kommunikation mit ihnen verantwortlich. Zur Realisierung der Parallelisierung der internen oder externen Simulatoren muss die Konfiguration jedoch von den Nutzern für das *Master Control Programm* oder den *Sim-Manager* fest codiert werden.

A9 Die Anforderung wird teilweise erfüllt. *Mosaik* bietet APIs zur Koordinierung von Simulationen, z.B. Primary-Secondary und ereignisbasierte Simulationen. Vor der Benutzung müssen die APIs von Nutzern jedoch szenariospezifisch erweitert werden, z.B. für eine ereignisorientierte Koordinierung ist *SimPy* zu implementieren. Außerdem wird keine Funktionalität zur Datentransformation in *Mosaik* bereitgestellt.

3.2.7 Dahmann et al.: High Level Architecture (HLA)

Bei der **H**igh **L**evel **A**rchitecture (*HLA*) handelt es sich um eine Spezifikation einer technischen Architektur zur Zusammenarbeit verteilter Simulationen. Sie wird im IEEE 1516 definiert. Die *HLA* ist auf verteilte Simulationen spezialisiert. Insbesondere wird die Wiederverwendung von Simulationen unterstützt. Es wird keine spezifische Implementierung vorgeschrieben [DKW98; DM98].

Bei der *HLA* wird die Co-Simulation als Ganzes als *Federation* und die einzelnen Simulationen als Teilnehmer dieser als *Federate* bezeichnet. Es gibt drei große Definitionsbereiche:

- *Rules*: Es gibt zehn Designregeln, die beschreiben, wie die Zuständigkeiten und Beziehungen zwischen den Federates und der Federation aussehen. Die Regeln 1-5 beschreiben die Federates, 6-10 die Federation.
- *Interface Specification*: Die Interface Specification schreibt die Kommunikation zwischen den Federates und der ***R**un**t**ime **I**nfrastructure (RTI)* vor. Die *RTI* ist der zentrale Punkt der Kommunikation. Sie bietet sechs verschiedene Serviceklassen, die unter anderem Synchronisation und Datenaustausch regeln.

- ***O**bject **M**odel **T**emplate (OMT)*: Das *OMT* hat zwei Typen von Templates zur Beschreibung von *Federates* und der *Federation*. *Federates* werden durch das ***S**imulation **O**bject **M**odel (SOM)* und *Federations* durch das ***F**ederation **O**bject **M**odel (FOM)* beschrieben. *SOM* und *FOM* basieren beide auf dem *OMT* [01].

Die Architektur der *HLA* wird in Abb. 3.8 gezeigt. Die Kommunikation läuft über die zentrale *RTI* ab, wobei die Aufrufe, im Gegensatz zu Mosaik, vom *Federate* ausgehen. Die *RTI* ruft die *Federates* zurück. Die *Federates* können verschiedene Services bei der *RTI* aufrufen und haben im Wesentlichen die Kontrolle über die Synchronisation. Der *Simulationscode* selbst wird durch den *Federate Ambassador* gekapselt. Dieser kommuniziert mit der *RTI* und tauscht über diese Daten mit anderen Simulationen aus. Es werden wesentlich komplexere Funktionsaufrufe als bei Mosaik unterstützt. Die Simulation, sowie deren Attribute und Objekte werden durch das *SOM* beschrieben. Das *SOM* ist wesentlich umfangreicher und detaillierter als die Metabeschreibung von *Mosaik*. Zum Aufbau einer *Federation* wird ein *FOM* verwendet. Dieses spezifiziert den Aufbau und die Interaktion der *Federates*, sowie die Daten, die innerhalb der *Federation* ausgetauscht werden können [Ste+18].

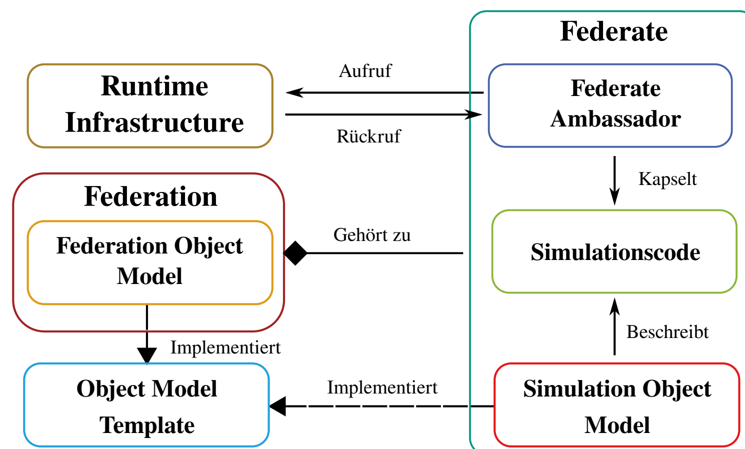

Abb. 3.8.: Architektur von *HLA* [Ste+18]

Die *HLA* ordnet sich bei den vorgestellten wesentlichen Eigenschaften wie folgt ein (siehe Abb. 3.9):

- Als Nutzer der *HLA* kommen generell all diejenigen infrage, die ein hohes Maß an Kontrolle über die Co-Simulation benötigen. Dafür ist die notwendige Implementierung entsprechend aufwändig. Da die *HLA* keine Implementierung vorschreibt oder vorgibt, muss alles von Grund auf implementiert werden. Dies ermöglicht ein großes Maß an Flexibilität. Gleichzeitig wird die *HLA* dadurch für kleinere Unternehmen oder die Bildungsbereiche zum Einstieg unattraktiv. Vorgefertigte Implementierungen der *HLA* für den Bereich Smart Grid erleichtern den Einstieg, wie zum Beispiel [Geo+12; NRS09].

- Bei den zeitlichen Aspekten lässt die *HLA* dem Implementierenden alle Freiheiten. So werden beim zeitlichen Verhältnis und bei der zeitlichen Auflösung alle vorgestellten Varianten theoretisch unterstützt.
- Die Kommunikationsreihenfolge kann im Gegensatz zu *Mosaik* auch iterativ erfolgen, da die einzelnen Simulationen weitgehend autonom über ihren eigenen Ablauf entscheiden können.
- Die Synchronisation wird ebenso durch die Simulationen selbst vorgenommen und somit kann jede der vorgestellten Methoden verwendet werden.
- Die *HLA* bietet Unterstützung für *HiL*. Eine Hardwareunterstützung zur Beschleunigung wird nicht spezifiziert, kann aber implementiert werden.
- Da die *HLA* nicht auf die Simulation von Smart Grids beschränkt ist, kann prinzipiell eine Vielzahl von Szenarien simuliert werden, wie z.B. kontinuierliche und diskrete Prozesse, sowie Rollen und statistische Elemente.
- Drei verschiedene Typen der Zugänglichkeit der Modelle werden alle von *HLA* unterstützt
- Da es sich bei *HLA* nur um eine Spezifikation einer Architektur zum Aufbau verteilter Co-Simulationen handelt, werden keine API oder GUI, sowie keine der vorgestellten Standards unterstützt.

<table>
<tr><td>Nutzer</td><td colspan="2">Netzbetreiber</td><td colspan="2">IKT-Produzent</td><td colspan="2">Netzkomp.-hersteller</td><td colspan="2">VKW Betreiber & E.-Vers.</td><td colspan="2">Politik & Regulation</td><td colspan="2">Bildungsbereich</td></tr>
<tr><td>Zeitliches Verhältnis</td><td colspan="4">Langsamer als Echtzeit</td><td colspan="4">Echtzeit</td><td colspan="4">Schneller als Echtzeit</td></tr>
<tr><td>Zeitliche Auflösung</td><td colspan="4">Gleichgewichtszustand</td><td colspan="4">Elektro-mechanisch</td><td colspan="4">Elektro-magnetisch</td></tr>
<tr><td rowspan="2">Kommunika-tionsreihenfolge</td><td colspan="6">Nicht iterativ</td><td colspan="6">Iterativ</td></tr>
<tr><td colspan="3">Parallel</td><td colspan="3">Seriell</td><td colspan="3">Parallel</td><td colspan="3">Seriell</td></tr>
<tr><td>Synchronisation</td><td colspan="4">Feste Zeitpunkte</td><td colspan="4">Ereignisgetrieben</td><td colspan="4">Master-Slave</td></tr>
<tr><td>Simulationstyp</td><td colspan="4">Co-Simulation</td><td colspan="4">Hardware-Unterstützung</td><td colspan="4">HiL</td></tr>
<tr><td>Elemente der Modellierung</td><td colspan="3">Kontinuierliche Prozesse</td><td colspan="3">Diskrete Prozesse</td><td colspan="3">Rollen</td><td colspan="3">Statistische Elemente</td></tr>
<tr><td>Zugänglichkeit</td><td colspan="4">Black Box</td><td colspan="4">White Box</td><td colspan="4">Grey Box</td></tr>
<tr><td>Nutzbarkeit</td><td colspan="6">API</td><td colspan="6">GUI</td></tr>
<tr><td>Standards</td><td colspan="3">IEC61850</td><td colspan="3">CIM</td><td colspan="3">OPC UA</td><td colspan="3">FMI</td></tr>
</table>

Abb. 3.9.: Einordnung von *HLA*

Bewertung

A1 Die Anforderung wird nicht erfüllt. In der *HLA* wird keine Konfigurierbarkeit definiert.

A2 Die Anforderung wird vollständig erfüllt. Die *HLA* ist nicht für den Aufbau von Energiesystemen konzipiert, sondern es kann prinzipiell alles simuliert werden. Verschiedene Applikationen können theoretisch durch den *Federate Ambassador* gekapselt werden.

A3 Die Anforderung wird vollständig erfüllt. Wie in [MR09; Güt+20] evaluiert wurde, können Daten mithilfe der *RTI* zwischen Simulationen performant ausgetauscht werden.

A4 Die Anforderung wird vollständig erfüllt. Wie die Autoren von [Ste+18] gezeigt haben, ist die *HLA* auf lange Sicht vielseitig und leistungsfähig für komplexe und umfangreiche Studien.

A5 Die Anforderung wird nicht erfüllt. Da die *HLA* nur als Architektur entworfen wird, wird keine Technologie zur Automatisierung der Applikation in der *HLA* erwähnt und eingesetzt.

A6 Die Anforderung wird nicht erfüllt. Denn es handelt sich bei *HLA* nur um eine Spezifikation einer Architektur zum Aufbau verteilter Co-Simulationen, es werden keine GUIs unterstützt.

A7 Die Anforderung wird vollständig erfüllt. Die in der *HLA* integrierten Applikationen können als *Federate* für verschiedene Simulationsszenarien wiederverwendet werden.

A8 Die Anforderung wird vollständig erfüllt. Die Kommunikationsreihenfolge der *HLA* kann entweder nicht iterativ und auch iterativ erfolgen, da die einzelnen Simulationen weitgehend autonom über ihren eigenen Ablauf entscheiden können. Darüber hinaus beschreiben [Son+07; XCW11; Liu+12], dass die *HLA* für skalierbare und parallele Simulation eingesetzt wird.

A9 Die Anforderung wird nicht erfüllt. *HLA* bietet keine API.

3.2.8 Analyse unberücksichtigter Forschungsarbeiten

Die Literaturrecherche für das vorliegende Framework umfasste viele weitere Artikel, jedoch wurden nicht alle Artikel berücksichtigt, um im aktuellen Kapitel ausführlich analysiert zu werden. Daher werden in diesem Abschnitt weitere Artikel kurz

beschrieben. Zusätzlich wird der Grund angegeben, warum sie nicht berücksichtigt wurden.

Die Artikel [PBR13; Cel+14; AN11; CSG08; Lév+12; Shu+18b; All+21] definieren jeweils eine Co-Simulations-Architektur, spezifizieren das Konzept jedoch nicht detailliert genug, sondern fokussieren sich auf die Modellierung einer stationären Simulation für Energiesystem-Simulationen. Daher sind sie in dieser Arbeit nicht zu bewerten.

In Tab. 3.2 werden die Co-Simulationsansätze dargestellt, in denen die drahtlose Sensornetzwerktechnologie zur Kommunikation eingesetzt wird [Cho+13]. Der erste Co-Simulationsansatz von *ns-2/adevs* (**A** **D**iscrete **EV**ent **S**ystem simulator) vermeidet einige Synchronisationsprobleme, die durch die Kopplung eingeführt werden, indem *adevs* [Nut22] als Modul in *ns-2* [Nut+07] integriert wird. Dies erhöht jedoch die Komplexität der Stromverhaltensformalisierung, welche die Leistung einer solchen Plattform drastisch verringern kann. Beim *ns-2/OpenDSS*-Kopplungsansatz [God+10] ist die Interkommunikationstechnik zwischen den beiden Simulatoren unzureichend definiert und muss manuell festgelegt werden. In [Lin+11] wird ein Co-Simulations-Framework durch Integration von *PSLF* (**P**ositive **S**equence **L**oad **F**low) und *ns-2* implementiert, um die praktische Untersuchung von Smart Grids zu verbessern und Mess- und Steuersysteme zu bewerten. Vor der Verwendung der Schnittstelle des Frameworks müssen jedoch viele neue Module in *ns-2* implementiert werden. Die *ns-2/Modelica*-Kopplungstechnik [LA11] kann zum Integrieren von Modellen drahtloser Sensornetzwerke mit den meisten vorhandenen Kommunikationstechnologien wie WIMAX (**W**orldwide **I**nteroperability for **M**icrowave **A**ccess [GB12]), Wi-Fi, WPAN (**W**ireless **P**ersonal **A**rea **N**etwork [KLZ04]) etc. in *Modelica* verwendet werden, aber *Modelica* kann nicht bestimmen, wann Daten an *ns-2* gesendet werden sollen. In [Ass22b] ist *THYME* (**T**oolkit for **HY**brid **M**odeling of **E**lectric power systems) nur eine Bibliothek und kein eigenständiger Simulator. Vor der Verwendung muss es mit *adevs* von den Nutzern zusammen implementiert werden. Alle genannten Ansätze zur Kopplung von drahtlosen Kommunikationsmodellen an Smart Grids werfen einige Probleme auf, daher sind sie in diesem Kapitel nicht mit den Anforderungen ausführlich evaluiert.

Außerdem gibt es noch viele Artikel (z.B. [BSS18; Wan+15; SGZ19; Wid+15; UPA19; ABL08; Al-12; Zha+10; Eyi+12; Bia+15]), die jeweils eine Plattform für Applikationen von Smart Grids oder anderen Forschungsbereichen darstellen. Alle Plattformen sind jedoch nur auf ein spezifisches Szenario oder bestimmte Typen von Simulatoren oder Softwareprodukten beschränkt. Sie bieten keine universelle Anwendbarkeit, Skalierbarkeit oder Erweiterbarkeit.

Tab. 3.2.: Liste der Co-Simulations-Frameworks mit drahtloser Netzwerktechnologie

Framework	Problem
NS-2/adevs [Nut+07]	Komplexität der Stromverhaltensformalisierung
NS/OpenDSS [God+10]	Die Interkommunikationstechnik ist nicht definiert
NS-2/PSLF [Lin+11]	Viele neue Module müssen in ns-2 implementiert werden
NS-2/Modelica [LA11]	Modelica kann nicht bestimmen, wann Daten an ns-2 gesendet werden sollen
THYME/OMNET++ or NS-2 [Ass22b]	THYME ist nur eine Bibliothek und kein Simulator

3.3 Zusammenfassung und resultierender Handlungsbedarf

Auf Grundlage dieser Auswertung von bestehenden Arbeiten (siehe Abschnitt 3.2) fasst die Tabelle 3.3 die Ergebnisse zusammen. Die Tabelle ist wie folgt zu interpretieren: Es wurde mit ● gekennzeichnet, wenn eine Anforderung vollständig von einer Arbeit erfüllt wird. Entsprechend repräsentiert ○ das Gegenstück, dass eine Anforderung von einer Arbeit nicht erfüllt bzw. nicht berücksichtigt wird. Außerdem wird mit ○ markiert, dass eine Anforderung zwar erwähnt ist, ohne jedoch weiter erörtert zu werden, oder dass ihre Rolle in der Forschungsarbeit unklar bleibt. Hingegen weist ◐ darauf hin, dass sich eine Anforderung von einer Arbeit nur teilweise erfüllen lässt.

Wie die Tabelle 3.3 zeigt, stellt keines der analysierten Frameworks eine Lösung dar, die alle Anforderungen für einen Entwurf zur Co-Simulation und Automatisierung von wissenschaftlichen Workflows in Abschnitt 3.1.2 erfüllt. Dieses Ergebnis bildet gleichzeitig die Motivation für die eigenen Beiträge in der vorliegenden Arbeit, die sich jedoch auch auf die Konzepte der hier ausgewerteten Arbeiten stützt.

Die Analyse der bestehenden Forschungsarbeiten zeigt, dass die Datenaustauschschnittstelle der meisten Frameworks auf einer Middleware beruht und daher als Kommunikationsinfrastruktur angesehen werden kann, um den Datenfluss zwischen Workflow-Applikationen performant zu organisieren und zu koordinieren (A3).

Allerdings findet sich in keinem der untersuchten Frameworks eine plattformunabhängige grafische Oberfläche zur Erstellung und Ausführung sowie Steuerung der Workflows (A6). Dies ist eine erste zu berücksichtigende Lücke und daher ist dieser Umstand bereits zu Beginn des Entwurfs in adäquater Weise einzubeziehen.

Da sich Architekturentwürfe zunehmend auf Modularität und Portabilität konzentrieren, ist die Unterstützung des Software-Architekten beim Entwurf im Hinblick auf die Qualitätsteilmerkmale der Wiederverwendbarkeit entscheidend. An dieser Stelle wird die Anforderung A7 jedoch von fast allen relevanten Frameworks nur teilweise erfüllt, da die integrierten Applikationen für jedes Framework nur in der spezifischen Laufzeitumgebung wiederverwendbar und schwer umzusetzen sind. Dies ist eine zweite zu berücksichtigende Lücke.

Weiter ist eine Benutzeroberfläche zur einfachen Konfiguration (A1) oder Integration (A2) der Applikationen erforderlich, was keines der analysierten Frameworks im Detail aufzeigt und bereitstellt. Stattdessen verlangen alle untersuchten Frameworks, dass Konfigurationsdateien oder Integrationsschnittstellen von den Nutzern selber fest codiert werden müssen. Dies stellt eine dritte Lücke dar.

Darüber hinaus bieten nur die relativ generischen Frameworks, z.B. *HELICS*, *Snakemake*, *Mosaik* und *HLA*, die Fähigkeit zur Automatisierung (A5), zur Parallelisierung (A8) oder die Unterstützung zur Ausführung großer und komplexer Workflows (A4). Allerdings kann kein einziges Framework alle drei Anforderungen gleichzeitig vollständig erfüllen. Dies bildet eine vierte Lücke.

Schließlich wird die letzte Anforderung A9 von allen Frameworks auch nur partiell abgedeckt. Dies präsentiert die letzte Lücke.

In den folgenden drei Kapiteln werden drei Beiträge der vorliegenden Arbeit beschrieben, von denen die in diesem Kapitel definierten neun Anforderungen vollständig erfüllt werden.

Tab. 3.3.: Übersicht der Anforderungen, die durch verwandte Arbeiten erfüllt werden: ○-nicht; ◐-teilweise; ●-vollständig

Framework	A1	A2	A3	A4	A5	A6	A7	A8	A9
OOCoSim CoSimulating Communication Networks and Electrical System for Performance Evaluation in Smart Grid [Kim+18]	○	○	●	○	○	○	◐	◐	◐
FNCS Open-source framework for power system transmission and distribution dynamics co-simulation [Hua+17]	◐	○	●	●	○	○	◐	○	◐
HELICS Design of the HELICS high-performance transmission-distribution-communication-market co-simulation framework [Pal+17b]	○	○	●	●	●	○	◐	○	◐
Snakemake a scalable bioinformatics workflow engine [KR12]	◐	○	◐	●	●	○	◐	◐	◐
PowerNet Smart grid communication and co-simulation [LA11]	○	○	○	○	○	○	○	○	◐
Mosaik A framework for modular simulation of active components in Smart Grids [SST11]	◐	○	●	○	●	○	◐	◐	◐
HLA High Level Architecture [DKW98]	○	●	●	●	○	○	●	●	○

4 Framework-Architektur

Nach der Einführung der wichtigen Basistechnologien ist dieses Kapitel das erste von drei Kapiteln, in denen die im Rahmen der Arbeit erarbeiteten wissenschaftlichen Beiträge vorgestellt werden. Dieses Kapitel erörtert daher die Frage: Wie lassen sich wissenschaftliche Applikationen ausführen, um Aufbau und Automatisierung von Workflows auf einer web-basierten grafischen Oberfläche zu ermöglichen? Dabei wird der erste Beitrag aus zwei Teilen basierend auf den Anforderungen 3.1.2 (A1, A4, A5, A6, A7, A9) gebildet:

i Konzeption und Umsetzung einer Grundarchitektur für die Beschreibung und Ausführung von wissenschaftlichen Workflows und Co-Simulationen unter Verwendung modularer Bausteine (sogenannter Framework-Prozesse). Die Bausteine kapseln die Softwarewerkzeuge und ihre Ausführungsumgebung in einer Weise, dass sie automatisiert auf Computing-Umgebungen ausgeführt werden können.

ii Integration / Nutzung einer benutzerfreundlichen Weboberfläche zum Erstellen, Betreiben und Verwalten von Workflows und Erweiterung der Lösung um fehlende Funktionalitäten für wissenschaftliche Workflows.

Das Kapitel ist folgendermaßen strukturiert: Abschnitt 4.1 erläutert zunächst die wichtigsten Begrifflichkeiten für die Analyse des wissenschaftlichen Hintergrunds und die Beschreibung der PROOF-Architektur. Anschließend wird eine Multiplayer-Simulations-Architektur (Co-Simulations-Architektur) zur Vorbereitung des Entwurfs der PROOF-Architektur in Abschnitt 4.2 näher erläutert. Danach fokussiert sich Abschnitt 4.3 auf die Vorstellung der PROOF-Architektur und ihre Umsetzung unter Verwendung einer Microservices-Architektur, welche Containervirtualisierung verwendet. Abschnitt 4.4 stellt den Aufbau eines PROOF-Prozesses vor, um eine Applikation unter Berücksichtigung ihrer Laufzeit- und Arbeitsumgebung auf einem Cluster automatisiert auszuführen und als einen NiFi-Prozessor auf der Weboberfläche repräsentieren zu können. Abschnitt 4.5 beschreibt die Integration der auf Apache NiFi basierenden Weboberfläche zum grafischen Aufbau wissenschaftlicher Workflows im Detail. Abschnitt 4.6 fasst die zuvor definierten Anforderungen zusammen, die von dem ersten Beitrag erfüllt werden.

4.1 Begrifflichkeiten

Bevor die Framework-Architektur vorgestellt wird, werden einige Begrifflichkeiten eingeführt und mögliche mehrdeutige Definitionen geklärt, sodass sie bei der Beschreibung von Framework-Architektur und -Funktionalitäten in diesem und den folgenden Kapiteln eindeutig verwendet werden können.

Testdaten

Testdaten sind Daten, die extern in das System eingespielt werden können, um dessen Verhalten bzgl. vorgegebener Evaluationseigenschaften im wissenschaftlichen Kontext eines konkreten Anwendungsfalls zu testen.

Systemmodell

Ein Systemmodell dient der Evaluation des Verhaltens einer Menge von Komponenten im Simulationskontext. Dazu gehören nicht nur Applikationen der einzelnen Systemkomponenten, sondern auch die Modellierung von deren Interaktionsschnittstellen im systemischen Kontext. Darüber hinaus muss der Einfluss der externen Umgebung auf das System realisiert werden, z.B. durch Schnittstellen zum Einspielen von Testdaten für das Testsystemmodell oder durch die Integration weiterer Hilfsberechnungen/Werkzeuge zum Erzeugen oder Sammeln von Testdaten.

Wissenschaftliche Applikation

Eine wissenschaftliche Applikation ist eine einzelne Komponente, die im wissenschaftlichen Systemmodell als ein Modell, ein Simulator oder eine Anwendung gelten kann.

Framework-Prozess/Baustein

Ein Framework-Prozess/Baustein repräsentiert einen Prozess, der eine wissenschaftliche Applikation und ihre Abhängigkeiten (z.B. Laufzeitumgebung, Bibliotheken usw.) in einem individuellen Docker-Image einpackt und in einem Docker-Container ausgeführt werden kann.

NiFi-Prozessor

Auf der PROOF-Weboberfläche gilt ein Framework-Baustein als ein NiFi-Prozessor (NiFi-Block), der in der Prozess-Bibliothek von PROOF bereitgestellt und von Nutzern konfiguriert, gesteuert und ausgeführt wird.

Simulation

Eine Simulation ist das Durchführen des Testsystemmodells. Die Durchführung besteht aus vier Teilen:

1. Spezifikation des Testfalls und Einsammeln sowie Einspielen der entsprechenden Testdaten,
2. Ausführung der wissenschaftlichen Applikation mit einem Testfall,
3. Festhalten der Ergebnisse,
4. Visualisierung sowie Analyse der Ergebnisse.

4.2 Multiplayer-Simulations-Architektur

Modellierungs- und Simulationstechniken werden heutzutage in verschiedenen Bereichen der Industrie und Wissenschaft, wie z.B. bei Energiesystemen, in der Produktionsindustrie und den Sozialwissenschaften umfassend eingesetzt. Die einzelnen Teile größerer Systeme werden typischerweise mit unterschiedlichen Techniken, Werkzeugen und Algorithmen modelliert und simuliert. Darüber hinaus nutzen Experten aus unterschiedlichen Disziplinen und Anwendungsdomänen verschiedene Modellierungs- und Simulationstechniken. Deshalb wird die Bewertung des Gesamtverhaltens solcher Systeme in jeder Phase ihrer Entwicklung komplizierter. Grund hierfür sind die zunehmende Komplexität von Systemen, des Marktwettbewerbs und der Spezialisierung der Teilsysteme.

Zur Evaluation der Simulationsergebnisse einer komplexen Systemlösung (z.B. Energiesystemlösung) müssen ihre Komponenten zu einer systemischen Gesamtsimulationsumgebung verknüpft und zusammengesetzt werden. Damit kann die Interaktion verschiedener technischer Anlagen und nicht-technischer Komponenten (wie z.B. Marktmodelle) in ihrem Zusammenspiel getestet werden. Man spricht für eine spezielle Form dieser Verknüpfung häufig auch von Multiplayer-Simulation (Co-Simulation) [Wik22a; Gom+18; Sch+19], die eine Methodik zum Aufbau solcher

Systemmodellierungs- und -analyseumgebungen darstellt. Bei der Co-Simulation kann eine globale Simulation eines gekoppelten Systems erreicht werden, indem die Simulationen seiner Teile zusammengeführt werden. Beispielsweise kann ein Primary-Simulator dabei andere Modellausführungen als Secondary-Simulatoren einbinden. Alle Subsystemmodelle mit ihren Simulationswerkzeugen werden auf ihren Verhaltensebenen miteinander verbunden und jeweils als Blackbox betrachtet, die Outputs erzeugen und Inputs verbrauchen kann [Van+12].

Wie in Abb. 4.1 dargestellt, erfordert die Simulation von Smart Grids die Zusammenarbeit vieler verschiedener Subsystemmodelle: Solaranlage, Windkraftanlage, Gebäude, Batterie, Marktsimulation usw. Diese Modelle unterscheiden sich sowohl in ihrer Darstellung als auch in ihrer Laufzeitumgebung. Sie müssen in einer Co-Simulation integriert und zusammengefügt werden, um große transdisziplinäre Multi-Domain-Energiesysteme aufzubauen und zu simulieren, indem solche Modelle miteinander kommunizieren und auch koordiniert werden.

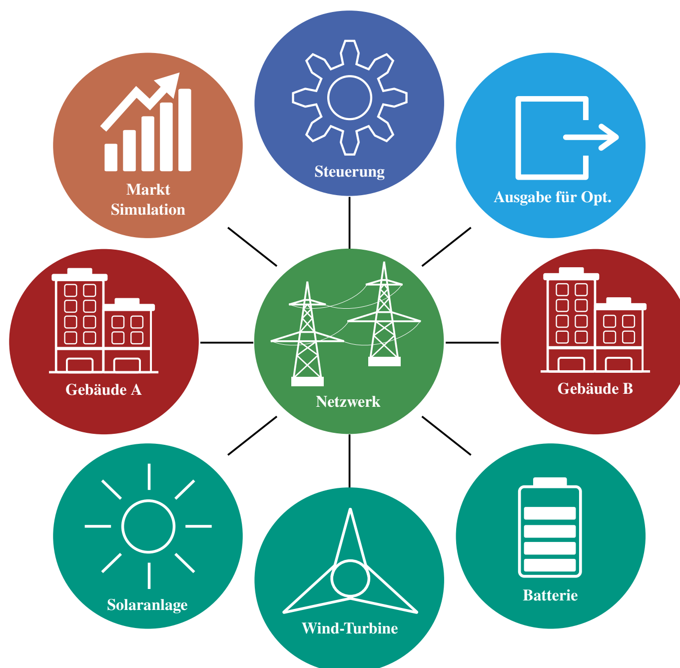

Abb. 4.1.: Co-Simulation im Bereich Smart Grids

Zur Simulation von Energiesystemlösungen wird eine Co-Simulations-Architektur in der vorliegenden Arbeit implementiert und zur Evaluation in einem Windkraftanlagen-Anwendungsfall eingesetzt. Im folgenden Abschnitt 4.2.1 wird eine Übersicht über die Co-Simulations-Architektur gegeben. In Abschnitt 4.2.2 erfolgt die ausführliche Vorstellung der Web-Benutzeroberfläche, die zur Erstellung von Co-

Simulationen entwickelt wird. Am Ende wird eine beispielhafte Co-Simulation aus drei Simulatoren mit einem speziellen Adapter für Matlab-Simulatoren in Abschnitten 4.2.3-4.2.4 vorgestellt, um weiter zu erklären, wie die Ausführung von Simulationsmodellen in Docker-Containern verwaltet und wie die Interaktion zwischen den Matlab-Simulatoren und anderen Komponenten über die Kommunikationsschnittstelle (Apache Kafka) implementiert wird. In Anhang A.1 wird der Anwendungsfall einer Windkraftanlage als Beispiel einer Co-Simulation zur Bewertung der Praxistauglichkeit beschrieben.

4.2.1 Übersicht

Die Multiplayer(Co)-Simulations-Architektur ist in Abb. 4.2 veranschaulicht. Sie besteht aus einer Web-Benutzeroberfläche mit mehreren Hauptansichten (siehe oberer Teil von Abb. 4.2), einem Asset-Manager zur Verwaltung von Docker-Images, dem Co-Simulationsdienst, der Kommunikationsinfrastruktur und den Adaptern zum Anschließen der Simulatoren an die Co-Simulationsumgebung.

Die web-basierte **Oberfläche** stellt grafische Benutzerschnittstellen bereit, um verschiedene Simulatoren einzufügen und mit der Kommunikationsschnittstelle zu verbinden sowie ihre Eingabe und Ausgabe zu definieren, und um Co-Simulationen grafisch zu erstellen. Hierauf wird im folgenden Abschnitt 4.2.2 näher eingegangen.

Als Schlüsselkomponente baut der **Co-Simulationsdienst** die Interaktion zwischen der Weboberfläche und der Kommunikationsinfrastruktur auf. Basierend auf einer **Co-Simulationskonfiguration**, die durch ein JSON-Format beschrieben wird, können die entsprechenden Docker-Images der auf der Weboberfläche dargestellten Simulatoren im **Asset-Manager** herausgefunden werden. Mithilfe der Kommunikationsinfrastruktur kann der Co-Simulationsdienst verschiedene Docker-Container starten, überwachen und steuern, in denen die Simulatoren automatisch ausgeführt werden. Dann können die Ergebnisse von Co-Simulationen mit dem Co-Simulationsdienst zurück an die Weboberfläche weitergeleitet und schließlich darauf visualisiert werden. Der Co-Simulationsdienst funktioniert sowohl als Brücke zwischen dem Co-Simulationsfrontend und -backend zur Zusicherung der Datenübertragung als auch als ein Primary-Simulator zur Steuerung anderer Secondary-Simulatoren.

Als Message-Server wird Apache Kafka in der **Kommunikationsinfrastruktur** integriert, die generische Microservices zum Zugriff auf Kafka-Nachrichtenkanäle für Datenaustausch zwischen einzelnen Simulationsknoten und dem Co-Simulationsdienst bietet.

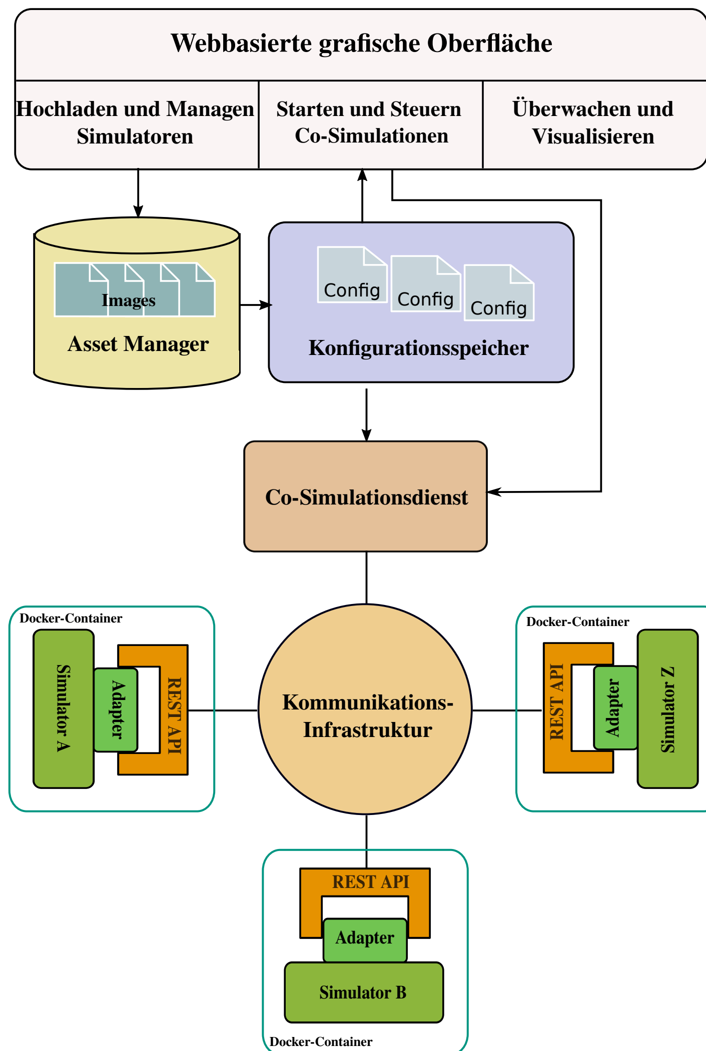

Abb. 4.2.: Co-Simulations-Architektur

Spezifische **Adapter** werden für verschiedene Arten von Simulatoren als Kommunikationsschnittstelle implementiert. Damit können unterschiedliche Simulatoren mit der Kommunikationsinfrastruktur sowie dem Co-Simulationsdienst Daten austauschen. Die Rolle der Adapter wird in Abschnitt 4.2.4 beispielhaft an einem Adapter für Matlab-Simulatoren erklärt. Im Folgenden werden alle Komponenten der Architektur aus technischer Sicht detailliert vorgestellt.

4.2.2 Web-basierte Benutzeroberfläche

Wie der obere Teil von Abb. 4.3 zeigt, bietet die Co-Simulationsplattform verschiedene grafische Benutzerschnittstellen auf der Weboberfläche. Hierzu können Be-

nutzer verschiedene Simulationsmodelle als Modellkomponenten hochladen, Co-Simulationen definieren und Simulationsabläufe bedienen, steuern sowie überwachen und analysieren. Unter Verwendung des Web-Frameworks *Angular2* [Com22a; TAV21] und der Webtechnologien *HTML5* [Hic22] und *Typescript* [Com22c] ist die Weboberfläche der Co-Simulationsplattform entwickelt.

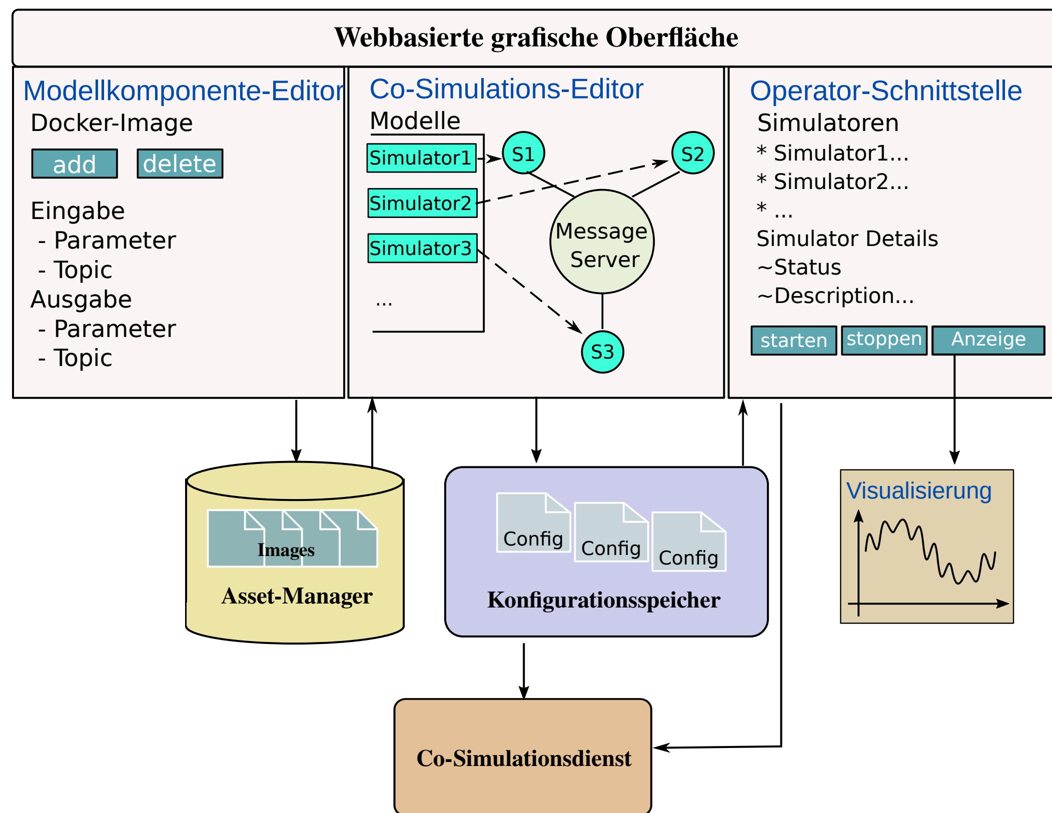

Abb. 4.3.: Web-basierte grafische Oberfläche

Die Weboberfläche enthält einen Modellkomponente-Editor, einen Co-Simulations-Editor und eine Operator-Schnittstelle.

- Unter Verwendung des **Modellkomponente-Editors** (siehe links oben in Abb. 4.3) können nicht nur verschiedene Arten von Simulationsmodellen, z.B. zur Modellierung von Stromnetzen und technischen Anlagen, sondern auch verschiedene Arten von Datenquellen und Datenknoten, die eine Schnittstelle zu realen Hardwareknoten darstellen, hochgeladen und in die Co-Simulationsplattform integriert werden. Daher müssen Docker-Images, die die Software und Abhängigkeiten zur Implementierung der Laufzeitumgebung oder der Datenquellenfunktionalität enthalten, über Metadaten als Modellkomponente definiert und beschrieben werden. Diese Komponente kann dann zur Co-Simulationsmodellbibliothek hinzugefügt werden. Die Beschreibung

solcher Modellkomponenten werden im *Asset-Manager* gespeichert, um später vom Co-Simulation-Editor und dem Co-Simulationsdienst verwendet werden zu können.

- Co-Simulationen können mithilfe des **Co-Simulations-Editors** erstellt werden (siehe oberer mittlerer Teil von Abb. 4.3). Auf der linken Seite dieser Ansicht werden verfügbare Modellkomponenten vom *Asset-Manager* in der Tabelle „Modelle“ aufgelistet. Benutzer können die zugehörige Modellkomponente mit der Maus auswählen und per Drag-and-Drop auf die rechte Seite in der Nähe des Message-Servers ziehen, um einen neuen Knoten für die Co-Simulation zu erstellen. Der Knoten kann dann zum Datenaustausch mit der Message-Serverwarteschlangen verbunden werden. Nach dem Aufbau einer Co-Simulation werden die entsprechenden Konfigurationsdaten für die gesamte Co-Simulation in einer JSON-Datei im Konfigurationsspeicher abgespeichert, um später vom Co-Simulationsdienst zur Erstellung entsprechender Docker-Container verwendet werden zu können.
- Die **Operator-Schnittstelle** der Weboberfläche (siehe oben rechts in Abb. 4.3) listet die Informationen der im Co-Simulations-Editor aufgebauten Co-Simulation gestützt auf der Konfigurationsdatei im Konfigurationsspeicher auf. Darüber hinaus bietet der Operator-Schnittstelle Schaltflächen als Steuerelemente, um zugeordnete Funktionen auszulösen, z.B. die Steuerung (starten, stoppen) der Co-Simulation. Außerdem wird eine weitere Schaltfläche zur Visualisierung von Ergebnissen der Co-Simulation bereitgestellt. Nach einem Simulationslauf werden alle zu einer Simulation gehörenden Daten zum anschließenden Abrufen und Analysieren in jeweils einer **D**aten**b**ank (DB) gespeichert, z.B. Stammdaten in einer NoSQL-DB [PK17; Gup+17], Zeitreihendaten in OpenTSDB [Ope22] oder InfluxDB [Inf22].

4.2.3 Beispielhafter Simulationsablauf

Abb. 4.4 zeigt zur Veranschaulichung das Konzept einer generischen Co-Simulation mit drei Simulatoren und Apache Kafka als Message-Server. Aufgrund des Consumer-Group-Concepts von Kafka kann jeder Simulator seine Daten in einer oder mehreren Topic-Warteschlangen in Kafka veröffentlichen und ein oder mehrere Topics abonnieren, um Daten vom Kafka-Server zu empfangen. Simulatoren in verschiedenen Consumer-Gruppen können auch Daten zu einem Topic gemeinsam teilen. Darüber hinaus werden Datenströme in der verteilten, replizierten Serverinfrastruktur von Apache Kafka fehlertolerant zwischengespeichert.

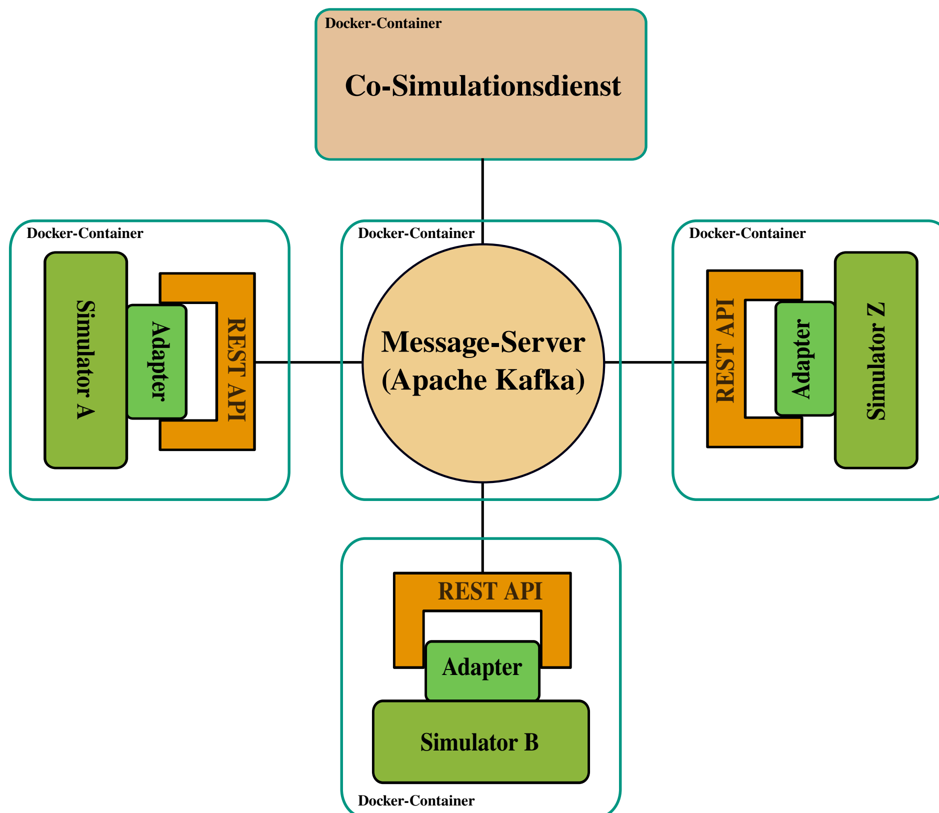

Abb. 4.4.: Co-Simulation mit drei Simulatoren

Alle Komponenten des Konzepts sind in Abb. 4.4 mit einem abgerundeten, grünen Viereck umrahmt und werden als Microservices in unabhängigen Docker-Containern betrachtet. Alles, was zum Ausführen der Simulatoren erforderlich ist, z.B. Quellcode, Laufzeitumgebung, Bibliotheken, Konfigurationsdateien usw., wird in einen Container verpackt und bereitgestellt. Daher ist es nicht erforderlich, bestimmte Betriebssystemumgebungen für verschiedene Arten von Simulatoren zu installieren. Alle Simulatoren können als separate, aber synchronisierte Prozesse in ihren Docker-Containern auf dem Computing-Cluster ausgeführt werden. Durch die Nutzung der Docker-API können die Simulatoren außerdem einfach und effizient verwaltet und gesteuert werden.

Als Blackbox enthält jeder Simulator seine eigene Simulationsaufgabe und kommuniziert mit dem externen Message-Server lediglich über eine Schnittstelle (REST-API) und einen geeigneten Adapter. Die Schnittstelle und der Adapter sind wiederverwendbar und können ohne Änderung für neue Simulatoren in Docker-Containern eingesetzt werden. Daraus ist ersichtlich, dass das Konzept in Abb. 4.4 nicht auf nur drei Simulatoren beschränkt, sondern skalierbar für verschiedene komplexe Anwendungsszenarien ist.

4.2.4 Rolle der Adapter

Dieser Abschnitt erklärt die Rolle von Adaptern, indem demonstriert wird, wie ein Adapter einen Simulator mit der Kommunikationsinfrastruktur der Co-Simulation zum Datenaustausch verbindet (siehe Abb. 4.5).

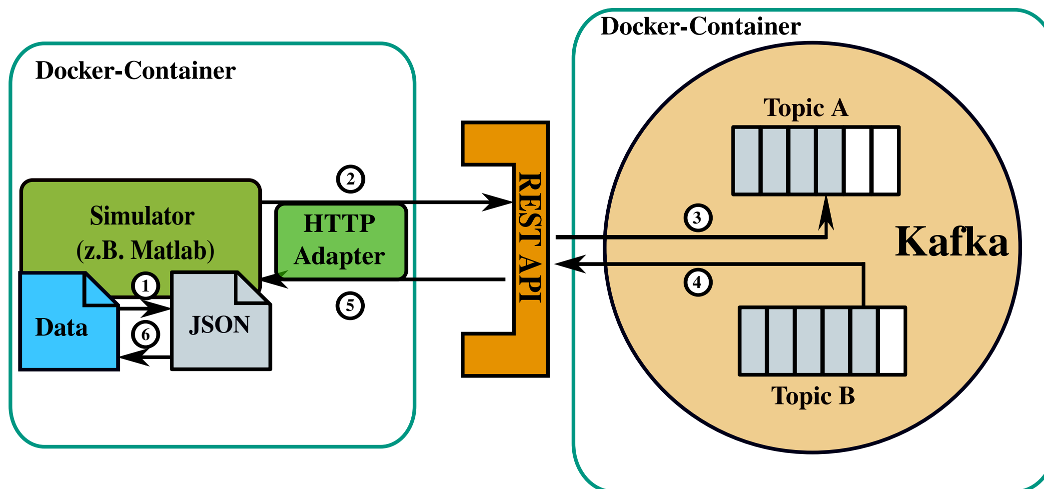

Abb. 4.5.: Kommunikation zwischen einem Simulator und Apache Kafka

In Abb. 4.5 wird beschrieben, wie der Datenaustausch zwischen einem einzelnen Simulator-Knoten und der Co-Simulations-Kommunikationsinfrastruktur (Apache Kafka) implementiert wird. Die zum Senden und Empfangen von Daten erforderlichen Kommunikationsschritte können unterteilt werden in:

Daten senden

1. Die zu sendenden Daten vom Simulator werden in ein JSON-Objekt konvertiert.
2. Über einen HTTP-Adapter wird das JSON-Objekt an das REST-API-Frontend der Kommunikationsinfrastruktur gesendet.
3. Das REST-API-Frontend sendet das JSON-Objekt dann an ein bestimmtes Kafka-Topic, das für diese Art von Daten erstellt wurde. Das Topic speichert die Daten in einer Warteschlange auf dem Kafka-Server, bis alle anderen Simulatoren, die dieses Topic abonniert haben, den Verbrauch dieses Objekts abgeschlossen haben.

Daten empfangen

4. Der Simulator fordert Daten an, die von der Kommunikationsinfrastruktur an die REST-API des Adapters über einen HTTP-Aufruf aus der abonnierten Kafka-Topic-Warteschlange abgerufen werden.

5. Der HTTP-Adapter ruft die entsprechenden Daten aus der REST-API ab. Sobald die Daten ankommen, stellt der Adapter sie in einem JSON-Objekt für den Simulator bereit.
6. Der Simulator empfängt das JSON-Objekt, das dann zur weiteren Verwendung in geeignete Typen (z.B. ein Matrixobjekt für Matlab-Simulatoren) umgewandelt werden kann.

Während der in Abb. 4.5 dargestellte HTTP-Adapter eindeutig Matlab-spezifisch ist, unterstützen viele Simulatoren (einschließlich Matlab) den FMI-Standard für den Modell- und Datenaustausch in Co-Simulationen. Für den Datenaustausch definiert der FMI-Standard eine Standard-C-API, die zusammen mit allen Simulatoren verwendet werden kann, die diesen Standard unterstützen. Deshalb wird ein generischer FMI-Adapter von der Co-Simulationsplattform bereitgestellt. Damit können alle Simulatoren, die die FMI-Spezifikation implementieren, mit der Co-Simulationsplattform integriert und verknüpft werden. Dies verbindet eine ganze Reihe von Simulatoren auf sehr generische Weise mit der Co-Simulationsplattform. Hierbei handelt es sich um eine große Menge von Simulatoren für vielfältige Systemlösungen, um hohe Skalierbarkeit und Erweiterbarkeit der Co-Simulationsplattform zu erzielen.

4.3 PROOF-Konzept

Nachdem mit dem vorherigen Abschnitt eine Übersicht über den Entwurf der Co-Simulationsplattform sowie eine Beschreibung ihrer Komponenten (Weboberfläche, Co-Simulationsdienst, Kommunikationsinfrastruktur und Adapter) und des damit einhergehenden Anwendungsfalls der Windanlage gegeben wurde, widmen sich die folgenden Abschnitte der Vorstellung der PROOF-Architektur und ihrer Komponenten.

4.3.1 Übersicht

Systemmodelle lassen sich relativ einfach implementieren und simulieren, wenn ihre Komponenten und Artefakte in einer Implementierungsumgebung mit einer einzigen Programmiersprache/Software implementiert werden können. In der Praxis besteht ein Systemmodell jedoch aus verschiedenen Komponenten, die in unterschiedlichen Systemen entwickelt und zusammengebaut werden müssen. Dann ist ein verteilter Simulationsansatz erforderlich, bei dem ausführbare Applikationen/Modelle

zunächst unabhängig voneinander in eigenen Laufzeitumgebungen ausgeführt und miteinander verknüpft werden, um vollständige Systemmodelle aufzubauen und zu simulieren. Systemmodelle werden in vielen wissenschaftlichen Projekten, insbesondere im Ingenieurwesen, als wissenschaftliche Workflows bezeichnet und benötigen eine Plattform nicht nur zur Co-Simulation, sondern auch zur Erfassung und Verarbeitung ihrer Daten, zur Automatisierung der Ausführung ihrer Simulationen und Modelle sowie zur Persistierung und Präsentation ihrer Ergebnisse.

Basierend auf der im vorherigen Abschnitt vorgestellten Co-Simulationsarchitektur wird ein neuer Simulationsansatz, nämlich PROOF, als eine erweiterte Plattform zur verteilten Ausführung von wissenschaftlichen Workflows konzipiert, die nicht nur Co-Simulationen, sondern auch verteilte Simulationen und Datenflüsse unterstützt. Auf dieser durchgängigen Plattform können Datenerfassung, -verarbeitung und -übertragung realisiert werden. Zur Simulation eines Systemmodells können komplexe wissenschaftliche Workflows auf der Plattform flexibel aufgebaut und ausgeführt werden, um die Evaluation für das Systemmodell mit Testdaten zu erleichtern. Die grundlegende Architektur von PROOF ist in Abb. 4.6 dargestellt. Die Struktur der Architektur besteht aus den folgenden Komponenten, die jeweils in einem Docker-Container ausgeführt werden:

- Die **Weboberfläche** basiert auf der Apache NiFi-Benutzeroberfläche. Darauf werden verschiedene **NiFi-Prozessoren**, die Framework-Prozesse visuell veranschaulichen, aus der Prozess-Bibliothek ausgewählt, eingefügt und anschließend gekoppelt, um wissenschaftliche Workflows visuell aufzubauen. Durch Konfiguration der NiFi-Prozessoren können die entsprechenden Framework-Prozesse und ihre Applikationen parametriert werden.
- Die **NiFi-Prozessorschnittstelle** wird als Brücke zur Interaktion zwischen dem NiFi-Prozessor und der Kommunikationsinfrastruktur des Frameworks erweitert, indem die Java-Bibliothek *Jedis* zum Aufbau der Verbindung mit dem Redis-Server eingesetzt wird. Dadurch werden Aufgaben (Tasks), Ein- und Ausgaben ausgetauscht.
- Die **Kommunikationsinfrastruktur** wird basierend auf Redis zur Realisierung des Datenaustauschs und der Koordinierungslösung für Framework-Prozesse implementiert. Sie wird als in-memory Datenstrukturspeicher, Datenbank, Cache und Nachrichtenbroker für den Aufbau der Kommunikation zwischen allen Komponenten von PROOF eingesetzt.
- Das **Prozessmanagement** steuert Framework-Prozesse in dynamischen ausführbaren Containern, z.B. starten, stoppen, parallelisieren.

- Der **Framework-Prozess** beinhaltet eine wissenschaftliche **Applikation**, ihre Laufzeitumgebung und Schnittstelle sowie einen **Ein-/Ausgabe-Adapter**. Jeder Framework-Prozess wird in einem Docker-Image gekapselt und als Blackbox in Containern ausgeführt. Der Ein-/Ausgabe-Adapter, der basierend auf der Schnittstelle der Applikation von PROOF bereitgestellt wird, unterstützt verschiedene Schnittstellen zum Datenaustausch zwischen der Applikation und den externen Services, wie z.B. Dateisystem, Datenbank oder Kommunikationsinfrastruktur.
- Das **Datenspeicher-Volume** wird automatisch vom Container zum einfachen Datenaustausch zwischen der gekapselten Applikation und der Workflow-Framework-Umgebung bereitgestellt. Für einige spezielle Anwendungsfälle müssen große Datenmengen zwischen Applikationen ausgetauscht werden, die relativ langsam über die Kommunikationsinfrastruktur übertragen werden können. Das Volume wird von einem verteilten Dateisystem bereitgestellt und kann von allen Containern gemeinsam geteilt und verwendet werden, um das Speichern und Übertragen großer Datenmengen als Dateien zu realisieren.

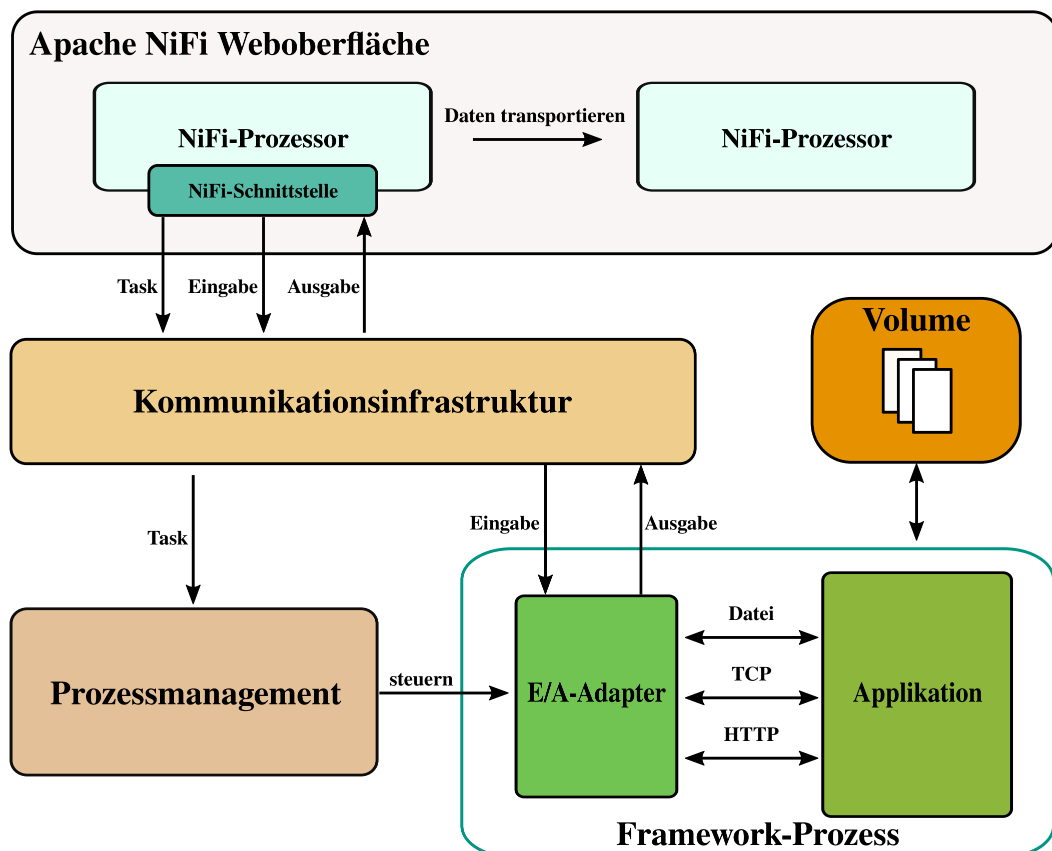

Abb. 4.6.: PROOF-Architektur

4.3.2 Überführung des PROOF-Konzepts in eine Microservices-basierte Architektur

Für die Umsetzung des PROOF-Konzepts (siehe in Abb. 4.6) wird eine Microservices-basierte Architektur eingesetzt, welche die aus dem Cloud-Computing-Bereich bekannte Technologie der Containervirtualisierung und Container-Orchestrierung verwendet, um Softwarewerkzeuge unter Berücksichtigung ihrer Laufzeit- und Arbeitsumgebung automatisiert auf einem Cluster ausführen zu können.

Abb. 4.7 zeigt die konzeptionelle Microservice-Architektur von PROOF, die mit einer hierarchischen Struktur entworfen wird, mit dem Ziel, die Flexibilität, Skalierbarkeit und Wartbarkeit von PROOF zu verbessern. Dabei wird jeder Microservice durch ein Hexagon repräsentiert. Die Komponenten der PROOF-Architektur, die in Abb. 4.6 im vorherigen Abschnitt vorgestellt wurden, nämlich Apache NiFi Weboberfläche, Redis, Prozessmanagement und PROOF-Prozess, werden als lose gekoppelte, leichte, unabhängige Microservices gekennzeichnet. PROOF besteht folglich aus separaten, containerisierten Microservices.

Die Architektur verbirgt die technischen Aspekte der zugrundeliegenden Computerplattform, indem sie eine mehrschichtige Architektur mit zwei Hauptschichten einführt, nämlich der Benutzerschnittstellenschicht im Frontend und der Clusterschicht im Backend. Im Frontend können Nutzer wissenschaftliche Workflows auf der Weboberfläche aufbauen und ausführen. Nutzer müssen nicht berücksichtigen, wie entsprechende Simulationen ausgeführt und benötigte Datenflüsse in den Simulationen realisiert werden. Dies wird vom Backend übernommen und verarbeitet. Das Backend ist in zwei Schichten unterteilt:

1. Die **Containerschicht** umfasst zahlreiche Microservices, die verschiedene Framework-Prozesse betrachten, und einen Microservice, der das Prozessmanagement zur Steuerung anderer Framework-Prozesse repräsentiert. Wie aus Tab. 2.1 ersichtlich ist, weist die Microservice-basierte Architektur im Vergleich zur monolithischen Architektur nützliche Eigenschaften auf, um die genannten Ziele (z.B. hohe Flexibilität, Modularität und Skalierbarkeit) zu erreichen. Jeder Microservice erfüllt eine spezielle Aufgabe (Task) und verwendet seine eigene Technologie und Programmiersprache. Dies gilt als ein wertvolles Merkmal, um parallele und skalierbare Lösungen für PROOF aufzubauen, da jeder Microservice die am besten geeigneten Technologien verwenden und horizontal skaliert werden kann. Darüber hinaus ist jeder Microservice in einem eigenen Container gekapselt, der bei Bedarf dynamisch erstellt werden, um Laufzeitautomatisierung und Plattformunabhängigkeit zu gewährleisten.

2. Die **Datenschicht** widmet sich der Speicherung von Daten und fungiert als Vermittler für den Datenaustausch. Um diese Funktionalitäten abzubilden, realisiert die Datenschicht die Speicherung in permanenter und temporärer Weise. Für PROOF wird Redis als temporärer Speicher eingesetzt, um Zwischendaten zu speichern, die während der Simulation von Workflows zwischen den Framework-Prozessen ausgetauscht werden. Andererseits ist der permanente Speicher (Dateisystem oder Datenbank) dafür verantwortlich, große Datenmengen, Applikationen und Ergebnisse von Workflows zu speichern. Dafür ist eine ausführliche Beschreibung dem Kapitel 5 zu entnehmen.

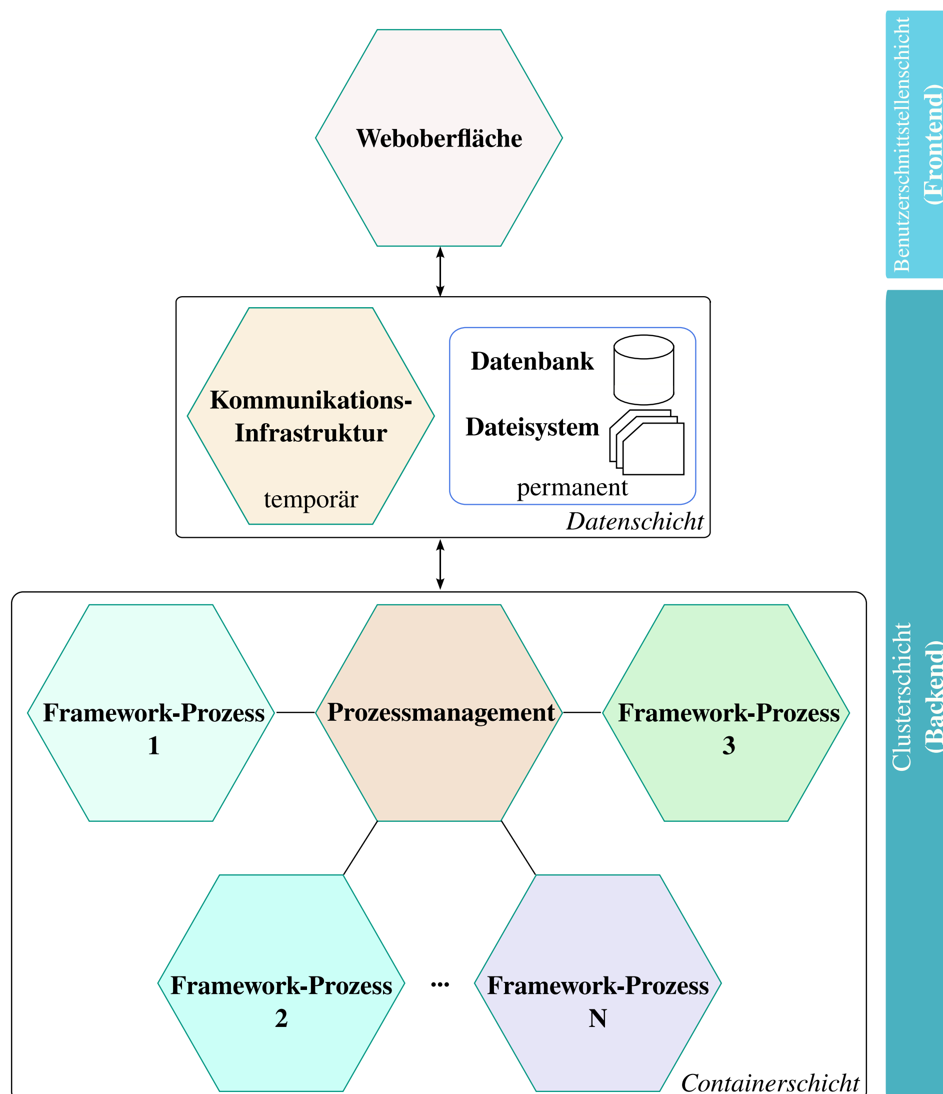

Abb. 4.7.: Die konzeptionelle Microservices-Architektur von PROOF

4.4 PROOF-Prozess

Mit diesem Abschnitt soll die zentrale Fragestellung geklärt werden, wie eine wissenschaftliche Applikation vorbereitet, wie ein entsprechendes Docker-Image angelegt und wie ein korrespondierender NiFi-Prozessor implementiert wird, damit die Applikation in PROOF automatisiert ausgeführt werden kann.

4.4.1 Konzept

Um die Ausführung eines Workflows in PROOF zu unterstützen, müssen alle betroffenen ausführbaren Applikationen in Docker-Containern automatisiert werden. Wie in Abb. 4.8 gezeigt, wird eine *Applikation* und ihre Abhängigkeiten (z.B. *Funktionen*, *Konfigurationen*, *Solver*, *Bibliotheken* und *Laufzeitumgebung*) zur späteren Verwendung in einem *Docker-Image* gekapselt. Darüber hinaus muss eine Modellbeschreibung mit allen wesentlichen Befehlen, die zum Ausführen der Applikation erforderlich sind, von den Benutzern angegeben werden. Die Befehle werden danach in ein Python-Skript mit dem Namen *Wrapper* zum Ausführen der Applikation integriert. Zusätzlich wird ein für die ausführbare Applikation geeigneter *E/A-Adapter* als Schnittstelle verwendet, um den Datenaustausch mit der ausführbaren Applikation und mit externen Services zu instrumentieren. Wie in Abb. 4.8 dargestellt, bietet PROOF in Bezug auf den Datenaustausch verschiedene E/A-Adapter an, z.B. *Datei-Adapter*, *TCP-Adapter*, *HTTP-Adapter* usw. Basierend auf dem Docker-Image kann das im vorherigen Abschnitt beschriebene *Prozessmanagement* einen oder mehrere *Docker-Container* als Blackbox mittels eines Wrappers starten, um die Applikation auszuführen und zu steuern. Weitere Details des Konzepts hinsichtlich des PROOF-Wrappers werden im nächsten Abschnitt 4.4.2 erläutert.

Die Integration neuer Applikationen und Werkzeuge bedeuten zwar einen zusätzlichen Aufwand, jedoch bringen sie einige Vorteile, die diesen Aufwand rechtfertigen. Sobald die Integration abgeschlossen ist, müssen sich Benutzer nicht mehr um die Laufzeitaspekte für die Ausführung der Workflow-Aufgabe kümmern:

- Es muss kein Computer eingerichtet werden, damit die Simulation erfolgreich ausgeführt werden kann
- Es ist nicht nötig, fehlende Bibliotheken oder einen Patch des Betriebssystems nachzutragen, damit die Ausführung der Applikation nicht abgebrochen werden kann

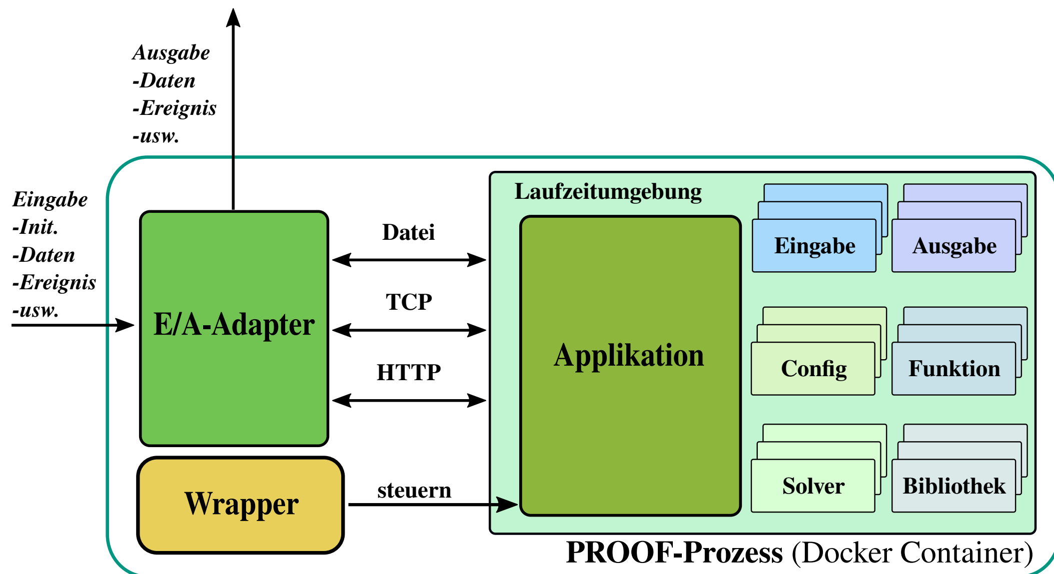

Abb. 4.8.: Integration einer Applikation in Docker-Container

- Die Ressourcennutzung zur Ausführung der Applikation muss nicht explizit gesteuert werden, sondern wird von PROOF übernommen
- Endbenutzer müssen nicht berücksichtigen, auf welchen Rechenknoten die Simulation des Workflows gestartet werden. Durch die Nutzung von containerisierten Microservices und Kubernetes können Applikationen einfach und flexibel in verschiedenen unabhängigen Simulationsknoten auf einem Computercluster ausgeführt werden
- Der Datenfluss und der Laufzeitstatus der Applikation bzw. Workflows kann mithilfe eines Webbrowsers für den Zugriff auf die PROOF-Weboberfläche überwacht werden
- Die Interaktion der Applikation mit externen Services wird durch die Verwendung von Adaptern und Redis in PROOF automatisiert durchgeführt, ohne dass Benutzer ihre ausführbare Applikation ändern oder Schnittstellen und Adapter zusätzlich implementieren müssen

4.4.2 Wrapper

In der Softwaretechnik ist das Wrapper(Adapter)muster ein Entwurfsmuster, das es ermöglicht, die Schnittstelle einer vorhandenen Klasse über eine andere Schnittstelle zu verwenden [FF04; Chl+22]. Es wird immer eingesetzt, um vorhandene Klassen mit nicht passenden Schnittstellen zu verwenden, ohne deren Quellcode zu ändern.

Der Hauptzweck des Wrappers besteht darin, eine Möglichkeit bereitzustellen, um wiederkehrende Entwurfsprobleme zu lösen. Außerdem kann somit flexible und wiederverwendbare objektorientierte Software entworfen werden, d.h. Objekte, die einfacher zu implementieren, zu ändern, zu testen und wiederzuverwenden sind [Gam+15]. Das Wrappermuster löst Probleme wie [w3s22]:

- Wie kann eine Klasse wiederverwendet werden, die keine Schnittstelle hat?
- Wie können Klassen mit inkompatiblen Schnittstellen zusammenarbeiten?
- Wie kann eine alternative Schnittstelle zur Kommunikation für eine Klasse bereitgestellt werden?

Die PROOF-Plattform wird so konzipiert, dass verschiedene Applikationen als voneinander unabhängige, aber über eine Kommunikationsinfrastruktur miteinander verknüpfte PROOF-Prozesse ausgeführt werden, um komplette wissenschaftliche Workflows aufzubauen und zu simulieren. Alle Applikationen haben ihre eigene Laufzeitumgebung und keine einheitliche Schnittstelle zur Kommunikationsinfrastruktur. Zudem ist es sehr komplex und zeitaufwendig, den Quellcode jeder Applikation zur Anpassung an die Kommunikationsinfrastruktur zu ändern. Zur Lösung dieser Probleme wird ein PROOF-Wrapper als ein generischer Adapter basierend auf [Gam+15] implementiert, um die Komplexität und den Aufwand einer Quellcodeänderung zu vermeiden. In Abb. 4.9 wird das Konzept des PROOF-Wrappers zur Erhöhung der Ausführungsflexibilität von Applikationen in Workflows dargestellt.

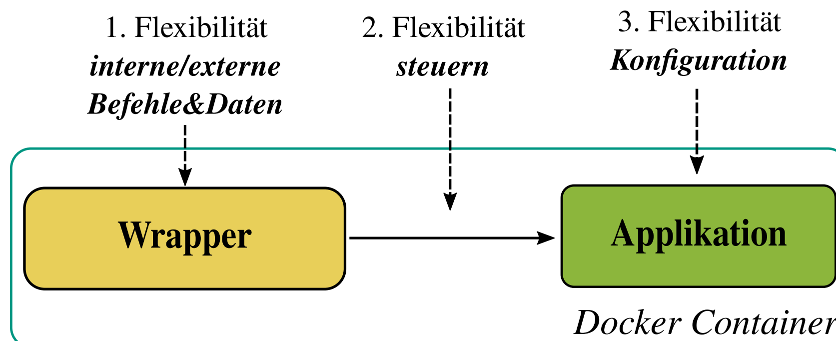

Abb. 4.9.: PROOF-Wrapper

Zur Steuerung der gekoppelten Applikation arbeitet der Wrapper als Controller. Neue Ansteuerungsinteraktionen mit der Applikation finden über den Wrapper statt. Damit wird Flexibilität dreiteilig bereitgestellt:

1. Der Wrapper unterstützt neue Befehle zur Ansteuerung der nachgelagerten Applikation, z.B. starten, anhalten, stoppen sowie die Einstellung statischer und dynamischer Parameter.

2. Zwei verschiedene Formen der Ablaufsteuerung werden vom Wrapper bereitgestellt:
   - Einmalige Ausführung
     Sobald eine neue Eingabe für eine Applikation vom Adapter bereitgestellt wird, wird die Applikation einmal vom Wrapper ausgeführt (siehe den oberen Teil von Abb. 4.10). Nachdem die Applikation ihre Ausgabe erzeugt hat, wird sie vom Wrapper gestoppt und ist wieder bereit für die nächste Ausführung mit neuen Eingaben.
   - In der Schleife (in the loop) Ausführung
     Beim Start eines Docker-Containers wird die entsprechende Applikation auch mit gestartet. Wie in Abb. 4.10 unten zu sehen ist, läuft die Applikation durchgehend und überwacht die Eingabedatei. Sobald die Eingabedatei geändert wird, wird von der Applikation eine Ausgabe durch Berechnung oder Simulation erstellt.
3. Durch Konfiguration sind Applikationen (z.B. Applikation A oder B in Abb. 4.10) vom Wrapper flexibel auszuwählen. D.h. mehrere Applikationen sind in einem Docker-Container oder im Volume (Dateisystem) zur Ausführung verfügbar. Gemäß der Konfiguration, die den Namen oder die Adresse einer Applikation beinhaltet, kann der PROOF-Wrapper die Applikationen finden und ausführen.

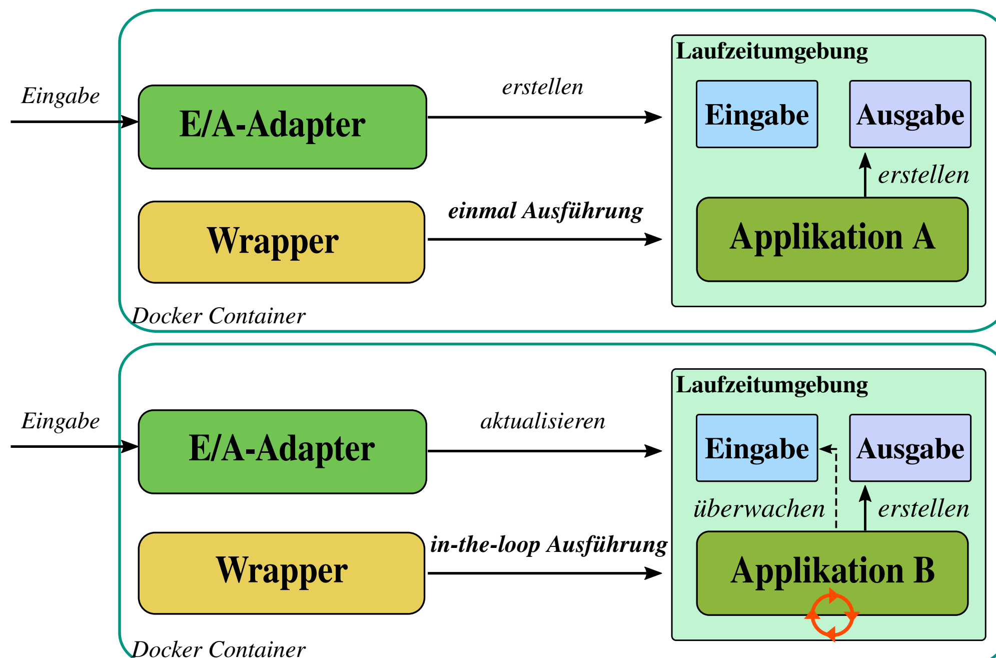

Abb. 4.10.: Ablaufsteuerung vom PROOF-Wrapper

4.4.3 Aufbau eines PROOF-Prozesses

Im Folgenden wird beschrieben, wie ein PROOF-Prozess Schritt für Schritt erstellt wird. Grundsätzlich sind die folgenden Schritte notwendig:

1. Vorbereiten einer ausführbaren Applikation wie z.B. Matlab-Modell, FMU-Modell, Python-Skript usw.
2. Anlegen einer Docker-Konfigurationsdatei (Dockerfile) zum Erstellen eines Docker-Images, das alle benötigten Installationen und Dateien zum Ausführen der Applikation enthält.
3. Erstellen des Docker-Images und speichern in einem Docker-Image-Repository.
4. Implementieren eines entsprechenden NiFi-Prozessors und Hinzufügen zur Prozess-Bibliothek.
5. Aufbauen, Konfigurieren und Ausführen des NiFi-Prozessors über die Weboberfläche.

Jeder der fünf Schritte wird in den folgenden Abschnitten detailliert erläutert.

1. Vorbereiten einer ausführbaren Applikation

In wissenschaftlichen Systemlösungen werden verschiedene Technologien, Programmiersprachen oder Software-Werkzeuge zur Modellierung, Simulation und Optimierung verwendet, z.B. in Energiesystemlösungen: C++, Java, Python, R [Fou22e; GCM22], **G**eneral **A**lgebraic **M**odeling **S**ystem (GAMS) [Wik22b; Gmb22; Ros22], Matlab, Julia [Jul22; Bez+12], Modelica [EOM98; Ass22c], Dymola [1] usw. In den meisten Programmiersprachen wie z.B. C++, Java, R, Python, GAMS oder Julia können Modelle oder Applikationen als eigenständig ausführbare Dateien exportiert werden, damit sie ohne Änderung direkt in PROOF integriert werden können. Die Modelle von Modelica und Dymola können als FMUs ausgegeben werden, die mithilfe des Python-Pakets FMPy [Sys22] in PROOF ausgeführt werden. Weitere Applikationsvorbereitungen werden in Anhang A.2 detailliert erläutert.

2. Anlegen einer Docker-Konfigurationsdatei (Dockerfile) zum Erstellen eines Docker-Images

Nach der Vorbereitung einer ausführbaren Applikation wird ein Docker-Image aufgebaut, um die verpackte eigenständige Applikation danach als einen PROOF-Prozess in Docker-Containern auszuführen. Dafür wird ein Dockerfile benötigt. Ein Dockerfile

[1] DYMOLA Systems Engineering: https://www.3ds.com/products-services/catia/products/dymola/

enthält ein Skript mit Anweisungen, die Docker zur Erstellung eines Container-Images verwendet.

Wie in Auflistung 4.1 beispielhaft dargestellt, umfasst dies im Detail die Anweisungen zur Installation einer Programmbibliothek (Zeile 2), von der die ausführbare Applikation abhängt. Zudem wird ein benötigter Ordner angelegt (Zeile 3), die für Eingabe, Zwischenergebnisse oder Ausgabe benutzt werden. Anschließend wird die Quellcodedatei der ausführbaren Applikation (Zeile 5) geladen. Dann werden die Adapter zur Interaktion mit der Applikation ausgewählt (Zeile 6-7), die selbst mit dem Framework angeboten werden. Zuletzt werden die Python-Skripte *RedisConsumer.py* (Zeile 8) und *RedisPublisher.py* (Zeile 9) zum Datenaustausch mit dem Message-Server *Redis* und *Wrapper.py* (Zeile 10) zur Steuerung der Applikation beigefügt. Das Kommunikations- und Steuerungssetup ist für alle ausführbaren Dateien identisch und muss nicht von Benutzern angepasst werden. Ein beispielhaftes vollständiges Dockerfile ist in Anhang A.3 zu finden.

```
1  FROM python:3
2  RUN pip3 install redis
3  RUN mkdir /data
4
5  ADD application.py /application.py
6  ADD FileAdapterInput.py /FileAdapterInput.py
7  ADD FileAdapterOutput.py /FileAdapterOutput.py
8  ADD RedisConsumer.py /RedisConsumer.py
9  ADD RedisPublisher.py /RedisPublisher.py
10 ADD Wrapper.py /Wrapper.py
11 ENTRYPOINT ["python3", "/Wrapper.py"]
```

Auflistung 4.1: Dockerfile zur Integration eines Python-Modells

3. Erstellen des Docker-Images

Unter Verwendung des im vorherigen Abschnitt angelegten Dockerfiles kann ein entsprechendes Docker-Image mit dem in Auflistung 4.2 dargestellten *build*-Befehl in der Kommandozeile erstellt werden. Zur Wiederverwendbarkeit wird das Docker-Image in einem Docker-Image-Repository abgespeichert. Danach kann das Prozessmanagement das Docker-Image aus dem Repository entnehmen, um die entsprechende Applikation in Docker-Containern auszuführen.

```
1  docker build -t name-of-the-image path-of-the-Dockerfile
```

Auflistung 4.2: Befehl zum Erstellen eines Docker-Images

Als ein wesentlicher Aspekt in Zusammenhang mit den Software-Domänenprozessen innerhalb des Workflow-Entwicklungszyklus (siehe Abb. 1.1) ermöglicht die Nutzung der Container-Virtualisierung des Frameworks die Integration aller Arten von Modellen und anderen ausführbaren Applikationen, unabhängig vom Betriebssystem oder

den verwendeten Programmiersprachen. Solange alle für die Ausführung erforderlichen Abhängigkeiten bereitgestellt werden, können Dockerfiles bzw. Docker-Images für jede Anwendung erstellt und in PROOF eingebettet werden. Durch die Nutzung des Frameworks ist es also nicht erforderlich, dass bereits bestehende Modelle bzw. Applikationen – bei Erfüllung der entsprechenden wissenschaftlichen Anforderungen – neu entwickelt werden müssen, nur weil sie nicht in die Software-Infrastruktur einer Forschungsorganisation integriert werden können. Dadurch kann bei jedem komplexen Modell, das zusätzlich als Blackbox behandelt werden kann und nicht nachkonstruiert werden muss, erheblicher Entwicklungs- und Implementierungsaufwand eingespart werden. Wenn jedoch kein geeignetes Modell verfügbar ist, muss ein neues Modell immer noch von Grund auf entwickelt werden. Aber auch in diesem Fall bietet PROOF den Vorteil, dass es indirekt ein hohes Maß an Freiheit bei der Wahl des Technologie-Stacks für die Modellimplementierung ermöglicht, da viele Programmiersprachen, Technologieumgebungen und Ressourcen innerhalb des Softwareimplementierungsprozesses verwendet werden können.

Neben der Möglichkeit, unterschiedliche ausführbare Dateien zu integrieren, muss der damit verbundene Aufwand diskutiert werden, um den Nutzen des Frameworks zur Unterstützung computergestützter Forschungsprozesse zu beschreiben. Wie in Abschnitt 4.4.3 erläutert, muss ein Dockerfile geschrieben werden, um eine ausführbare Datei für das Framework anwendbar zu machen. Dieser kurze Setup-Prozess kann von Benutzern mit wenig Programmiererfahrung abgeschlossen werden, unabhängig davon, ob es sich bei der ausführbaren Datei um eine Eigenentwicklung handelt oder nicht.

4. Implementierung eines NiFi-Prozessors

Im nächsten Unterkapitel 4.5 wird beschrieben, wie eine Weboberfläche basierend auf der Apache NiFi-GUI in PROOF integriert wird. Die Implementierung und Integration des Lebenszyklus [Tea22] eines NiFi-Prozessors findet sich in Abschnitt 4.5.1.

5. Aufbauen, Konfigurieren und Ausführen des NiFi-Prozessors

Auf Punkt 4 aufbauend beschreibt Abschnitt 4.5.2 den Aufbau, die Konfiguration eines NiFi-Prozessors und die Erstellung eines Workflows.

4.5 PROOF-Weboberfläche

Für den komfortablen, interaktiven Einsatz von PROOF wird eine benutzerfreundliche Weboberfläche in PROOF integriert, mit der wissenschaftliche Workflows grafisch aufgebaut, betrieben und verwaltet werden können. Alle integrierten Softwaremodelle oder Applikationen sind über eine Prozess-Bibliothek als PROOF-Prozesse verfügbar und können als NiFi-Prozessoren auf die web-basierte Arbeitsfläche gezogen werden, um Workflows visuell zu erstellen. Parameter, Eingaben und Ausgaben können für jeden Prozessor in Konfigurationsdialogen eingestellt werden. Ein zugehöriges Konzept definiert, welche funktionalen Mechanismen zum Aufbau von Workflows (Eingabe- und Ausgabe-Elemente sowie deren Verknüpfungen, Konfigurationsparameter von Applikationen, Synchronisationsmechanismen usw.) erforderlich sind und wie Workflows maschinenlesbar abgespeichert und eingelesen werden können.

Dieses Unterkapitel widmet sich dem Aufbau eines NiFi-Prozessors, indem alle Funktionen zur Definition des Lebenszyklus eines NiFi-Prozessors implementiert werden. Anschließend wird beschrieben, wie aufgebaute NiFi-Prozessoren auf der Weboberfläche benutzt, konfiguriert und miteinander verknüpft werden, um Workflows zu erstellen. Die Beschreibung einer XML-Datei, die zur Wiederverwendbarkeit alle Informationen eines erstellten Workflows umfasst, schließt das Unterkapitel ab.

4.5.1 Apache NiFi-Prozessor

Basierend auf Apache NiFi definiert die PROOF-Weboberfläche eine visuelle Benutzeroberfläche für den Aufbau eines Datenfluss-Workflows aus NiFi-Prozessoren, die einerseits von PROOF verwendet wird, um eine definierte Schnittstelle zwischen der Weboberflächen-Engine und Framework-Prozessen als Workflowkomponenten zu haben. Andererseits stellt ein NiFi-Prozessor ein grafisches Element für einen Framework-Baustein in der Arbeitsfläche des NiFi-Workflow-Editors dar. Das Element kann in der Prozess-Bibliothek ausgewählt und in die Arbeitsfläche gezogen werden, um einem Workflow visuell und logisch eine Komponente hinzuzufügen.

Abb. 4.11 veranschaulicht die Vorgehensweise zum Hinzufügen eines eigenen NiFi-Prozessors zur Prozess-Bibliothek innerhalb der NiFi-Umgebung:

1. Zuerst wird eine *Java-Prozessorklasse* (Java-Datei) geschrieben, welche die von NiFi vordefinierte Prozessor-Basisklasse erweitert.

2. Dann wird die neue Java-Datei mit den anderen Dateien als ein ***NiFi Ar**chive (nar)-Datei* zusammengepackt (siehe linke Seite von Abb. 4.11). Die *nar-Datei* wird zum Hauptartefakt des Projekts hinzugefügt für die Installation und Bereitstellung der NiFi-Prozessoren.
3. Anschließend aktualisiert die *nar-Datei* die *Prozessor-Bibliothek*, um die neue Prozessorklasse einzufügen.
4. Schließlich kann die neue Prozessorklasse als NiFi-Prozessor in der Prozessor-Bibliothek ausgewählt und auf der Benutzeroberfläche zum Erstellen von Workflows verwendet werden.

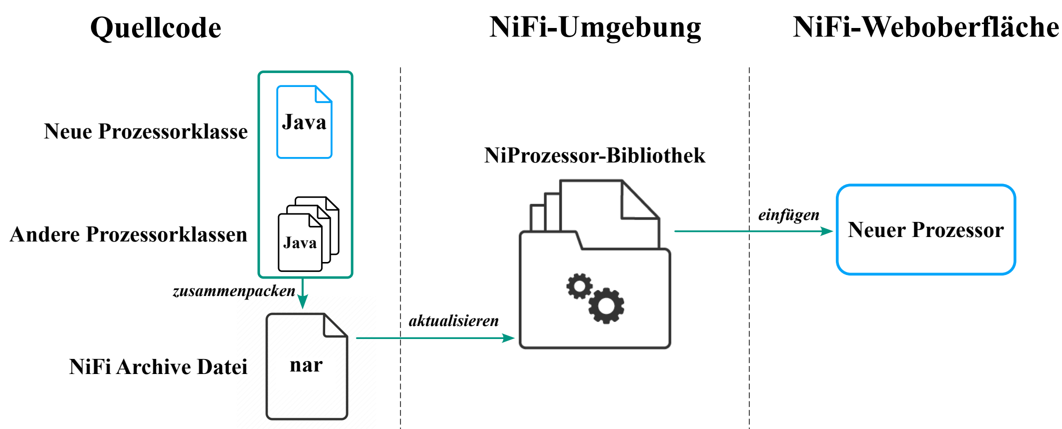

Abb. 4.11.: Hinzufügen eines eigenen NiFi-Prozessors zur Prozess-Bibliothek

Implementierung eines NiFi-Prozessors

Um einen neuen NiFi-Prozessor zur Prozess-Bibliothek hinzuzufügen, muss eine Java-Prozessorklasse implementiert werden, welche die Prozessor-Basisklasse *AbstractProcessor* erweitert und alle benötigten Attribute und Funktionen für den neuen Prozessor bereitstellt. Dazu werden das Vorgehen (Initialisieren, Triggern und Löschen) und die entsprechenden Methoden definiert und implementiert, um den Lebenszyklus eines NiFi-Prozessors zu beschreiben:

Initialisieren

Wie in Auflistung 4.3 dargestellt, wird zur Initialisierung eines NiFi-Prozessors die *init*-Methode (s. Zeile 4-7) mit einem *ProcessorInitializationContext-Objekt* als Argument zuerst aufgerufen. Diese Methode initialisiert die vordefinierten Attribute (Zeile 1) und Beziehungen (Zeile 2) zum Prozessor, die sich während der gesamten

Lebensdauer der Prozessorinstanz nicht ändern. Eine beispielhafte vollständige Implementierung der *init*-Methode wird in Anhang A.2 dargestellt.

```
1   Definiere alle Attribute
2   Definiere alle Beziehungen
3
4   FUNKTION init(ProcessorInitializationContext-Objekt)
5       Definiere eine Attribute-Liste und initialisiere die vordefinierten Attribute
6       Definiere eine Beziehungsliste und initialisiere die vordefinierten Beziehungen
7   End FUNKTION
```

Auflistung 4.3: Die init-Funktion eines NiFi-Prozessors

Wenn der Prozessor aktiviert werden soll (siehe Auflistung 4.4), wird die *onScheduled*-Methode jedes Mal aufgerufen. Zur Aktivierung wird ein Task-Objekt (Zeile 2) erstellt, das alle Informationen bezüglich des Prozessors sammelt (Zeile 3). Beispielsweise zeigt Auflistung 4.5 ein JSON-Objekt zur Definition eines Task-Objekts, um einen Prozessor in einem Docker-Container zu aktivieren:

- Eine eigene UUID, die den umgebenden Docker-Container identifiziert und benennt.
- Die Namen für Eingabe- und Ausgabe-Topics in Redis, welche die UUID enthalten.
- Der Name des zugehörigen Docker-Images *Image-Name*.
- Der Wert des Parameters *replicas*, der vom Nutzer bei der Konfiguration auf der Weboberfläche als *concurrent tasks* angegeben werden muss und der bezeichnet, wie viele Container für den Task vom Prozessmanagement parallel gestartet werden.
- Der Parameter *operation* mit dem Wert *create* zur Erstellung eines Docker-Containers.

```
1   FUNKTION onScheduled(ProcessContext-Objekt)
2       Definiere ein Task-Objekt
3       Sammel relevante Daten aus dem ProcessContext-Objekt und speichere sie im Task-Objekt
4       Sende das Task-Objekt an das Redis-Topic "PROOF_MAIN" zur Aktivierung eines Prozessors
5   End FUNKTION
```

Auflistung 4.4: Die onScheduled-Funktion

```
1   {
2       "id" : "UUID",
3       "inputTopic" : "inputs_container_UUID",
4       "outputTopic" : "outputs_container_UUID",
5       "dockerImage" : "imageName",
6       "replicas" : "n",
7       "operation" : "create"
8   }
```

Auflistung 4.5: Ein Task zur Aktivierung eines Prozessors

Der Aktivierungsprozess schließt damit ab, dass das Task-Objekt von einem *RedisPublisher* an das Redis-Topic *PROOF_MAIN* gesendet wird (siehe Zeile 4 in Auflistung 4.4), um das Starten des erforderlichen Docker-Containers im Hintergrund vorzubereiten. Eine beispielhafte vollständige Implementierung der *onScheduled*-Methode wird in Anhang A.3 dargestellt.

Triggern

Nach der Initialisierung wird ein Prozessor getriggert, wenn für den Prozessor Arbeit vorhanden ist. Dabei wird die Methode *onTrigger* aufgerufen, die die Logik zur Datenverarbeitung in Auflistung 4.6 definiert:

- Zeile 2
  Die Eingabedaten werden zuerst als ein *FlowFile* aus der *ProcessSession* aufgenommen und anschließend im Parameter *value* gespeichert.
- Zeile 3
  Danach werden die Eingabedaten mittels eines *RedisPublisher* über Redis an den entsprechenden Docker-Container einschließlich des Modells übermittelt.
- Zeile 4
  Anschließend wird ein *JedisSubscriber* in der *onTrigger*-Funktion erstellt, der das Output-Topic abonniert und abwartet, bis die Ausgabe vom Modell an das Topic zurückgesendet wird.
- Zeile 5
  Die Ausgabedaten werden zuletzt als ein *FlowFile* an den nächsten verknüpften NiFi-Prozessor transferiert.

```
1  FUNKTION onTrigger(ProcessContext-Objekt, ProcessSession-Objekt)
2      Lies Eingabestrom als ein FlowFile-Objekt und setze den Wert als einen Value-Parameter
3      Sende den Value-Parameter ueber Redis an den entsprechenden Container
4      Erstelle ein JedisSubscriber-Objekt zum Empfang der Ausgabe vom Container
5      Transportiere die Ausgabe als ein FlowFile an den naechsten verknuepften NiFi-Prozessor
6  End FUNKTION
```

Auflistung 4.6: Die onTrigger-Funktion

Eine beispielhafte vollständige Implementierung der *onScheduled*-Methode wird in Anhang A.4 dargestellt.

Löschen

Falls der NiFi-Prozessor nicht mehr benötigt und seine Ausführung nicht mehr geplant ist, wird die *onRemoved*-Methode aufgerufen, um den NiFi-Prozessor und seinen Docker-Container bzw. den PROOF-Prozess komplett zu entfernen. Wie in Auflistung 4.7 zu sehen ist, wird ein neuer Task erstellt und dem Parameter *operation* der

Wert *delete* übergeben (Zeile 2). Der Task wird dann an das Redis-Topic *PROOF_MAIN* gesendet (Zeilen 4), um das Löschen des PROOF-Prozesses im Hintergrund durchzuführen. Eine beispielhafte vollständige Implementierung der *onScheduled*-Methode wird in Anhang A.5 dargestellt.

```
1  FUNKTION onRemoved(ProcessContext−Objekt)
2      Definiere ein Task−Objekt
3      Sammel relevante Daten aus dem ProcessContext−Objekt und speichere sie im Task−Objekt
4      Sende das Task−Objekt an das Redis−Topic "PROOF_MAIN" zum Loeschen eines Prozessors
5  End FUNKTION
```

Auflistung 4.7: Die onRemoved-Funktion

Nachdem solche Funktionen zur Definition des Lebenszyklus eines NiFi-Prozessors in einer Prozessorklasse implementiert und der Prozessorname in der von NiFi bereitgestellten Datei „*org.apache.nifi.processor.Processor*" eingefügt wurden, kann eine kompilierte Version der Prozessorklasse exportiert werden. Wie Abb. 4.11 bereits illustriert, wird die Version dann der Prozess-Bibliothek zur Wiederverwendbarkeit hinzugefügt. Dadurch kann das Framework problemlos um benutzerdefinierte Prozessoren mit eigenen Funktionen erweitert werden, um verschiedene komplexe Anwendungsfälle zu simulieren.

4.5.2 Weboberfläche

Nachdem eine Applikation mit einem Dockerfile in ein Docker-Image eingepackt und ein entsprechender NiFi-Prozessor dafür implementiert und in der Prozess-Bibliothek eingefügt wird, steht der NiFi-Prozessor schließlich auf der PROOF-Weboberfläche zur Verfügung.

Der vorliegende Abschnitt stellt zuerst die Funktionalitäten der Weboberfläche vor. Anschließend wird die Vorgehensweise zum interaktiven Aufbau eines Workflows beschrieben. Der Abschnitt schließt mit einer Beschreibung einer E**x**tensible **M**arkup **L**anguage (XML)-Datei, die alle Informationen eines aufgebauten Workflows speichert und als eine Vorlage zum Aufbau des Workflows auf der Weboberfläche wieder importiert werden kann.

Übersicht

Die Benutzeroberfläche von Apache NiFi bietet Mechanismen zum Erstellen, Visualisieren, Bearbeiten, Überwachen und Verwalten automatisierter Datenflüsse. Die

Benutzeroberfläche kann in mehrere Segmente unterteilt werden, die jeweils für unterschiedliche Funktionen der Anwendung verantwortlich sind. In Abb. 4.12 wird die gesamte Weboberfläche illustriert:

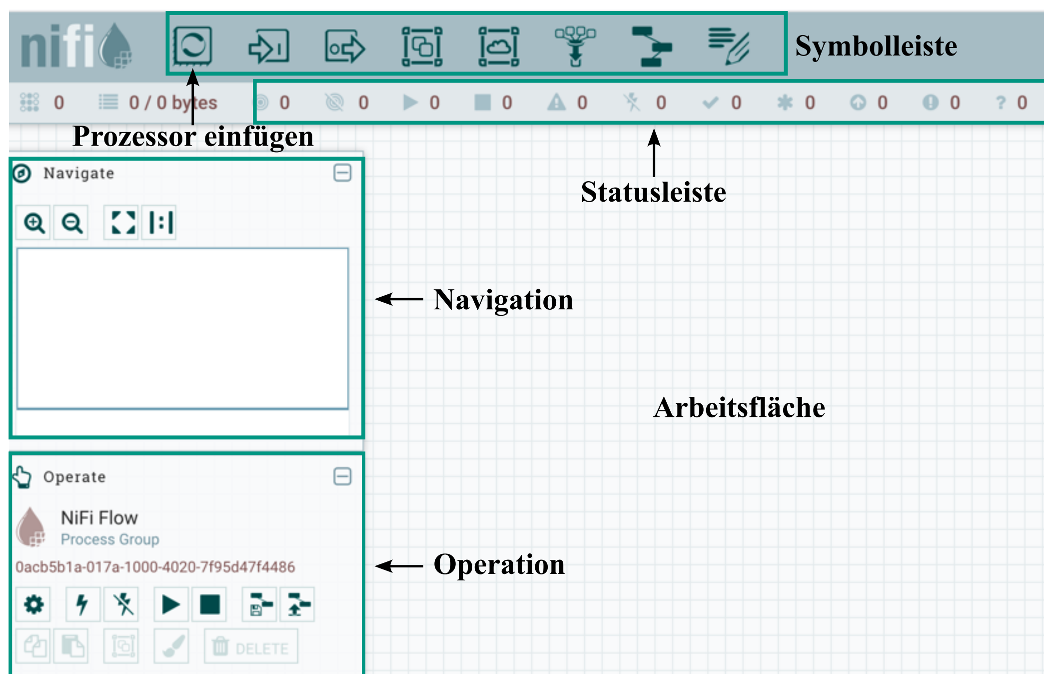

Abb. 4.12.: Apache NiFi Weboberfläche

- Die **Komponentensymbolleiste** wird im obersten Bereich der Weboberfläche ausgeführt. Sie besteht aus den Komponenten, die auf die Arbeitsfläche gezogen werden können, um Datenflüsse und Workflows zu erstellen. Die erste Komponente bietet die wichtige Funktion, Prozessoren in die Arbeitsfläche einzufügen.
- Die **Statusleiste** befindet sich unter der Komponentensymbolleiste. Die Statusleiste enthält Informationen über die Anzahl der Prozessoren in verschiedenen Status, die derzeit aktiv, gestoppt, ungültig oder deaktiviert sind, die Datenmenge, die derzeit vorhanden ist, die Anzahl versionierter Prozessgruppen in jedem Status (aktuell, lokal geändert, veraltet, Synchronisierungsfehler) und einen Zeitstempel, zu dem alle diese Informationen zuletzt aktualisiert wurden.
- Darüber hinaus verfügt die Benutzeroberfläche über einige Funktionen zur **Navigation**. Damit kann die Arbeitsfläche gezoomt, gescrollt und positioniert werden.
- Die **Operationsfunktionen** befinden sich auf der linken Seite unter der Navigation. Der Bereich besteht aus Buttons, die zum Verwalten des Datenflusses und

des Workflows verwendet werden können. Beispielsweise können Workflows damit gestartet, gestoppt, konfiguriert oder gespeichert werden.

Prozess-Bibliothek

Um einen Prozessor auf der Arbeitsfläche zu positionieren, wird ein Dialogfenster der Prozess-Bibliothek angezeigt (Abb. 4.13). Im linken Bereich werden verschiedene Stichwörter zum Herausfinden gewünschter Prozessoren angezeigt. Diese Stichwörter können beispielsweise python, storage, oder andere sein. In der oberen rechten Ecke kann ein Suchbegriff angegeben werden, um eine Liste von gewünschten Prozessoren basierend auf dem Suchbegriff zu filtern. Beispielsweise wird eine Liste von Prozessoren bezüglich des Stichworts „*Task*“ in Abb. 4.13 gefunden und dargestellt. Danach kann der benötigte Prozessor ausgewählt und in die Arbeitsfläche der Weboberfläche eingefügt werden. Alle Prozessoren der Prozess-Bibliothek können als Komponenten zum Aufbau der Workflows für verschiedene Forschungsszenarien wiederverwendet werden.

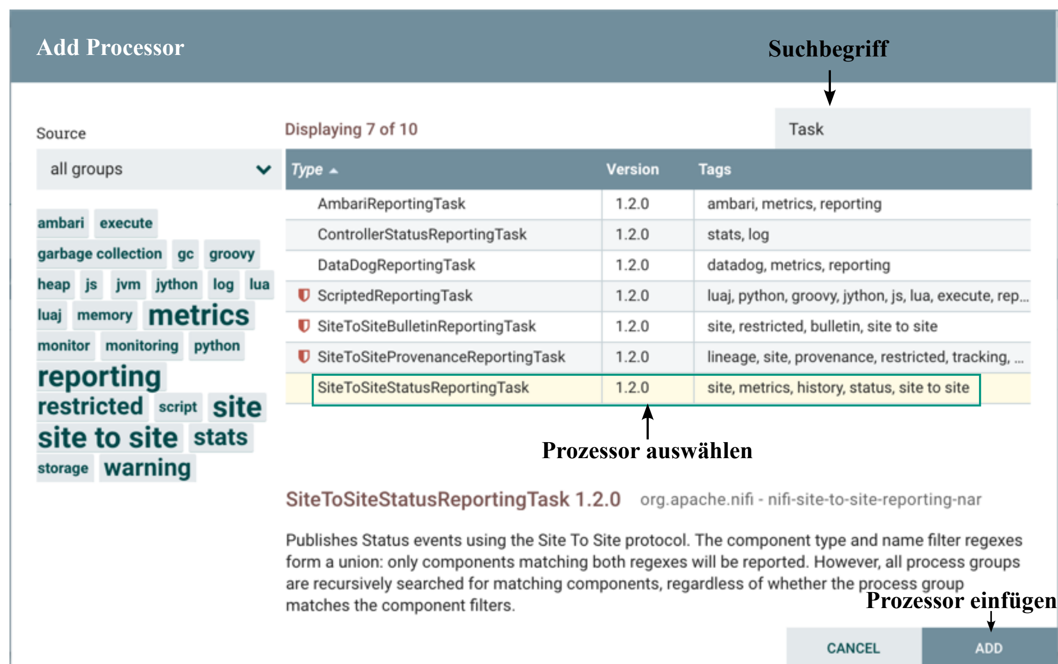

Abb. 4.13.: Prozess-Bibliothek

Konfiguration eines Prozessors

Die Weboberfläche stellt ein Dialogfenster bereit, um einen Prozessor zu konfigurieren. Wie in Abb. 4.14 zu sehen ist, hat das Dialogfenster vier untergeordnete

Registerkarten (Tabs), nämlich *SETTINGS*, *SCHEDULING*, *PROPERTIES* und *COMMENTS*. Im SETTINGS-Dialogfeld werden die Stammdaten des Prozessors, z.B. der Name, die ID, der Typ usw., angezeigt. In der SCHEDULING-Registerkarte wird ein Ablaufplan festgelegt (siehe Abb. 4.14). Dabei werden drei mögliche Optionen geboten:

- **Zeitgesteuert** (Timer driven)
  Der Prozessor wird so geplant, dass er in regelmäßigen Intervallen ausgeführt wird.
- **Ereignisgesteuert** (Event driven)
  Der Prozessor wird durch ein Ereignis ausgelöst.
- **CRON-gesteuert** ([HOS23])
  Der Prozessor wird so geplant, dass er nach einer vordefinierten CRON-Ablaufplanung regelmäßig ausgeführt wird.

Abb. 4.14.: Ablaufplan Konfiguration

Darüber hinaus bietet dieses Dialogfeld eine Konfigurationsoption mit dem Namen „*Concurrent Tasks*“. Hiermit wird gesteuert, wie viele Container der Prozessor parallel verwendet. Anders ausgedrückt: Eine Steuerung, wie viele Daten gleichzeitig von diesem Prozessor in Containern verarbeitet werden sollen. Durch Erhöhung dieses Werts kann der Prozessor mehr Daten in derselben Zeit verarbeiten. Die Funktionalität zur Parallelisierung und Koordination von Workflows wird in Kapitel 6 ausführlich beschrieben.

Im PROPERTIES-Tab können die Attribute und Parameter des Prozessors konfiguriert werden. Alle in der erweiterten NiFi-Prozessorklasse vordefinierten Attribute und Parameter werden in diesem Tab aufgelistet. Wie in Abb. 4.15 dargestellt, können die Parameter wie *Input File* und *Output File* eingegeben und eingestellt werden.

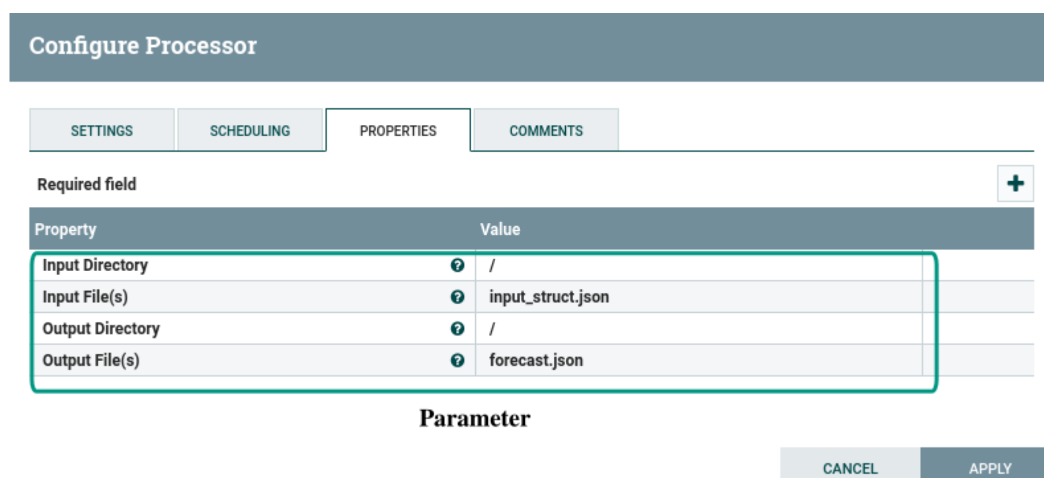

Abb. 4.15.: Konfiguration der Parameter

Die COMMENTS-Registerkarte bietet schließlich einen einfachen Bereich, in dem beliebige Kommentare eingefügt werden können, die für diese Komponente geeignet sind. Die Verwendung der Registerkarte ist optional.

Aufbau eines Workflows

Nachdem alle Prozessoren eines Workflows eingefügt, konfiguriert und zur Erstellung von Datenflüssen miteinander verknüpft wurden, ist der Aufbau des Workflows abgeschlossen. Auf der Weboberfläche in Abb. A.6 in Anhang A.5 wird ein Workflow exemplarisch veranschaulicht. Mithilfe der Operationsfunktionen kann der Workflow einfach gestartet, gestoppt, verwaltet und als XML-Datei gespeichert werden.

XML-Vorlage

Für eine immer detailliertere und genauere Analyse und Untersuchung wissenschaftlicher Probleme werden größere und komplexere Systemlösungen oder -modelle benötigt. Solche Systemlösungen oder -modelle bestehen aus zahlreichen Komponenten, die im Rahmen dieser Arbeit als wissenschaftliche Applikationen betrachtet werden. Wie Abb. A.6 zeigt, können auch sehr große und komplexe Workflows in PROOF erstellt werden, z.B. wird der Workflow in Abb. A.6 aus 20 Komponenten gebildet. Der Aufbau dieses Workflows oder noch größerer und komplexerer Systeme ist häufig fehlerbehaftet und zeitaufwändig für den Nutzer, denn viele Komponenten müssen eingefügt, konfiguriert und zur Erstellung von Datenflüssen miteinander gekoppelt werden. Außerdem kann die Verwendung kleiner Prozess-Bausteine bzw. großer Workflows mühsam werden, wenn die gleiche Logik für unterschiedliche

Simulationsszenarien mehrmals wiederholt werden muss. Daher spielt die Wiederverwendbarkeit von Workflows eine wichtige Rolle in der vorliegenden Arbeit. Dafür wird eine XML-Vorlage (Template) von PROOF bereitgestellt, mit dem erstellte Workflows als XML-Dateien exportiert und mit anderen Forschenden geteilt werden können. Für andere Anwendungsfälle können vorhandene Vorlagen in PROOF direkt importiert werden, um Workflows wiederzuverwenden.

Sobald ein Workflow erstellt wurde, kann er als Vorlage gespeichert werden. Wie in Abb. A.6 markiert, kann eine Vorlage durch das Klicken des Buttons „*Create Template*“ erstellt und heruntergeladen werden. Daneben bietet der Button „*Upload Template*“ die Funktion zum Hochladen und Importieren verfügbarer Workflow-Vorlagen. In Anhang A.6 werden Inhalt und Struktur einer beispielhaften XML-Vorlage zur Speicherung eines Workflows detailliert beschrieben.

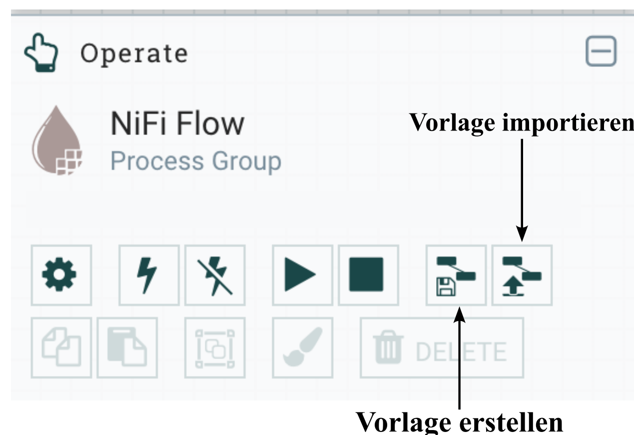

Abb. 4.16.: Operation zum Erstellen und Importieren einer XML-Workflow-Vorlage

4.6 Zusammenfassung

Im vorliegenden Kapitel wurde der erste erarbeitete wissenschaftliche Beitrag vorgestellt. Dabei wurde zunächst die Architektur der Co-Simulation zur Vorbereitung der PROOF-Architektur und eine technische Ansicht ihrer Komponenten erläutert. Im Anschluss daran schilderte das Kapitel die Konzeption der PROOF-Architektur schrittweise.

Danach fokussierte sich das Kapitel darauf, wie die PROOF-Architektur in eine containerisierte Microservice-Architektur überführt wird. Dabei wurden alle Komponenten der PROOF-Architektur als Microservices in Docker-Containern separat verpackt und ausgeführt.

Anschließend wurde die zentrale Fragestellung geklärt, wie eine wissenschaftliche Applikation mit ihren Abhängigkeiten und Schnittstellen vorbereitet und wie ein entsprechendes Docker-Image zum Kapseln der Applikation mitsamt ihren Abhängigkeiten und Schnittstellen sowie E/A-Adaptern angelegt werden kann. Außerdem wird erklärt, wie ein korrespondierender NiFi-Prozessor implementiert wird, damit die Applikation in PROOF als ein PROOF-Prozess automatisiert ausgeführt werden kann.

Zum Schluss wurde erklärt, dass eine Weboberfläche basierend auf Apache NiFi in PROOF erweitert und integriert wird. Auf der Weboberfläche repräsentieren NiFi-Prozessoren Framework-Prozesse visuell, wodurch wissenschaftliche Workflows anschaulich dargestellt werden können. Dabei wurde die Implementierung der Funktionen zur Beschreibung des Lebenszyklus eines Prozessors erläutert. Schritt für Schritt wurde beschrieben, wie NiFi-Prozessoren auf der Weboberfläche hinzugefügt, konfiguriert und gekoppelt werden, um einen Workflow aufzubauen, der automatisiert gesteuert werden kann. Zudem wurde eine XML-Vorlage im Appendix A.6 vorgestellt, mit der aufgebaute Workflows als XML-Dateien exportiert und dann zur Wiedererstellung der Workflows importiert werden können.

Zusammenfassend werden die nachstehenden Anforderungen 3.1.2 des ersten Beitrags erfüllt:

A1 **Hohe Konfigurierbarkeit**
Durch Erweiterung der vorgegebenen Prozessor-Basisklasse können neue Prozessoren mit ihren eigenen Eigenschaften und Funktionen für unterschiedliche wissenschaftliche Applikationen (Simulatoren, Prognose-Werkzeuge, wiss. Berechnungstools, Optimierer, usw.) implementiert werden. Ihre Eigenschaften und Parameter können mithilfe der Weboberfläche konfiguriert werden.

A4 **Unterstützung zum Ausführen großer und komplizierter Workflows**
Unter Verwendung der Container-Orchestrierungstechnologie kann das Prozessmanagement umfangreiche Framework-Prozesse in Containern in Pods als Microservices auf dem Computing-Cluster verteilen und ausführen. Dies unterstützt die Ausführung großer und komplexer Workflows und Simulationen aus einer Vielzahl von Komponentenmodellen, die parallel zueinander arbeiten können.

A5 **Automatisierung**
Durch Überführung des PROOF-Konzepts in eine containerisierte Microservice-Architektur können wissenschaftliche Applikationen mit ihren Laufzeitinfrastrukturen vollautomatisiert ausgeführt und gemanagt werden.

A6 Plattformunabhängige grafische Oberfläche
Durch Erweiterung und Integration einer Weboberfläche ist PROOF über Webbrowser plattformunabhängig verwendbar.

A7 Wiederverwendbarkeit
PROOF unterstützt eine Prozess-Bibliothek, welche die integrierten wissenschaftlichen Applikationen mit ihren Abhängigkeiten als Framework-Prozesse zur Verfügung stellt. Daraus können Framework-Prozesse zum Aufbau von Workflows und Simulationen in verschiedenen Applikationskontexten wiederverwendet werden. Außerdem stellt PROOF eine XML-Vorlage zur Speicherung und Wiederverwendung der Workflows bereit.

A9 Generische Funktionalitäten
Der PROOF-Wrapper wird als Hilfskomponente für generische Funktionalität zur dynamischen Steuerung des gekoppelten Bausteins bereitgestellt, um die Ausführungsflexibilität von Bausteinen in Workflows zu erhöhen.

5 Konfigurierbare Kommunikation zwischen Framework-Prozessen

Während im vorherigen Kapitel die grundlegende Struktur der PROOF-Architektur vorgestellt wurde, beschreibt das vorliegende Kapitel gemäß den Anforderungen 3.1.2 (A1, A2, A3, A9), wie sich die konfigurierbare Kommunikation zwischen Framework-Prozessen innerhalb von PROOF realisieren lässt. Dadurch wird ermöglicht, dass Forschende ihre wissenschaftlichen Applikationen unmittelbar auf der Weboberfläche konfigurieren und parametrisieren können, ohne die Quellcodes der Applikationen anzupassen und neue Schnittstellen zum Datenaustausch zu implementieren. Weiterhin wird in diesem Kapitel beschrieben, welche konfigurierbare Services von PROOF zur Implementierung unterschiedlicher Anwendungsfälle bereitgestellt werden.

Das Kapitel ist wie folgt strukturiert: Abschnitt 5.1 behandelt den Vergleich verschiedener Kommunikationsschnittstellen in Bezug auf die Anforderung 3.1.1 der vorliegenden Arbeit. Auf das Ergebnis des Vergleichs stützt sich dann in Abschnitt 5.2 die Ableitung des Konzepts in Bezug auf die Datenflussübertragung zwischen Framework-Prozessen. Dabei wird geklärt, wie die Komponenten des Konzepts zur Unterstützung konfigurierbarer Kommunikation innerhalb von PROOF implementiert werden. Anschließend findet sich die Vorstellung hinsichtlich des Konzepts für Volumes in Abschnitt 5.3. Ein Volume wird von einem verteilten Dateisystem bereitgestellt und kann von allen Containern gemeinsam geteilt und verwendet werden, um den Austausch großer Datenmengen und die Bereitstellung von großer Applikationen in Form von Dateien zu ermöglichen. Nachfolgend fokussiert Abschnitt 5.4 die Beschreibung unterschiedlicher konfigurierbarer Services, die von PROOF zur Realisierung verschiedener Anwendungsfälle bereitgestellt werden. Das Kapitel endet mit Abschnitt 5.5, der die in Abschnitt 3.1.2 vorgestellten Anforderungen zusammenfasst.

5.1 PROOF-Kommunikationsinfrastruktur

Nachdem mit Abschnitt 2.2.4 eine Vorstellung verschiedener Kommunikations- und Koordinierungsdienste, wie z.B. Apache Zookeeper, Apache Kafka, RabbitMQ und Redis, gegeben wurde, widmet sich dieser Abschnitt dem Vergleich der Funktionalitäten der Kommunikationsschnittstellen basierend auf der Anforderung 3.1.1 dieser Arbeit. Damit wird geklärt, ob welche Kommunikationsschnittstelle besser für PROOF geeignet ist.

Für größere transdisziplinäre, domänenübergreifende Datenverarbeitungs- und Co-Simulations-Workflows ist der Datenaustausch zwischen ausführbaren Applikationen sowie eine gewisse Koordinationslogik unerlässlich. Dafür ist eine verteilte Kommunikationsinfrastruktur in PROOF integriert, um die Kommunikation zwischen den Framework-Prozessen zu ermöglichen und Mittel zur Koordinierung bereitzustellen (s. Abb. 4.6). Redis und Apache Kafka, die beide über eine effiziente und sehr skalierbare Messaging-Funktionalität verfügen, können als Kommunikationsschnittstelle für PROOF verwendet werden und wurden hinsichtlich ihrer Bereitstellung der Nachrichtenfunktionalität getestet.

Unter Anforderung A3: Performanter Datenaustausch sind vier Unteranforderungen zu verstehen:

A3.1 In-Memory-Nachrichtenbroker als Kommunikationsschnittstelle

A3.2 Schnelles Lesen und Speichern von Daten

A3.3 Keine Notwendigkeit, Daten dauerhaft auf der Festplatte zu speichern

A3.4 Austausch von geringen Datenmengen über die Kommunikationsschnittstelle (für größere Datenmengen wird ein Volume (s. Abschnitt 5.3) eingesetzt)

Wie in Tab. 5.1 zu sehen ist, unterscheidet sich Redis in Bezug auf Datenspeicherung und verschiedene Funktionalitäten etwas von den anderen drei Kommunikationsschnittstellen gemäß den vier Unteranforderungen. PROOF erfordert eine schnelle Datenverarbeitung in Echtzeit mit geringem Kommunikations- oder E/A-Overhead. Im Kern ist Redis ein reiner In-Memory-Datenspeicher, der seinen Hauptspeicher für die Datenspeicherung und -verarbeitung verwendet, was ihn beim Lesen und Speichern von Daten viel schneller als die festplattenbasierten Apache Zookeeper, Apache Kafka und RabbitMQ macht [Ltd22b]. Darüber hinaus kann Redis als Hochleistungsdatenbank und Nachrichtenbroker verwendet werden, was den Unteranforderungen besser entspricht, da die meisten Daten in PROOF schnell übertragen werden müssen und es nicht notwendig ist, die Daten persistent auf

einer Festplatte zu speichern. Außerdem bieten Redis and RabbitMQ Push-basiertes Publish-/Subscribe-Messaging, während das Publish-/Subscribe-Messaging von Apache Kafka und Apache Zookeeper Pull-basiert ist. Das bedeutet, dass Nachrichten, die in Redis/RabbitMQ veröffentlicht werden, automatisch sofort an die Abonnenten zugestellt werden, während Nachrichten in Apache Kafka/Zookeeper nur auf Anfrage/Nachfrage der Verbraucher an die Verbraucher weitergegeben werden. Für diesen Fall, dass Verbraucher Daten nicht schnell genug verarbeiten können, muss Apache Kafka/Zookeeper Daten von einer Festplatte und nicht aus dem Memory lesen, was seine Leistung verlangsamt. Des Weiteren werden im Framework in den meisten Fällen kleine, kurzlebige Nachrichten versendet/ausgetauscht. Redis kann mithilfe eines In-Memory-Speichers die kleinen Nachrichten innerhalb der Redis zugewiesenen Speichergrenze speichern. Da der von Redis verwendete Arbeitsspeicher jedoch typischerweise kleiner als eine Festplatte ist, ist es notwendig, ihn regelmäßig zu leeren und Platz für neue Daten zu schaffen.

Tab. 5.1.: Vergleich verschiedener Kommunikationsschnittstellen basierend auf Anforderungen A3.1-A3.4: ○-nicht erfüllt; ◐-teilweise erfüllt; ●-vollständig erfüllt.

Kommunikationsschnittstellen	A3.1	A3.2	A3.3	A3.4
Apache Zookeeper	○	○	◐	●
Apache Kafka	○	○	◐	●
RabbitMQ	○	◐	◐	●
Redis	●	●	●	●

Zusammenfassend ist Redis daher im Vergleich zu den anderen Schnittstellen als eine generische nachrichtenorientierte In-Memory-basierte Kommunikationsschnittstelle die bessere Lösung für PROOF, um die Kommunikation zwischen den ausführbaren Applikationen und den E/A-Adaptern herzustellen (siehe Abb. 4.6).

5.2 Datenfluss zwischen Framework-Prozessen

Auf Redis als Kommunikationsschnittstelle für PROOF gestützt, wird das Konzept in Bezug auf die Datenübertragung zwischen Framework-Prozessen in diesem Abschnitt 5.2 erläutert. Dabei werden die Implementierungsdetails jeder Komponente zur Unterstützung konfigurierbarer Kommunikation innerhalb von PROOF vorgestellt.

5.2.1 Übersicht

In Abb. 5.1 wird eine Übersicht des Konzepts im Hinblick auf den Datenfluss eines Workflows aus zwei NiFi-Prozessoren ausführlich illustriert:

1. Zuerst werden die zwei NiFi-Prozessoren aus der Prozess-Bibliothek auf den Arbeitsbereich der Weboberfläche gezogen, konfiguriert sowie gekoppelt, um einen Workflow zu erstellen. Nachdem der Workflow gestartet wird, sendet der erste **NiFi-Prozessor** einen *Task*, der eine generierte *Task-UUID* und den Namen des entsprechenden *Docker-Images* beinhaltet, im JSON-Format über die **NiFi-Prozessorschnittstelle** an das *Task-Topic* von **Redis**.
2. Das **Prozessmanagement** abonniert das *Task-Topic* in **Redis** und nimmt daher den eingehenden Task auf.
3. Anschließend startet das **Prozessmanagement** einen **Framework-Prozess**, der die entsprechende *Applikation* mitsamt ihrer Schnittstelle und Laufzeitumgebung sowie den entsprechenden *E/A-Adaptern* in einem Docker-Container umfasst.
4. Wenn der **NiFi-Prozessor** eine Eingabe empfängt, wird die Eingabe als ein JSON-Objekt über die **Prozessschnittstelle** an das *Eingabe-Topic* gesendet.
5. Danach empfängt der *Eingabe-Adapter* die Eingabedaten aus dem Topic in **Redis**.
6. Dann werden alle Eingabedaten für die *Applikation* vom *Adapter* z.B. in Dateien oder als HTTP-Request vorbereitet.
7. Gemäß der Eingabedaten wird die *Applikation* ausgeführt.
8. Nach der Berechnung oder Simulation wird von der *Applikation* eine Ausgabe im Ausgabeordner erstellt.
9. Der *Ausgabe-Adapter* liest die Ausgabe.
10. Mithilfe des *Ausgabe-Adapters* werden die Ausgabedaten an das *Ausgabe-Topic* geschickt.
11. Danach erhält der **NiFi-Prozessor** die Ausgabedaten.
12. Schließlich wird die Ausgabe als Eingabe an den nächsten **NiFi-Prozessor** weiter transportiert.

Sobald alle Framework-Bausteine des Workflows mit diesem Datenfluss durchgeführt wurden, ist die Simulation des Workflows einmal durchgelaufen.

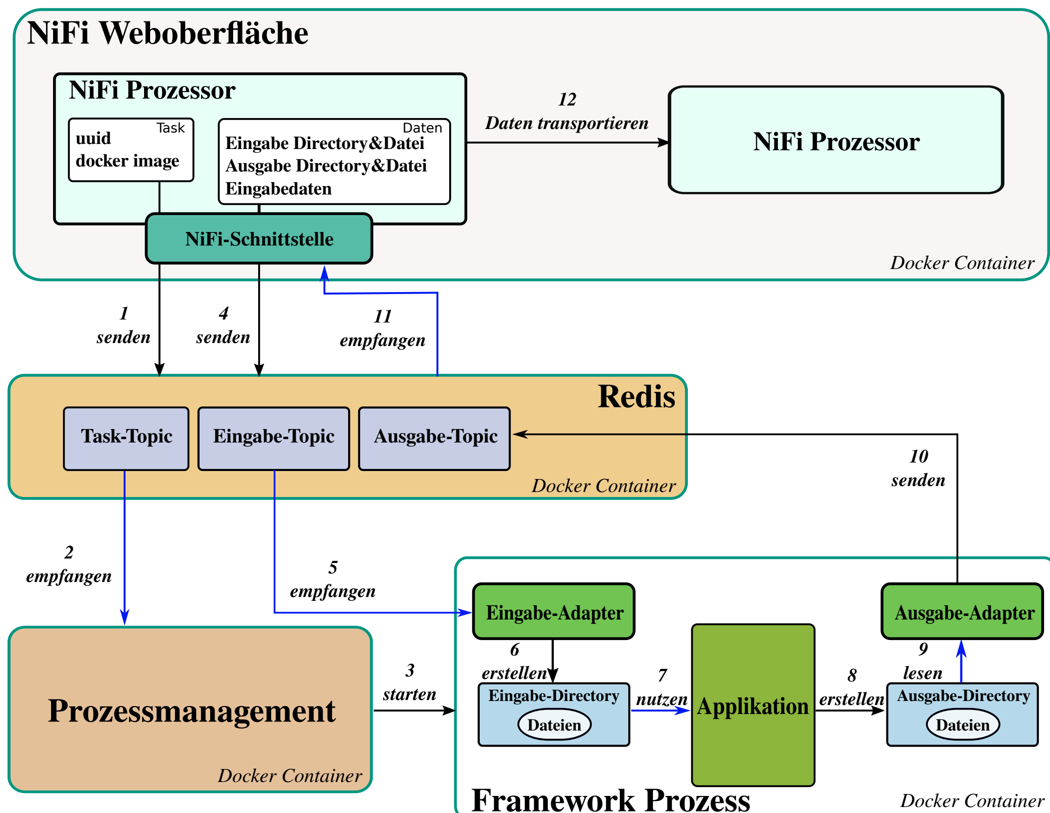

Abb. 5.1.: Datenfluss eines Workflows für zwei NiFi-Prozessoren

5.2.2 Parameter definieren

Für eine hohe Konfigurierbarkeit des Frameworks können Eigenschaften zur Parametrisierung und Anpassung des Verhaltens eines Framework-Prozesses zu einem NiFi-Prozessor hinzugefügt werden, die vom Workflow-Editor oder Benutzer über die Benutzeroberfläche in einem Konfigurationsdialog konfiguriert werden. Wie in Abb. 4.15 zu sehen ist, bietet PROOF eine PROPERTIES-Registerkarte auf der Weboberfläche zur Konfiguration der Attribute und Parameter jedes Prozessors. Alle in diesem Tab aufgelisteten Attribute und Parameter werden in der erweiterten NiFi-Prozessorklasse vordefiniert. Auflistung 5.1 zeigt exemplarisch die Definition der Parameter *INPUT_FILE* und *OUTPUT_FILE* in der Prozessorklasse als statische Attributen des Typs *PropertyDescriptor*, die anschließend als unveränderliche Liste festgelegt werden (s. Zeile 1). Die Definition eines Parameters umfasst einen Namen, eine Beschreibung dazu, eine Validierungslogik und einen Indikator dafür, ob der Parameter erforderlich ist, damit der Prozessor gültig ist oder nicht. Verschiedene Validierer können für einen Prozessor verwendet werden, um sicherzustellen, dass der eingegebene Wert in der Weboberfläche für einen Parameter gültig ist.

```
1  Definiere PropertyDescriptors INPUT_FILE und OUTPUT_FILE (Namen, Beschreibung und Validierungslogik)
2  FUNKTION init(ProcessorInitializationContext−Objekt)
3      Definiere eine PropertyDescriptor−Liste und initialisiere die zwei PropertyDescriptor−Objekte
4  End FUNKTION
```

Auflistung 5.1: Parameter definieren

Sobald die vordefinierten Parameter in der Liste von Parametern *descriptors* eingefügt wurden (siehe Zeile 3), sind die Parameter für Nutzer verfügbar zur Konfiguration (eine vollständige Implementierung zur Definition von Parametern wird in Anhang A.7 dargestellt). So können Nutzer über die Benutzeroberfläche Parameter eines Prozessors bzw. der entsprechenden Applikation flexibel einstellen, ohne den Quellcode der Applikation anpassen oder eine zusätzliche Konfigurationsdatei zur Parametrierung anlegen zu müssen (siehe Abb. 5.2).

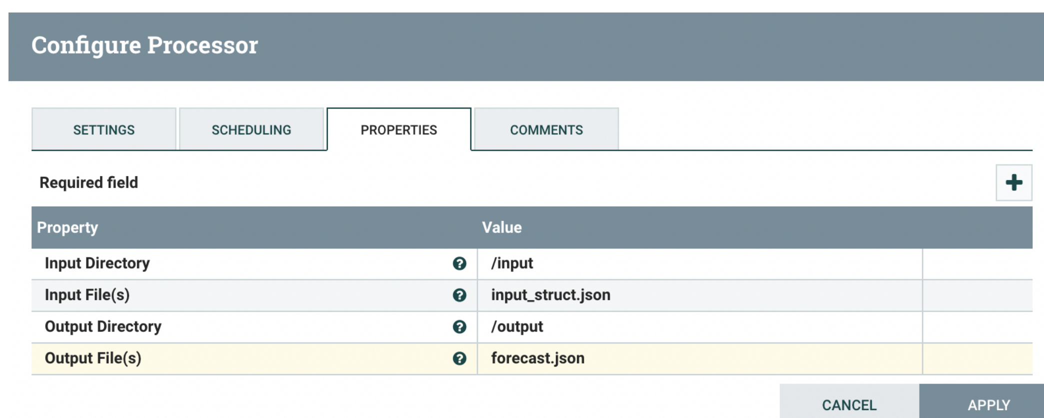

Abb. 5.2.: Konfigurationsdialogfenster

Neben der Konfiguration einer einzelnen Datei als Eingabe oder Ausgabe (siehe die Eingabedatei *input_struct.json* und die Ausgabedatei *forecast.json* in Abb. 5.2), bietet PROOF zur Vereinfachung auch die Funktion, mehrere Dateien in einem Ordner als Eingabe oder Ausgabe zu lesen oder zu schreiben. Dafür wird das Zeichen „*“ (Asterisk) verwendet. Wenn der Eingabeparameter beispielsweise mit „*.json“ konfiguriert wird, werden alle JSON-Dateien als Eingabedateien verarbeitet und beim Wert „Bus*“ des Ausgabeparameters werden alle Dateien mit dem Präfix „*Bus*“ als Ausgabedateien betrachtet.

5.2.3 Parameterwert übertragen

Nach der Konfiguration der Parameter eines Prozessors über die Benutzeroberfläche müssen ihre Werte an den entsprechenden PROOF-Prozess und die ausführbare Applikation zur Laufzeit weitergegeben werden.

Abb. 5.3 zeigt, wie die Parameterwerte mitsamt der UUID als ein JSON-Objekt nach der Konfiguration des Prozessors an das Eingabe-Topic publiziert werden. Die Parameterwerte werden mithilfe des Wrappers (s. Abschnitt 4.4.2) an die Applikation weitergeleitet. Jeder Prozessor hat sein eigenes Eingabe-Topic, das in Redis mit einer eigenen UUID von PROOF benannt und angelegt wird. Das Eingabe-Topic wird als ein Nachrichtenkanal zur Parameterübertragung zwischen seinem Prozessor und seinem PROOF-Prozess angesehen. Darauf aufbauend wird die Parameterübergabe aller Prozesse als separate Kommunikationsprozesse durchgeführt, die sich nicht gegenseitig beeinflussen, was Übertragungsfehler an falschen Empfängern oder Datenverlust oder übermäßige Wartezeiten bei der Parameterübertragung vermeidet.

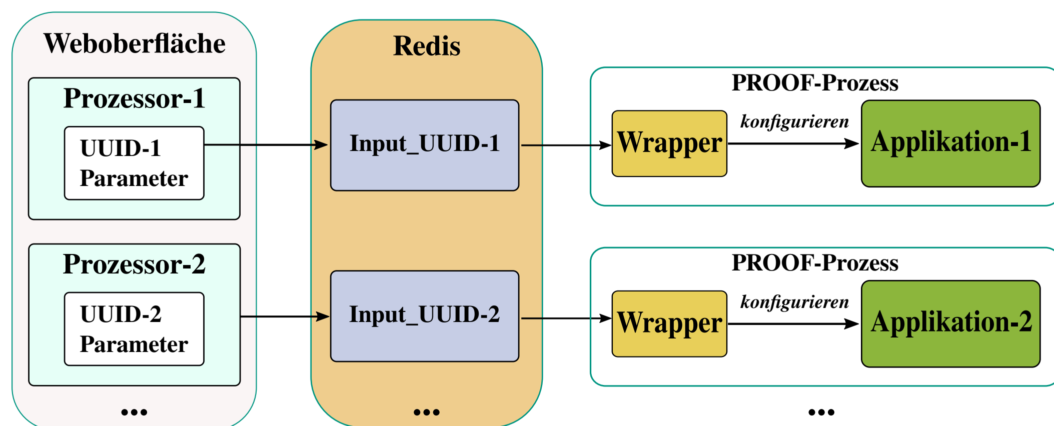

Abb. 5.3.: Parameterübertragung

5.2.4 Kopplung von Prozessoren

Nachdem alle Prozessoren eines Workflows Schritt für Schritt eingefügt und konfiguriert wurden, werden sie basierend auf dem Simulationsablauf zum Aufbau des Workflows miteinander gekoppelt. Wie in Abb. 5.4 illustriert, brauchen Nutzer nur mit der Maus eine grüne Linie von einem Prozessor zu einem anderen zu ziehen, um eine Verbindung zwischen ihnen herzustellen. Die Verbindung wird nach dem Aufbau mit *SUCCESS* benannt. Dafür ist keine extra Konfigurationsdatei anzulegen, sind keine Schnittstellen zu implementieren oder der Quellcode anzupassen. Dies wird von keinem der in Kapitel 3 analysierten Frameworks im Detail aufzeigt und bereitstellt. Stattdessen verlangen alle untersuchten Frameworks, dass Konfigurationsdateien oder Integrationsschnittstellen zum Aufbau der Verbindungen zwischen Applikationen von Nutzern selbst fest codiert werden müssen.

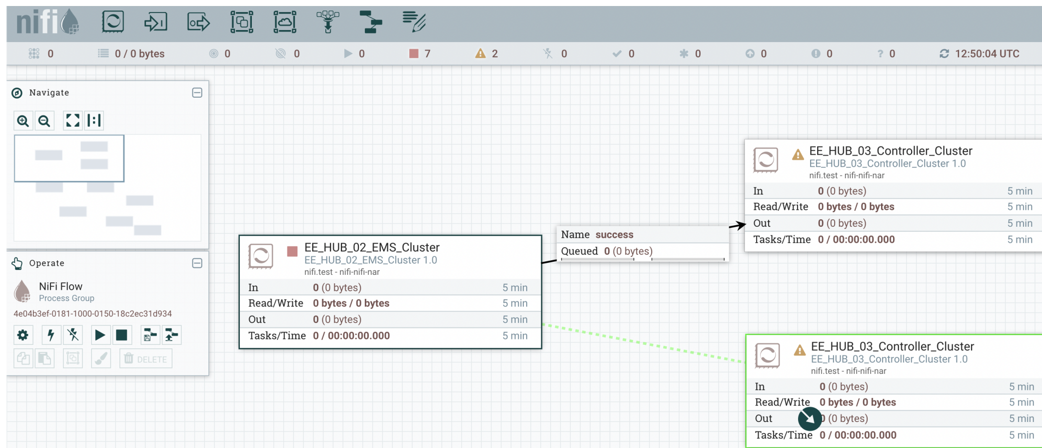

Abb. 5.4.: Kopplung von zwei Prozessoren

Über die Verbindung können Daten zwischen Prozessoren transportiert werden. Die Datenaustauschschnittstelle von PROOF beruht auf Redis und der NiFi-Engine. Beispielsweise sind die drei Prozessoren in Abb. 5.4 auf der Weboberfläche virtuell verknüpft. Tatsächlich bleiben die Framework-Prozesse 1, 2 und 3, die die drei Prozessoren repräsentieren, voneinander unabhängig im Backend. Wie in Abb. 5.5 zu sehen ist, laufen die drei Framework-Prozesse jeweils in einem Docker-Container als ein eigenständiger Microservice. Mithilfe von Redis und der NiFi-Engine tauschen sie miteinander Daten über Nachrichtentopic 1 basierend auf dem Publish/Subscribe-Messaging aus, d.h., dass Daten von Prozess 1 an Topic 1 gesendet und in einer Warteschlange gespeichert werden. Die Prozesse 2 und 3 empfangen die Daten nach der Regel FIFO. Die drei Prozesse besitzen keine unmittelbare Kommunikation miteinander und kennen sich nicht und sind daher entkoppelt. Dabei wird eine lose Kopplung etabliert, die die Flexibilität der Datenübertragung erhöhen kann. Im Gegensatz zur starken Kopplung kann diese lose Kopplung mehrere Datenübertragungsverfahren implementieren, z.B. eine Eins-zu-Viele- oder Viele-zu-Eins-Datenübertragung. Da bei einer starken Kopplung Sender und Empfänger besonders eng gekoppelt und schwer erweiterbar sind, werden dort Daten in der Regel dabei nur Eins-zu-Eins übertragen.

5.3 Volume – ein gemeinsamer Speicherbereich

Nachdem mit Abschnitt 4.3 eine Übersicht über das Volume zur Speicherung großer Datenmengen in Form von Dateien für die PROOF-Architektur gegeben wurde, widmet sich dieser Abschnitt dem Konzept des Volume, das sowohl zum Teilen großer

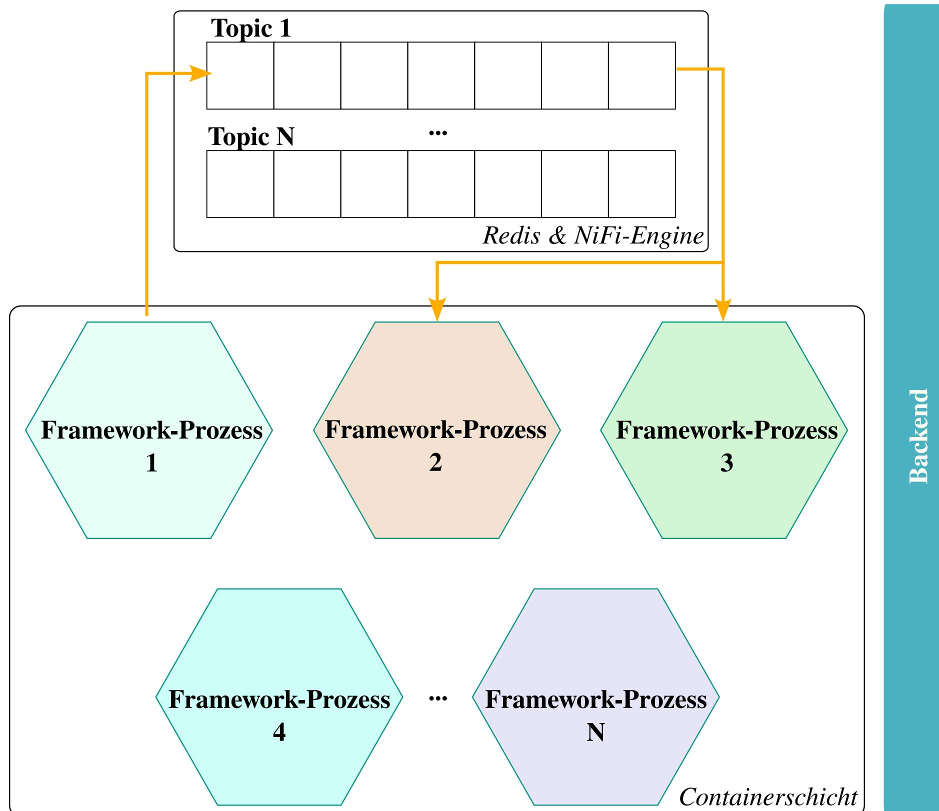

Abb. 5.5.: Datenübertragung zwischen Framework-Prozessen

Datenmengen als auch zum Speichern großer Anwendungsmodelle verwendet werden kann. Im folgenden Unterabschnitt 5.3.1 wird zunächst der Austausch großer Datenmengen unter Verwendung eines Volume anhand eines konzeptionellen Diagramms und eines Ablaufdiagramms erläutert. In Unterabschnitt 5.3.2 wird geklärt, wie große Anwendungsmodelle nicht in Docker-Images, sondern in einem Volume bereitgestellt werden, auf die jeder Framework-Prozess mithilfe eines Wrappers zugreifen und diese verwenden kann.

5.3.1 Austausch großer Datenmengen mit Volumes

In Abschnitt 5.1 wurde durch einen Vergleich entschieden, dass Redis als ein In-Memory-Datenspeicher für PROOF eingesetzt wird, um kleine Datenmengen hochperformant zu speichern und zu verarbeiten. Für einige spezielle Anwendungsfälle müssen allerdings auch große Datenmengen zwischen Applikationen ausgetauscht werden, die nicht einfach oder sehr langsam oder nicht sinnvoll über Redis oder andere Kommunikationsdienste (Apache Kafka, Apache Zookeeper etc.) übertragen

werden können. Für solche Fälle wird das Volume von einem verteilten Dateisystem auf dem Computing-Cluster für PROOF bereitgestellt.

Wie in Abb. 5.6 dargestellt, kann das Volume große Datenmengen als Dateien speichern. Alle Framework-Prozesse bzw. Docker-Container können das Volume gemeinsam teilen und verwenden, um das Speichern und Übertragen großer Daten-Dateien zu lösen. Im Gegensatz zu kleinen Datensätzen werden große Datenmengen nicht mehr in den Framework-Prozessen gehalten, sondern immer im Volume zur weiteren Verwendung gespeichert. Dabei können die Framework-Prozesse nicht viele Speicherressourcen beanspruchen und die Daten beim Schließen der Framework-Prozesse gelöscht werden. Mithilfe eines E/A-Adapters kann jeder Framework-Prozess im Volume Eingabedateien für seine Applikation lesen und Ausgabedateien seiner Applikation schreiben.

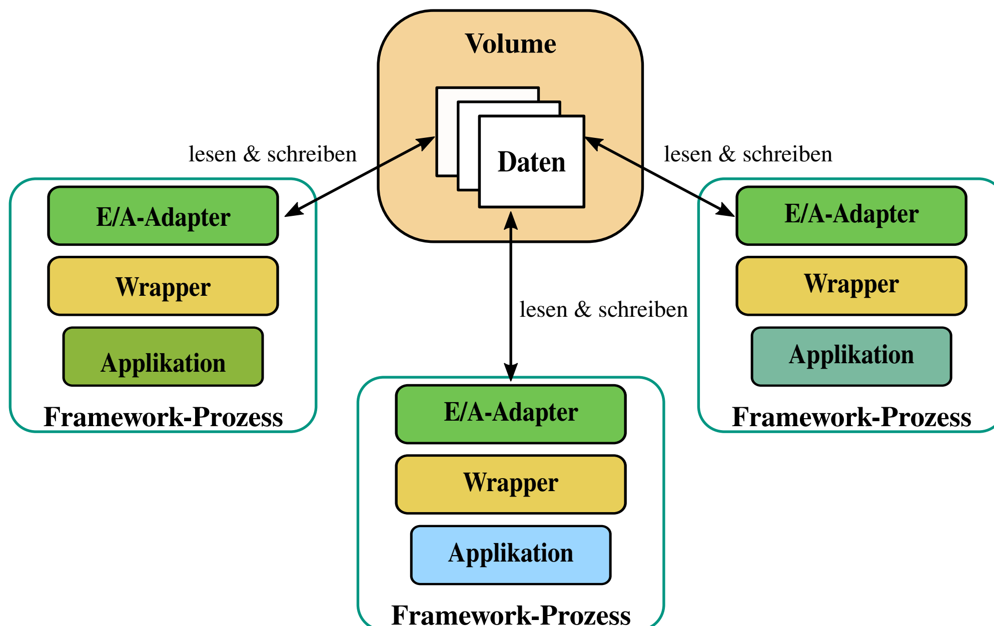

Abb. 5.6.: Austausch großer Datenmengen im Volume

Abb. 5.7 veranschaulicht das Ablaufdiagramm zur Verarbeitung von Daten mit den E/A-Adaptern innerhalb eines Framework-Prozesses. Wenn eine Nachricht in einem Prozess ankommt (siehe die linke Seite des Ablaufdiagramms), liest der Eingabeadapter sie und verwendet seinen Parameter *is_bigdata*, um zu bestimmen, ob es sich bei den zu verarbeitenden Daten um große Datenmengen handelt. Falls nein, liest der Eingabe-Adapter die kleinen Daten direkt aus der Nachricht und speichert sie im *Eingabe*-Parameter für seine Applikation; falls ja, ruft der Eingabe-Adapter den Dateinamen aus der Nachricht zum Herausfinden der Datei im Volume ab, die die großen Daten enthält. Die Datei wird nicht in den Framework-Prozess kopiert,

sondern direkt als Eingabe von der Applikation benutzt. Währenddessen stellt die rechte Seite des Ablaufdiagramms dar, wie die Ausgabedaten einer Applikation mit einem Ausgabe-Adapter verarbeitet werden. Wenn die Applikation durchgeführt ist, liest ihr Ausgabeadapter ihre ausgegebenen Daten und extrahiert die Datengröße. Wenn die Datengröße **kleiner** oder **gleich** 10 MB (empirisch ermittelt) ist, werden die Daten als kleine Daten in PROOF angesehen und direkt in einem JSON-Objekt eingepackt. Anschließend wird auch der Parameter *is_bigdata* als *false* belegt und dann zusammen mit dem JSON-Objekt an Redis gesendet. Bei einer Größe von **mehr als** 10 MB werden die Daten als große Daten betrachtet und direkt als eine Datei gespeichert. Die Datei wird nach dem Ausgabedatennamen benannt und an das Volume übergeben. Anschließend wird die Datei in dem Framework-Prozess gelöscht. Zudem wird auch dem Parameter *is_bigdata* der Wert *true* zugewiesen und zusammen mit dem Dateinamen in einem JSON-Objekt gespeichert, das schließlich an Redis gesendet wird.

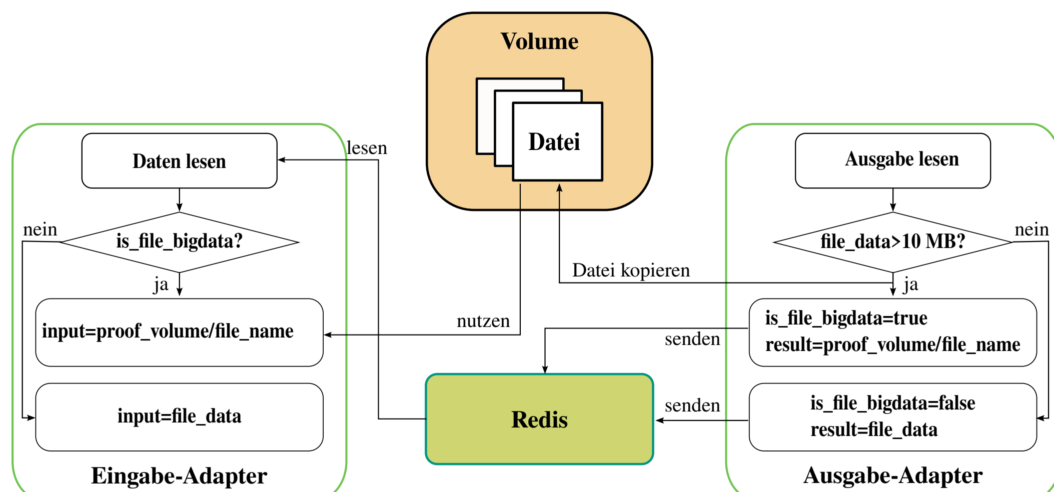

Abb. 5.7.: Ablaufdiagramm zur Datenverarbeitung innerhalb eines Framework-Prozesses

Durch die Verwendung von Volumes wird einerseits die Verarbeitungseffizienz für große Datenmengen erheblich verbessert, da große Datenmenge in Form von Dateien direkt in Volumes gelesen und geschrieben werden können. Darüber hinaus werden die Last und der Ressourcenverbrauch der Framework-Prozesse, die große Datenmengen verarbeiten müssen, stark reduziert, da diese Framework-Prozesse ihre großen Datenmengen nicht mehr über die ganze Laufzeit in ihren Containern halten. Dies vermeidet die Nachteile der Übertragung großer Datenmengen über Redis, wie z.B. übermäßige Wartezeiten oder sogar Datenverlust bei einem Systemabsturz.

5.3.2 Bereitstellung von Applikationen

Häufig wird immer dieselbe Software oder Programmiersprache für die Modellierung und Analyse in bestimmten wissenschaftlichen Systemlösungen verwendet. Beispielsweise werden in vielen unterschiedlichen Systemsimulationen Matlab-Modelle zur Lösung mathematischer Probleme, Python-Funktionen zur Datenverarbeitung, Modelica-Modelle als FMU zur Modellierung physikalischer Systeme und C++ zur Implementierung verschiedener Optimierungsalgorithmen verwendet.

Wenn ein spezielles Docker-Image für jede neue Applikation erstellt werden muss, um die Applikation in PROOF zu integrieren, führt dies zu einem stetigen und komplexen Integrationsaufwand. Dabei führt es dazu, dass viele Docker-Images verwaltet werden müssen, die zahlreiche Systemressourcen in Anspruch nehmen. Um mit dem Problem umzugehen, bietet PROOF unterschiedliche generische Framework-Prozesse an, z.B. den Matlab-Prozess, den Python-Prozess usw. Jeder generische Framework-Prozess ist verwendbar für alle Applikationen einer Software oder Programmiersprache, z.B. der generische Matlab-Prozess ist geeignet zur Integration jeder mit Matlab eigenständig ausführbaren Applikation in PROOF. Das bedeutet, dass nicht für jedes Matlab-Modell ein spezielles Docker-Image aufgebaut werden muss. Durch die ständige Verwendung desselben Matlab-Prozesses können alle erforderlichen Matlab-Modelle in PROOF integriert und ausgeführt werden. Dadurch kann der Integrationsaufwand stark reduziert werden.

Außerdem ist die Größe vieler Applikationen, z.B. Matlab-Applikationen oder FMU-Modelle, mehr als 1 GB. Sie eignen sich daher nicht zum Einkapseln in einem Docker-Image bzw. Docker-Container, da jedes Docker-Image bzw. jeder Docker-Container begrenzte Systemressourcen hat. Daher werden solche Applikationen nicht mehr in Docker-Containern bzw. Framework-Prozessen für die ganze Laufzeit ausgeführt, sondern in einem Volume bereitgestellt. Wie in Abb. 5.8 konzipiert, teilen sich alle generischen Framework-Prozesse das Volume. Unter Verwendung eines Wrappers kann jeder generische Framework-Prozess auf im Volume gespeicherte Applikationen zugreifen. Wenn ein Framework-Prozess auch mit großen Datenmengen umgehen muss, werden die großen Daten, wie im vorherigen Abschnitt beschrieben, auch im Volume gehalten. Für diesen Fall besitzt der generische Framework-Prozess dabei nur einen Wrapper zur Ausführung der Applikation und einen Eingabe-/Ausgabeadapter zur Datenverarbeitung.

Damit ein generischer Framework-Prozess seine Applikation im Volume herausfinden kann, bietet PROOF einen Parameter *Application Name* für jeden generischen Framework-Prozess auf der *PROPERTIES*-Registerkarte im Konfigurationsdialog an

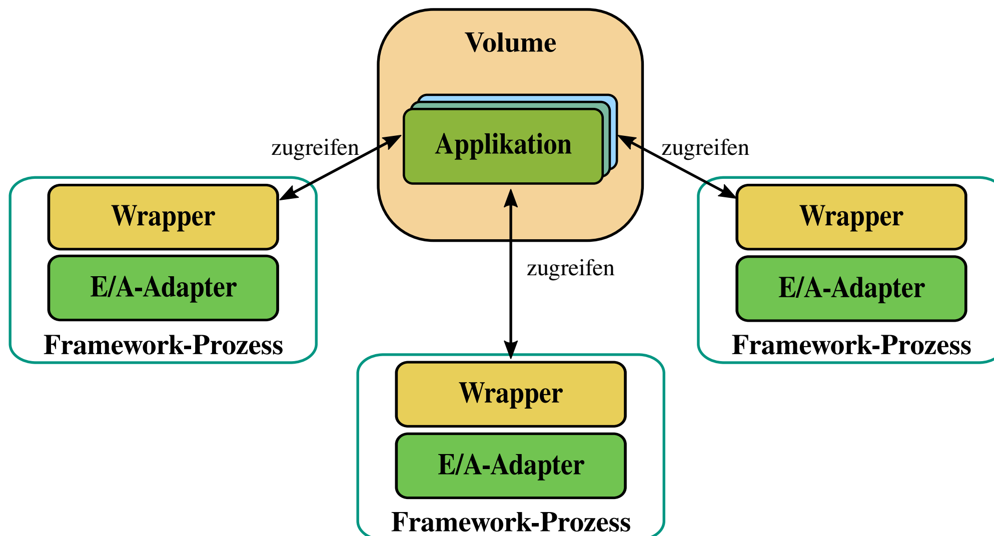

Abb. 5.8.: Bereitstellung von Applikationen im Volume

(siehe Abb. 5.9), der an den entsprechenden Framework-Prozess übergeben wird, damit die benötigte Applikation im Volume gefunden werden kann (s. Abschnitt 5.2.3).

Configure Processor

SETTINGS | SCHEDULING | PROPERTIES | COMMENTS

Required field

Property	Value
Input Directory	/input
Input File(s)	*.json
Output Directory	/output
Output File(s)	Bus* Line*
Application Name	Forecasting.fmu

CANCEL APPLY

Abb. 5.9.: Konfigurationsdialogfenster für Eingabe des Applikationsnamens

Durch die Nutzung generischer Prozesse und von Volumes wird einerseits die Implementierungseffizienz zur Integration von Applikationen in PROOF stark verbessert. Andererseits können die Systemanforderungen für Framework-Prozesse mit großen Applikationen dadurch reduziert werden. Dies verbessert die Systemstabilität und Arbeitseffizienz von PROOF.

5.4 Konfigurierbare Services

Neben den im vorherigen Abschnitt dargestellten generischen Framework-Prozessen (z.B. Matlab-, FMU-, C++-, Python-, Java-Prozesse usw.) werden verschiedene konfigurierbare Services von PROOF für unterschiedliche Anwendungsfälle bereitgestellt, um die in Abschnitt 3.1 beschriebene Anforderung 9 (Hilfsbausteine für generische Funktionalitäten) abzudecken. Zuerst wird der *MqttService* zur Erstellung der Schnittstelle mit externen Applikationen/Modellen/Anlagen in diesem Abschnitt beschrieben. Anschließend wird der *TimeseriesService* zum Zugriff auf die entsprechenden Daten vorgestellt. Danach finden sich verschiedene Dateioperationsservices zur Verarbeitung der xlsx-, CSV-, PKL- und PDF-Dateien in diesem Abschnitt. Schließlich werden weitere Services zur Implementierung verschiedener Anwendungsfälle in Anhang A.8.3 dargestellt.

5.4.1 MQTT-Service

Bei einigen Systemsimulationen sind verwendete Applikationen, Modelle oder Services zu groß oder zu komplex, um sie in PROOF zu integrieren. Um Interaktion mit solchen Objekten zu erstellen, stellt PROOF nicht nur den in NiFi eingebauten HTTP-Server, sondern auch eine MQTT-Dienstschnittstelle bereit, da viele physikalische Modelle/Anlagen eine MQTT-Schnittstelle (s. Auflistung 5.2) zum Senden der Nachrichten über ein Netzwerk mit geringer Bandbreite besitzen [MQT22; MK20].

```
1   Klasse MqttService
2       FUNKTION __init__()
3           Initialisiere die Parameter eines MqttServers
4       End FUNKTION
5       FUNKTION configure_client()
6           Konfiguriere ein Mqtt-Objekt MqttServers
7       End FUNKTION
8       FUNKTION connect()
9           Verbinde einen Mqtt-Server
10      End FUNKTION
11      FUNKTION on_connect()
12          Verbinde ein Mqtt-Topic zum Empfang von Nachrichten
13      End FUNKTION
14      FUNKTION create_mqtt_client()
15          Erstelle ein Mqtt-Objekt
16      End FUNKTION
17      FUNKTION publish()
18          Sende Nachrichten an ein Topic
19      End FUNKTION
20      FUNKTION disconnect()
21          Trenne eine Verbindung mit dem Mqtt-Server
22      End FUNKTION
```

Auflistung 5.2: MQTT-Service

Auflistung 5.2 stellt die Implementierung des *MqttServices* von PROOF dar. Damit kann ein *Mqtt-Subscriber* zum Empfangen der externen Nachrichten (siehe Zeile 11 bis 16) und ein *Mqtt-Publisher* zum Senden der Nachrichten nach außen (siehe Zeile 14 bis 19) aufgebaut werden. Dabei sind fünf grundlegende Parameter bei der Initialisierung zu konfigurieren, nämlich *broker*, *port*, *username*, *pw* und *topic* (siehe Zeile 3). Wie der im vorherigen Abschnitt 5.3.2 vorgestellte Parameter *Application Name* können die fünf Parameter auf der Registerkarte *PROPERTIES* im Konfigurationsdialog eingegeben werden, um eine Instanz vom *MqttService* zu erstellen. Die vollständige Implementierung des *MqttServices* wird in Anhang A.8 dargestellt.

5.4.2 Timeseries-Service

Bei Zeitreihendaten handelt es sich um eine nach Zeit strukturierte Gruppe von Datenpunkten, die über Zeitintervalle gesammelt werden, um Datenänderungen im Laufe der Zeit über Millisekunden, Tage oder sogar Jahre zu verfolgen. Die Daten werden in der Reihenfolge organisiert, in der sie auftreten und zur Verarbeitung gelangen. In Energiesystemen werden Zeitreihendaten häufig verwendet, um z.B. Wetter-, Elektroenergie oder Grid-Zustandsinformationen zu speichern. Außerdem eignet sich die Zeitreihenanalyse für die Prädiktivmodellierung und die Prognose von Ergebnissen in Energiemodellen. Historische Änderungsdatensätze können auf viele Energieprognosemodelle angewendet werden. Zum Lesen und Schreiben der Zeitreihendaten bietet PROOF den *TimeseriesService* ([Bra+17]), der in Auflistung 5.3 dargestellt wird. Zum Aufbau eines *TimeseriesServices* sind die Parameter *url* und *headers* (siehe Zeile 5-7) festzulegen, die im Konfigurationsdialog des *TimeseriesService*-Prozessors eingegeben werden können. Der Wrapper des *TimeseriesServices* kann nach Bedarf die Methode *read_data* zum Lesen von Daten aus der Zeitreihendatenbank oder die Methode *write_data* zum Schreiben von Daten in der Zeitreihendatenbank verwenden. Dieser Service wird für die Anwendungsfälle 7.2.2 [Gon+21] und 7.2.3 [Pop+21] in Kapitel 7 verwendet.

```
1   import json
2   import requests
3
4   class TimeseriesService:
5       def __init__(self, url, headers={"Accept":"text/csv"}):
6           self.url = url
7           self.headers=headers
8
9       def read_data(self):
10          response = requests.get(self.url, headers=self.headers)
11          return response.content
12
13      def write_data(self, data_to_write):
14          request.post(self.url, data=data_to_write, headers=self.headers)
```

Auflistung 5.3: Zeitreihendaten-Service

5.4.3 Dateioperation

Dieser Unterabschnitt konzentriert sich auf die Dateioperation zur Verarbeitung verschiedener Dateien.

Excel

Mit Microsoft Excel können Daten in einer Tabelle formatiert, organisiert und berechnet werden. Als Tabellenkalkulationsprogramm ermöglicht Excel umfangreiche Berechnungen mit Formeln und Funktionen, unter anderem mit kaufmännischen, statistischen und Datumsfunktionen. Außerdem kann Excel auch als relationale Datenbank verwendet werden. Beispielsweise werden im Bereich Energiesysteme Excel-Dateien zur Speicherung und Berechnung von Energiedaten genutzt. Dafür wird ein Service im Hinblick auf *Exceloperation* von PROOF wie in Auflistung 5.4 bereitgestellt. Dieser Service kann eine Excel-Datei basierend auf dem in der Benutzeroberfläche konfigurierten Dateinamen zum Lesen (s. Zeile 10-12) und Schreiben ((s. Zeile 6-8)) öffnen. Die vollständige Implementierung des Services wird in Anhang A.9 dargestellt. Durch Verwendung dieses Services wird die manuelle Verarbeitung von Excel-Tabellendaten automatisiert. Dies verbessert die Effizienz der Excel-Datenverarbeitung erheblich, insbesondere wenn die Datenmenge sehr groß ist.

```
1   Klasse ExcelOperation
2       FUNKTION __init__()
3           Initialisiere einen Dateiname−Parameter
4       End FUNKTION
5
6       FUNKTION write_data()
7           Schreib Daten in eine Exceldatei
8       End FUNKTION
9
10      FUNKTION collect_data()
11          Lies und Sammel Daten von einer Exceldatei
12      End FUNKTION
```

Auflistung 5.4: Excel-Operation

CSV

Das Dateiformat CSV beschreibt den Aufbau einer Textdatei zur Speicherung oder zum Austausch strukturierter Daten. Es wird oft in vielen Forschungsprojekten benutzt, um Daten zwischen unterschiedlichen Computerprogrammen auszutauschen. Beispielsweise werden CSV-Dateien neben Excel-Dateien in Energiesystemen häufig verwendet, um Energiedaten aufzuzeichnen, die von physikalischen Modellen/Anlagen gesammelt oder von Softwaremodellen simuliert werden. Um CSV-Dateien in PROOF zu lesen und zu schreiben, wird ein *CSVOperation*-Service implementiert,

der in Auflistung 5.5 dargestellt wird. Nach dem Lesen des auf der Weboberfläche vorgegebenen Dateinamens kann der Service mit der Methode *read()* den Dateiinhalt im JSON-Format (Dict-Format in Python) lesen und zurückgeben (siehe Zeile 6 bis 8). Außerdem kann der Service den Inhalt eines übergebenen JSON-Parameters in eine CVS-Datei schreiben (siehe Zeile 10 bis 12). Die vollständige Implementierung des Services wird in Anhang A.10 dargestellt. Dieser Service wird für den Anwendungsfall 7.2.1 [Liu+18; Liu+19] in Kapitel 7 verwendet.

```
1  Klasse CSVOperation
2      FUNKTION __init__()
3          Initialisiere einen Dateiname−Parameter
4      End FUNKTION
5
6      FUNKTION read()
7          Lies Daten einer CSV−Datei
8      End FUNKTION
9
10     FUNKTION write()
11         Schreib Daten in eine CSV−Datei
12     End FUNKTION
```

Auflistung 5.5: CSV-Operation

PKL

Eine PKL-Datei wird als ein Python-Modul benutzt, um Objekte in Dateien auf der Festplatte zu serialisieren und zur Laufzeit zu deserialisieren. Das Modul kann mit Python *pickle* und der *dump*()-Methode erstellt und mit Python *pickle* und der *load()*-Methode geladen werden. Zum Lesen der Daten aus den PKL-Dateien für weitere Simulationsschritte und zum Schreiben der Daten in die PKL-Dateien als Ergebnisse wird ein *PKLReader*-Service wie in Auflistung 5.6 in PROOF implementiert. Die vollständige Implementierung des Services wird in Anhang A.11 dargestellt. Dieser Service wird für den Anwendungsfall 7.2.3 [Pop+21] in Kapitel 7 verwendet, um die Stammdaten und Ressourcenplanung von den physikalischen Modellen (BHKW, Elektrolyse, Gasspeicher, Batterie, Methanisierung) zu speichern.

```
1  Klasse PKLReader
2      FUNKTION __init__()
3          Initialisiere einen Dateiname−Parameter
4      End FUNKTION
5
6      FUNKTION load_obj()
7          Lies Daten einer Pkl−Datei
8      End FUNKTION
9
10     FUNKTION write()
11         Schreib Daten in eine Pkl−Datei
12     End FUNKTION
```

Auflistung 5.6: PKL-Operation

PDF

Das Portable Document Format (PDF) ist ein plattformunabhängiges Dateiformat. Es wird verwendet, um Informationen wie wissenschaftliche Artikel, Bücher usw. in einem Dokument zu speichern. In wissenschaftlichen Studien werden Daten häufig manuell aus PDF-Dateien ausgelesen. Daher wird ein *PdfOperation*-Service von PROOF bereitgestellt (siehe Auflistung 5.7), um den Extrahierungsvorgang zu automatisieren. Mit dem vorgegebenen Dateinamen kann der Service die PDF-Datei öffnen und dann die Textinformationen der angeforderten Seite lesen und zurückgeben (siehe Zeile 6 bis 8). Des Weiteren kann der Service Daten in einer bestimmten Zeile, einer bestimmten Spalte aus einer bestimmten Tabelle auf einer bestimmten Seite lesen (siehe Zeile 10 bis 12). Die Daten können auch nach Faktor umgerechnet werden. Die vollständige Implementierung des Services wird in Anhang A.12 dargestellt. Unter Verwendung des Services kann der manuelle Lesevorgang der Daten aus PDF-Dateien automatisiert werden, damit die Effizienz beim Lesen verbessert wird. Dieser Service wird bei der Zusammenarbeit mit dem Institut für Technische Chemie verwendet.

```
1  Klasse PdfOperation
2      FUNKTION __init__()
3          Initialisiere einen Dateiname-Parameter
4      End FUNKTION
5
6      FUNKTION read_text()
7          Lies Daten einer Seite einer Pdf-Datei
8      End FUNKTION
9
10     FUNKTION read_data_from_table()
11         Lies Daten einer Tabelle einer Pdf-Datei
12     End FUNKTION
```

Auflistung 5.7: PDF-Operation

5.4.4 Weitere Services

In PROOF wurden weitere Services für verschiedene Anwendungsfälle entwickelt. Details von den Services sind in Anhang A.8.3 zu finden.

5.5 Zusammenfassung

In diesem Kapitel wurde der Beitrag B2 vorgestellt. Dabei wurden zunächst verschiedene Kommunikationsschnittstellen basierend auf den Anforderungen bezüglich der

Datenübertragung verglichen. Ergebnis des Vergleichs ist, dass Redis als Kommunikationsinfrastruktur für PROOF besser geeignet ist.

Im Anschluss schilderte das Kapitel das Konzept im Hinblick auf den Datenfluss eines Workflows aus zwei NiFi-Prozessoren schrittweise. Dabei wurde geklärt, wie Parameter/Attribute in einer Prozessorklasse definiert, wie sie unter dem Tab *PROPERTIES* im Konfigurationsdialog auf der Weboberfläche konfiguriert und danach an den entsprechenden Framework-Prozess übertragen werden. Außerdem wurde der Aufbau zur Ankopplung von Prozessoren auf der Weboberfläche erläutert.

Danach widmete sich das Kapitel dem Konzept des Volumes, das sowohl zum Teilen großer Datenmengen als auch zur Bereitstellung von Anwendungsmodellen/-Applikationen verwendet wird, um die Verarbeitungseffizienz großer Daten und die Integrationseffizienz von Applikationen mithilfe von generischen Framework-Prozessen in PROOF zu verbessern.

Zum Schluss wurden verschiedene konfigurierbare Services von PROOF für unterschiedliche Anwendungsfälle vorgestellt. Dabei wurde der MQTT-Service als Schnittstelle mit externen Applikationen/Modellen/Anlagen präsentiert. Danach wurden der Zeitreihendaten-Service zum Zugriff der entsprechenden Daten beschrieben. Zusätzlich wurden verschiedene Dateioperation-Services zur Verarbeitung der xlsx-, CSV-, PKL- und PDF-Datei in diesem Unterabschnitt dargestellt.

Zusammenfassend werden die nachstehenden Anforderungen 3.1.2 durch den zweiten Beitrag erfüllt:

A1 **Hohe Konfigurierbarkeit**
In der Weboberfläche können Applikationen flexibel parametriert werden, ohne eine zusätzliche Schnittstelle zu implementieren oder zusätzliche Dateien anzulegen oder den Quellcode anzupassen.

A2 **Einfache Integration**
Unter Verwendung der generischen Framework-Prozesse, z.B. Matlab-, FMU-, Python-, Java-Prozesse usw., kann der Integrationsaufwand stark reduziert werden. Alle Applikationen/Modelle einer Software oder Programmiersprache können sich ihren generischen Framework-Prozess teilen und damit in PROOF integriert werden, ohne ein spezifisches Docker-Image für jede Applikation zu erstellen.

A3 **Performanter Datenaustausch**
Kleine Daten werden über Redis innerhalb von PROOF performant übertragen. Große Datenmengen werden im Volume gespeichert, um von allen Framework-Prozessen direkt verarbeitet zu werden.

A9 Generische Funktionalitäten

Die generischen Framework-Prozesse werden als Hilfsbausteine für generische Funktionalität zur Integration von Applikationen bereitgestellt. Die konfigurierbaren Services bieten verschiedene Funktionalitäten zur Interaktion mit externen Applikationen/Modellen/Anlagen, zum Lesen und Schreiben der Zeitreihendaten und zum Bearbeiten unterschiedlicher Dateien.

6 Parallelisierung und Koordination von Workflows

Nachdem mit den vorangehenden Kapiteln 4 und 5 die PROOF-Architektur und die konfigurierbare Kommunikation zwischen PROOF-Prozessen beschrieben wurden, diskutiert dieses Kapitel beruhend auf den Anforderungen 3.1.2 (A8, A9), wie Workflowapplikationen als PROOF-Bausteine in unabhängig voneinander lauffähigen Containern auf dem Rechnercluster koordiniert, skaliert und parallelisiert werden. Dies ermöglicht die Parallelisierung von Workflowapplikationen, die keine eigene Parallelverarbeitung anbieten, aber durch Partitionierung der Eingabe und parallele Ausführung mehrerer Instanzen parallelisiert werden können, um die Leistungsfähigkeit von Workflows zu verbessern. Ohne Werkzeuge wie PROOF ist die Bearbeitung der wissenschaftlichen Arbeitsschritte nicht nur kompliziert zu parallelisieren, sondern auch die Erhöhung der Skalierbarkeit bedeutet zusätzlichen Aufwand. Weiterhin wird in diesem Kapitel beschrieben, wie sich parallele Datenflüsse aus verschiedenen Framework-Bausteinen synchronisieren und in einen Datenfluss zusammenführen sowie sich Primary-Secondary-Co-Simulationen in PROOF realisieren lassen.

Das Kapitel ist wie folgt strukturiert: Zunächst beschreibt Abschnitt 6.1 das Konzept zum parallelen Ausführen mehrerer Instanzen eines PROOF-Bausteins. Hierzu wird ein Koordinationsservice für jeden PROOF-Baustein benötigt, um seinen PROOF-Baustein in einer entsprechenden Anzahl separater Container parallel zu starten. Außerdem kann der Service Daten aus einer Warteschlange eines Datenflusses empfangen und als Eingaben in verschiedene Instanzen verteilen sowie Ausgaben von allen Instanzen sammeln und weiter übertragen. Anschließend stellt Abschnitt 6.2 einen weiteren wichtigen Datenflussverarbeitungsservice vor, um mehrere eingehende Datenflüsse zu einem Ausgabedatenfluss zusammenzufügen. Hierfür wird eine neue deskriptive Sprache definiert, um Datenelemente aus Eingabedatenflüssen nach verschiedenen Strategien zu kombinieren und als Ausgabedaten weiterzugeben. Nachfolgend fokussiert Abschnitt 6.3 das Konzept der Primary-Secondary-Koordination. Hierzu wird ein PROOF-Prozess *Primary*-Prozess implementiert, der basierend auf einem vordefinierten Fahrplan andere Secondary-Prozesse steuern kann. Abschnitt 6.4 fasst schließlich die im Abschnitt 3.1.2 definierten, relevanten Anforderungen zusammen, die durch den dritten Beitrag gelöst werden.

6.1 Konzept der Parallelisierung und Koordination

Ein wissenschaftlicher Workflow, der in PROOF aufgebaut und gestartet wird, wird schrittweise (Modell für Modell), d.h. sequentiell ausgeführt. Dies bedeutet, dass zur Simulation einer Systemlösung ein Workflow die ganze Simulation in kleinere Applikation aufteilt. Diese diskreten Applikationen werden dann nacheinander in PROOF ausgeführt. Erst wenn eine Applikation beendet ist, beginnt die nächste, d.h. wenn ein Workflow gestartet wird, wird immer nur eine Applikation des Workflows im aktuellen Verarbeitungsschritt ausgeführt. Dieses Problem in der Informatik bedeutet eine enorme Verschwendung von Hardware-Ressourcen und Rechenzeit, da nur ein Teil der Hardware für bestimmte Applikation ausgeführt wird. Außerdem ist es ineffizient, komplexe Echtzeitsysteme unter Verwendung von serieller Datenverarbeitung zu implementieren.

Daher bietet sich das Parallel-Computing an. Während die serielle Verarbeitung jeweils eine einzelne Aufgabe ausführt, führt die parallele Verarbeitung mehrere Aufgaben gleichzeitig aus. Das Seriell-Computing „verschwendet" die potenzielle Rechenleistung, sodass Parallel-Computing die Hardware besser ausnutzt. Der Cluster, auf dem PROOF läuft, kann effektiv genutzt werden, wenn auf ihm viele Aufgaben parallel laufen können. Hierzu werden zwei Parallelismen in PROOF realisiert:

- **Task-Parallelismus**
  Die gleichzeitige Ausführung verschiedener Workflows/Tasks in PROOF wird in Abschnitt 4.3 beschrieben. Unter Verwendung der Container-Orchestrierungstechnologie können verschiedene Framework-Prozesse vom Prozessmanagement in Containern als Microservices auf dem Computing-Cluster verteilt und parallel ausgeführt werden.
- **Daten-Parallelismus**
  Die gleichzeitige Anwendung einer Operation/Task mit Verzweigung und Zusammenführung von Daten wird in diesem Abschnitt vorgestellt.

6.1.1 Übersicht

In Abb. 6.1 wird das Konzept zum parallelen Ausführen mehrerer Instanzen eines Workflowprozesses in PROOF illustriert. Hierzu stellt das Framework einen Koordinationsservice bereit, um nicht nur eine parallele Datenverarbeitung und Ausführung eines Framework-Prozesses zu ermöglichen, sondern auch Datenströme zu verzweigen und zusammenzuführen. In der Konfigurationsdatei *config* wird definiert, wie

viele Container zum parallelen Ausführen eines Framework-Prozesses vom Koordinationsservice gestartet werden. Im vorliegenden Beispiel werden drei Container c-x, c-y und c-z mit unterschiedlichen UUIDs gestartet. Jeder Container hat seinen eigenen Kommunikationskanal (Topic t-x, t-y und t-z) für den Datenaustausch mit dem Koordinationsservice. Sobald die Eingabe-Datenflusselemente (grüne Symbole) ankommen, verteilt der Koordinationsservice sie über die Kommunikationskanäle auf die drei Container. Nachdem ein Container seine Datenverarbeitung beendet hat, erhält der Koordinationsservice ein Ausgabe-Datenflusselement als Rückgabe (orange Symbole).

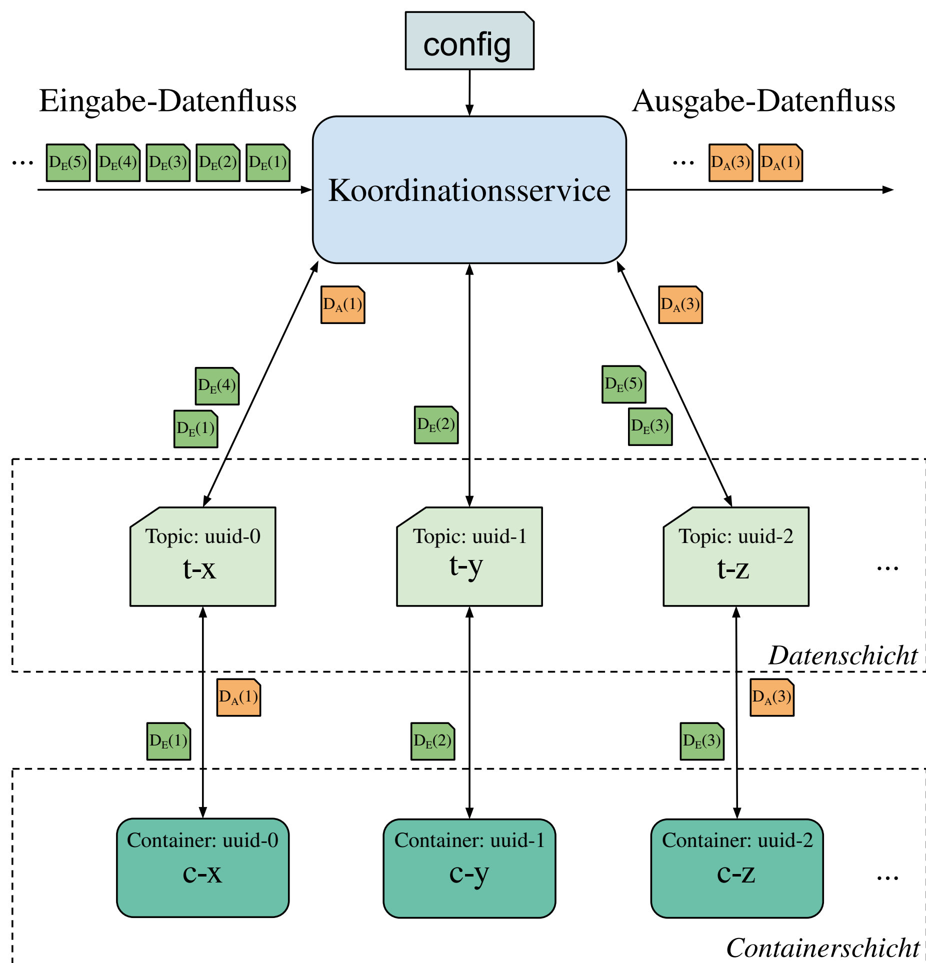

Abb. 6.1.: Konzept der Parallelisierung und Koordination

Um dieses Konzept mathematisch zu beschreiben, kann ein Datenfluss als eine geordnete Folge von Datenelementen

$$D_X(i) = V_i, i \in Z^+, V \in M, \tag{6.1}$$

mit

Z^+ = Menge geordneter Indizes, immer positiv,

M = Menge möglicher (strukturierter) Datenwerte, und

X = ID eines Datenflusses definiert werden.

Eine übliche Form der parallelen Datenflussverarbeitung wäre nun, die Elemente eines Datenflusses als Eingaben auf eine Reihe paralleler Instanzen eines Workflowprozesses zu verteilen, sodass in jeder Instanz des Prozesses gleichzeitig dieselbe Berechnung/Simulation für verschiedene Elemente des Datenflusses ausgeführt werden. Selbst in diesem Szenario sind viele Lösungen für die Verteilung der Datenelemente auf die verschiedenen Prozessinstanzen möglich, weshalb eine einfache „Worker-Pool"-ähnliche Verteilung verwendet wird, bei der für jedes Datenelement geplant ist, dass es von der nächsten freien Prozessinstanz verarbeitet wird.

6.1.2 Implementierung

Um eine solche Datenflussverarbeitungsstrategie im vorgestellten Framework zu implementieren, muss zunächst die Anzahl der gleichzeitigen Aufgaben und die Art der Verteilung (z.B. „Worker-Pool") in der Konfigurationsoberfläche eines NiFi-Prozessors durch den Workflow-Editor definiert werden. Wie in Abschnitt 4.5.2 schon vorgestellt, bietet das Dialogfeld *Configure Proccessor* der Weboberfläche dafür eine Konfigurationsoption mit dem Namen „*Concurrent Tasks*" (s. Abb. 6.2). Hiermit wird gesteuert, wie viele Container eines Prozessors parallel verwendet werden, d.h. dies steuert, wie viele Datenflusselemente gleichzeitig von diesem Prozessor bzw. der entsprechenden Applikation in Containern verarbeitet werden sollen.

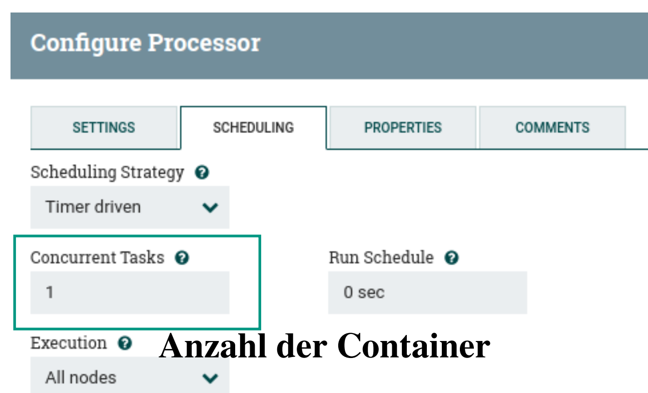

Abb. 6.2.: Parameter „Concurrent Tasks" zur Definition der Anzahl der Container

Nach der Konfiguration startet ein ausführbarer Koordinationsservice, der sich in jedem NiFi-Prozessor als Teil der abstrakten Prozessorklasse befindet, die vorgegebene Anzahl parallel arbeitender Instanzen in einer entsprechenden Anzahl separater Container (s. Abb. 6.1). Zusätzlich wird die „Worker-pool"-basierte Verteilung der Datenflusselemente in der Datenschicht instrumentiert. Daher wird, wie in Abb. 6.1 dargestellt, ein Eingabe-Datenfluss $D_{\mathrm{E}}(i)$ an den ausführbaren Koordinator übertragen. Die entsprechenden Datenelemente werden in einer Warteschlange zur Verteilung gespeichert. Durch die Verwendung der unterschiedlichen Kommunikationskanäle und der Verteilungsstrategie verteilt der ausführbare Koordinator Datenelemente an die Aufgabeninstanzen, z.B. $D_{\mathrm{E}}(1)$, $D_{\mathrm{E}}(2)$ und $D_{\mathrm{E}}(3)$ werden parallel von den Prozessinstanzen in den Containern c-x, c-y und c-z behandelt. Sobald eine Instanz ihre Datenverarbeitung beendet, wird das entsprechende Ergebnis an den Koordinator zurückgesendet. Anschließend erhält die Instanz die nächsten Eingabedaten aus der Warteschlange. Wie in Abb. 6.1 beispielsweise demonstriert, werden die nächsten Eingabedaten $D_{\mathrm{E}}(4)$ und $D_{\mathrm{E}}(5)$ an die beiden verfügbaren Instanzen in den Containern c-x und c-z übertragen, nachdem die Ergebnisdaten $D_{\mathrm{A}}(1)$ und $D_{\mathrm{A}}(3)$ beim Koordinator ankommen. Wenn es keine weiteren Verarbeitungsanforderungen für die Folge von Ausgabedaten gibt, leitet der Koordinator die empfangenen Ergebnisdaten an den nächsten NiFi-Prozessor weiter. Die Ergebnisse werden in einem zeitbasierten Ausgangsdatenfluss gesammelt, der basierend auf der Verfügbarkeit der Ergebnisse nach der Regel „FIFO" sortiert wird. Wenn die Ergebnisdaten auch in der gleichen Reihenfolge wie der vom Workflow-Editor definierte Eingabedatenfluss sortiert werden müssen („Reihenfolge beibehalten"), sortiert der Koordinator die Ergebnisdaten sequentiell basierend auf dem Index i der Eingabeelemente. Wenn der nächste NiFi-Prozessor, der den Ausgabedatenfluss erhält, für einen Datenfluss erfordert, dass alle parallel verarbeiteten Ergebnisdaten als ein großer Ergebnisdatensatz verfügbar sein müssen, wartet der Koordinator außerdem, bis alle Ergebnisdaten vollständig ankommen, und sortiert und verbindet sie dann als einen großen Ergebnisdatensatz entsprechend den Indizes i, und überträgt ihn schließlich.

Durch die Verwendung der Parallelverarbeitung des Koordinationsservices werden Workflows besonders effizient ausgeführt, ohne den Quellcode der Workflows anzupassen oder zusätzliche Schnittstellen zu implementieren. Dafür muss nur der Parameter „*Concurrent Tasks*" eines NiFi-Prozessors auf der Weboberfläche eingestellt werden. Durch Erhöhen dieses Werts kann der Prozessor eines Workflows in separaten Containern auf dem Computing-Cluster flexibel skaliert werden. Komplexe, große Datensätze und deren Verwaltung können mit dem Ansatz des Parallel-Computings organisiert werden. Die effektive Nutzung der Ressourcen und Hardware

(des Computing-Clusters) ist durch Parallelisierung sichergestellt, während bei der seriellen Berechnung nur ein Teil der Hardware verwendet wurde und der Rest im Leerlauf war. Eine weitere Diskussion in Bezug auf den Parallelisierung-Benchmark wird in Abschnitt 7.3.4 detailliert vorgestellt.

6.2 Merge-Service zur Synchronisierung mehrerer Datenflüsse

In vielen komplexen Anwendungsfällen müssen die Datenflüsse paralleler Berechnungen für die weitere Verwendung aggregiert und synchronisiert werden. Beispielsweise wird Energie aus unterschiedlichen Quellen (Windanlage, Solaranlage, Biogasanlage, Elektrolyse, Blockheizkraftwerk, Kernkraftwerk, etc.) in einem Multienergiesystem Energy-Hub aggregiert. Im Anschluss wird Energie nach verschiedenen Merkmalen kategorisiert und gespeichert, z.B. erneuerbare Energie oder saubere Energie. Hierzu wird ein weiterer Datenflussverarbeitungsservice *Merge-Service* als ein generischer Framework-Prozess in PROOF implementiert, um mehrere eingehende Datenflüsse zu einem Ausgabeergebnisdatenfluss zusammenführen zu können, indem er Datenflusselemente mit demselben Index i kombiniert.

6.2.1 Übersicht

Wie in Abb. 6.3 angezeigt, wird ein Merge-Service in diesem Fall drei Datenflüsse $D_{\mathrm{A}}(i)$, $D_{\mathrm{B}}(i)$ und $D_{\mathrm{C}}(i)$ als Eingaben parallel erhalten. Jedes Element von $D_{\mathrm{A}}(i)$ muss mit seinem entsprechenden Element in $D_{\mathrm{B}}(i)$ und $D_{\mathrm{C}}(i)$ kombiniert werden. Die Strategie, die definiert, wie die A-, B- und C-Datenflüsse zusammengeführt werden, wird als JSON-Objekt in einer Config-Datei beschrieben. Als Ausgabedatenelemente des Datenflusses $D_{\mathrm{O}}(i)$ wird der Merge-Service Kombinationen von Datenelementen der drei Eingabedatenflüsse erzeugen.

Das Ablaufdiagramm des Merge-Services wird in Abb. 6.4 präsentiert. Immer wenn neue Elemente in den zu den Datenflüssen gehörenden Warteschlangen ankommen, identifiziert der Merge-Service die Elemente gemäß ihren IDs. Anschließend liest der Merge-Service die vordefinierte Strategie (s. Abschnitt 6.2.2) in seiner Config-Datei. Danach überprüft der Service basierend auf der Strategie, ob alle entsprechenden Elemente bereits vorhanden sind. Dann werden sie zusammen an eine Aufgabeninstanz übertragen, die ein Ergebnis berechnet, das in die Ausgabewarteschlange vom

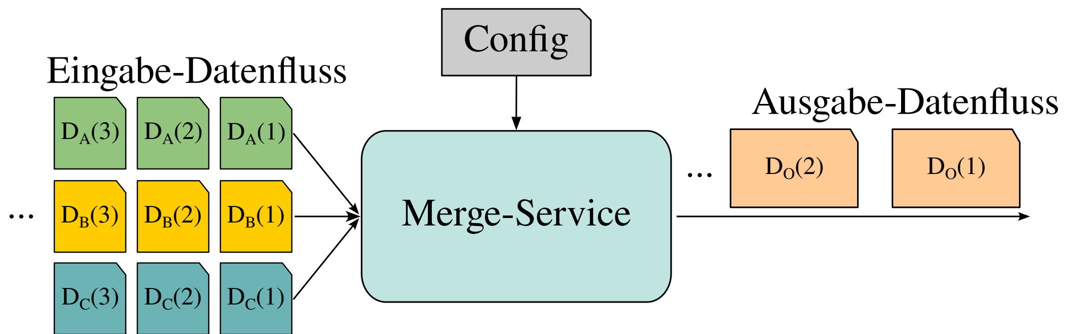

Abb. 6.3.: Merge-Service zur Synchronisierung mehrerer Datenflüsse

Ausgabedatenfluss gestellt wird. Wenn nicht alle entsprechenden Eingabeelemente verfügbar sind, werden die ankommenden Elemente gespeichert, bis ein vollständiges Tupel einschließend aller benötigten Elemente verfügbar ist. Zusammenfassend kann der Merge-Service unterschiedliche Strategien anwenden, um Eingabedaten an die Aufgabeninstanzen zu verteilen und die entsprechenden Ergebnisse zu sammeln und zu verwalten, ohne Datenverfälschung oder -verlust zu verursachen. Im Folgenden werden verschiedene Strategien zum Zusammenführen von Datenströmen dargestellt.

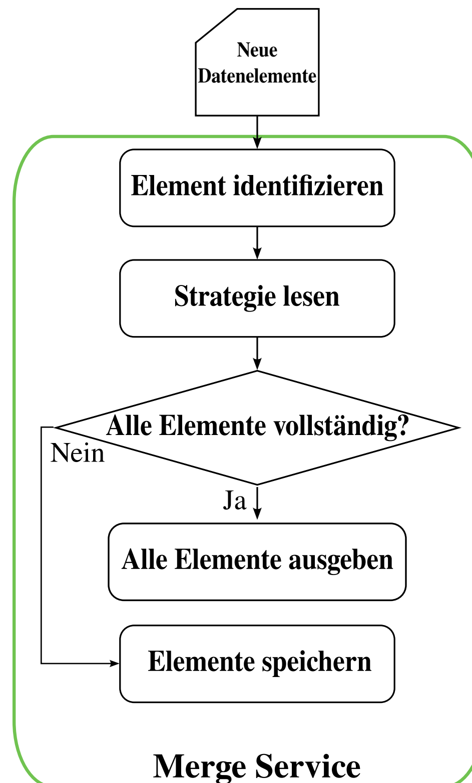

Abb. 6.4.: Ablaufdiagramm vom Merge-Service

6.2.2 Strategien

Zur Beschreibung verschiedener Strategien wird definiert, dass $N(D_X(i))$ die Anzahl der Elemente vom Eingabe-Datenfluss X repräsentiert, welche sich im Ausgabe-Datenfluss $D_O(i)$ befinden.

Strategie 1

Die Strategie 1 beschreibt den Fall FIFO, nämlich eine Warteschlange, aus der Elemente in der Reihenfolge abgerufen werden, in der sie zuvor abgelegt wurden. Außerdem ist die Anzahl der Elemente von den Eingabe-Datenflüssen pro Ausgabe-Datenfluss gleich 1, also $N(D_X(i)) = 1$. Wie in Abb. 6.5 dargestellt, umfasst jedes Ausgabeelement vom Datenfluss $D_O(i)$ ein Element jeweils vom Datenfluss $D_A(i)$, $D_B(i)$ und $D_C(i)$ mit dem identischen Index i.

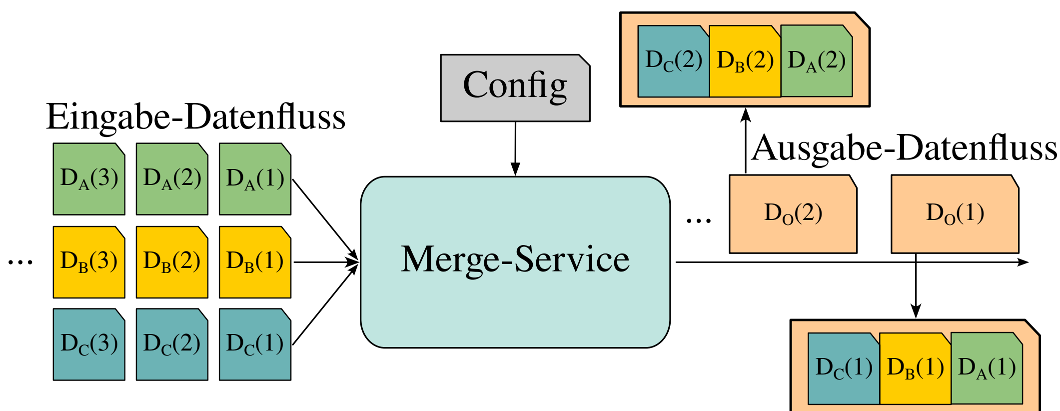

Abb. 6.5.: Strategie 1: FIFO & die Anzahl der Elemente von den Datenflüssen A, B und C ist gleich 1

Strategie 2

Die Strategie 2 bezeichnet ebenfalls eine FIFO-Situation. Anders als im ersten Fall ist die Anzahl der benötigten Elemente aller drei Datenströmen beliebig und nicht mehr gleich, nämlich $N(D_A(i)) \neq N(D_B(i)) \neq N(D_C(i))$. Wie in Abb. 6.6 angezeigt, hat jedes Datenelement vom Ausgabedatenfluss $D_O(i)$ zwei Elemente (Index 1 und 2) vom Datenfluss A, drei Elemente (Index 1, 2 und 3) vom Datenfluss B und ein Element (Index 1) vom Datenfluss C.

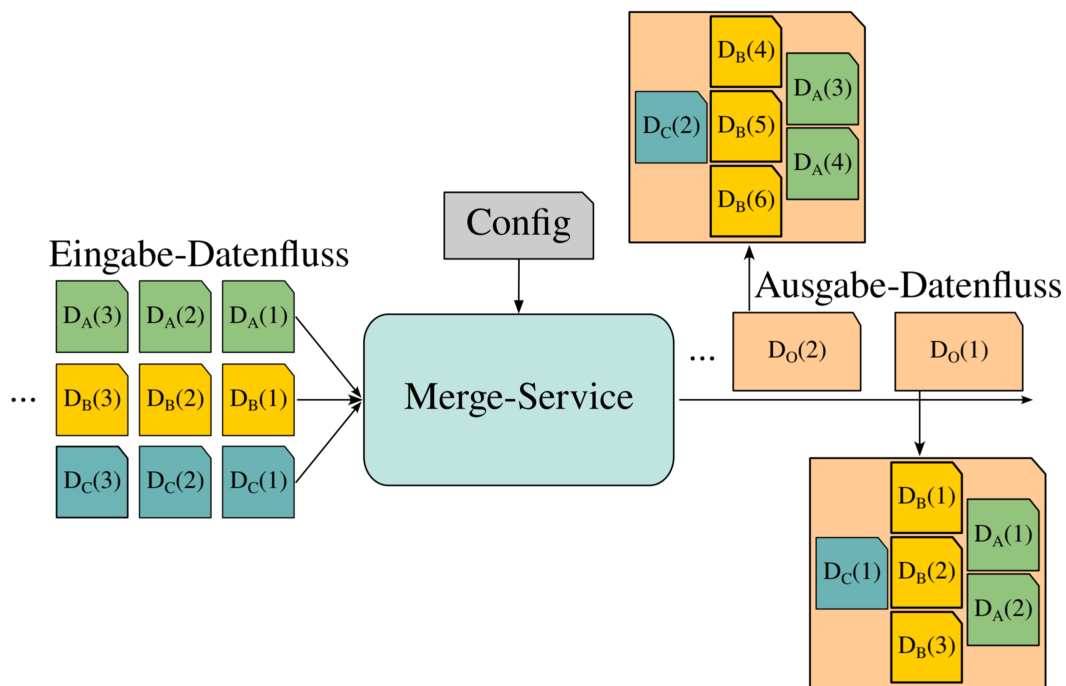

Abb. 6.6.: Strategie 2: FIFO & die Anzahl der Elemente von den Datenflüssen A, B und C ist ungleich

Strategie 3

Mit der dritten Strategie wird das FIFO-Prinzip nicht mehr eingehalten. Stattdessen werden Elemente von Datenströmen basierend auf einem bestimmten Attribut zusammengeführt. Außerdem ist die Anzahl der benötigten Elemente aller drei Datenströmen nicht identisch, nämlich $N(D_{\mathrm{A}}(i)) = 2, N(D_{\mathrm{B}}(i)) = 3, N(D_{\mathrm{C}}(i)) = 1$. Beispielsweise haben alle Elemente in Abb. 6.7 ein Attribut *Stadt*. Elemente mit demselben *Stadt*-Attribut werden zusammen gruppiert, z.B. das erste Ausgabeelement vom Datenfluss $D_{\mathrm{O}}(i)$ enthält die Elemente, deren *Stadt*-Attribut *Heidelberg* ist und die Elemente mit dem *Stadt*-Attribut *Karlsruhe* sind als eine Kombination danach auszugeben.

Strategie 4

Auf Basis von Strategie 3 werden mehr Attribute in der Strategie 4 berücksichtigt, um Elemente zusammenzuführen. Wie in Abb. 6.8 demonstriert, definiert die Config-Datei beispielsweise drei Attribute *Stadt*, *Nationalität* und *Fachrichtung* als Referenzbedingungen zur Beschreibung der Informationen von Studierenden in Deutschland. Unter diesen Referenzbedingungen werden die Elemente aus den Datenflüssen $D_{\mathrm{A}}(i)$, $D_{\mathrm{B}}(i)$ und $D_{\mathrm{C}}(i)$ klassifiziert und als eine Instanz ausgegeben,

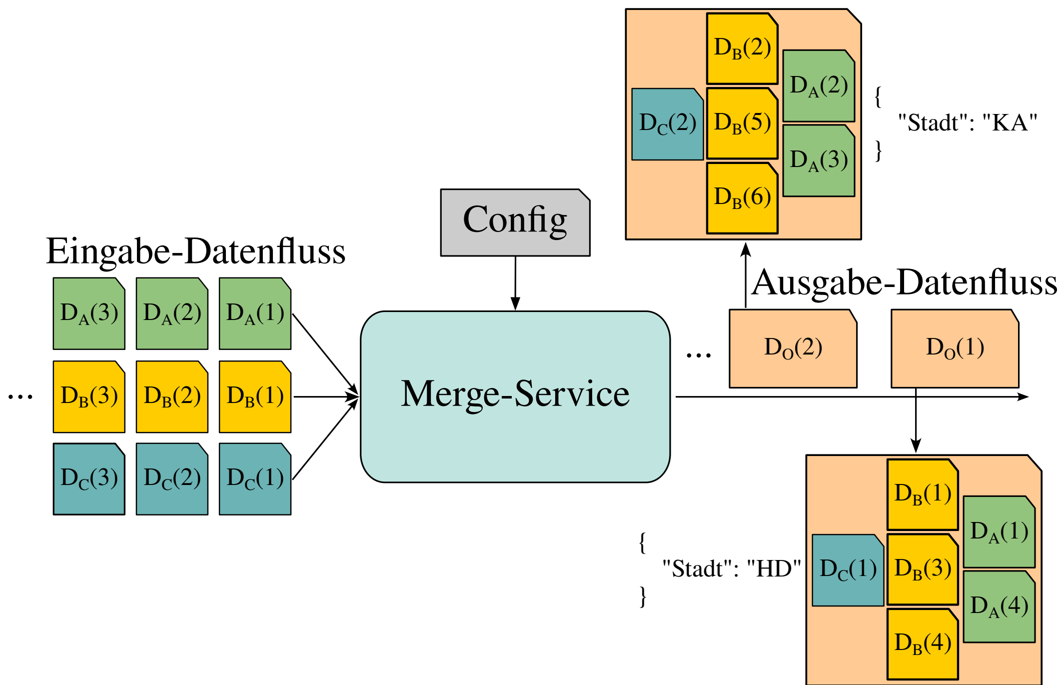

Abb. 6.7.: Strategie 3: Basierend auf einem Attribut, z.B. Stadt & die Anzahl der Elemente von den Datenflüssen A, B und C ist ungleich

z.B. in Abb. 6.8 repräsentieren die Elemente vom ersten Ausgabedatenelement des Datenflusses $D_O(i)$ Studierenden, die aus Frankreich kommen und Medizin an der Universität Heidelberg studieren.

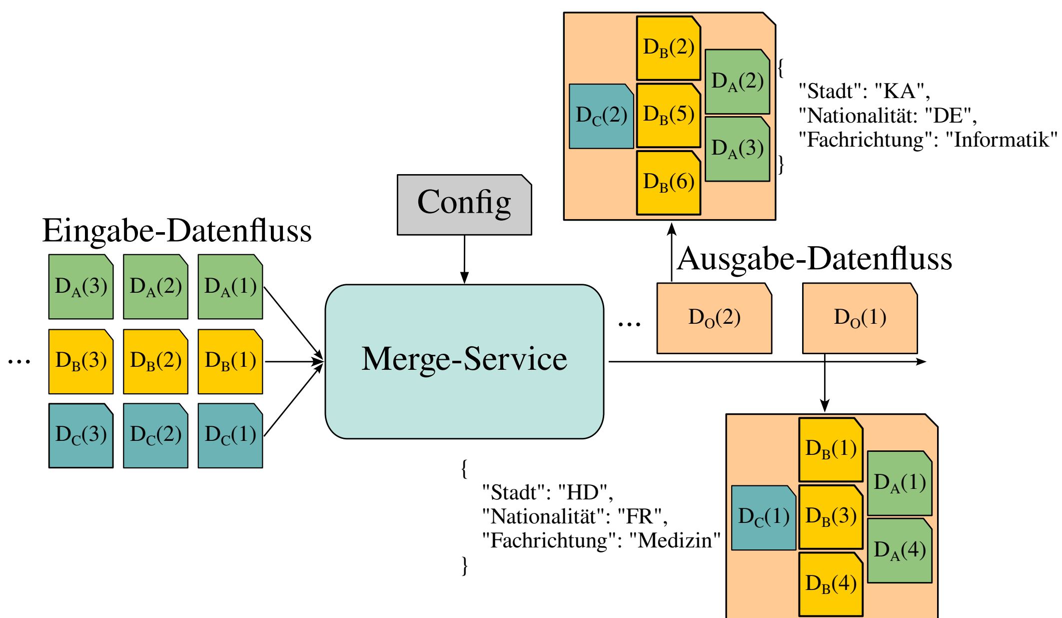

Abb. 6.8.: Strategie 4: Basierend auf mehreren Attributen, z.B. Stadt, Nationalität, Fachrichtung & die Anzahl der Elemente von den Datenflüssen A, B und C ist ungleich

Strategie 5

Basierend auf Strategie 4 hat Strategie 5 einen zusätzlichen limitierenden Faktor. Die Elemente vom Datenströmen $D_A(i)$ und $D_B(i)$ dürfen nicht mehr gemeinsam kombiniert, sondern jeweils mit den Elementen vom Datenstrom $D_C(i)$ wie in Abb. 6.9 dargestellt gruppiert werden. Das heißt, Elemente im Ausgangsdatenstrom können unterschiedliche Kombinationen von Elementen aus Eingangsdatenströmen sein.

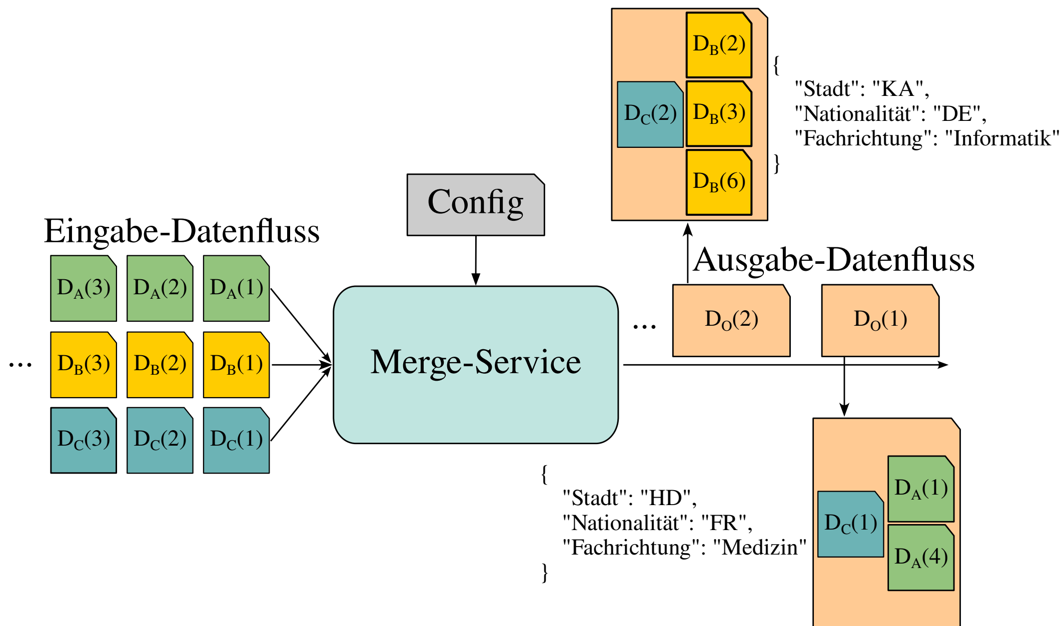

Abb. 6.9.: Strategie 5: Basierend auf mehreren Attributen, z.B. Stadt, Nationalität, Fachrichtung & verschiedene Kombinationen von Elementen aus Datenflüssen

6.2.3 Deskriptive Sprache zur Definition der Strategien

Zur Definition der vorherigen vorgestellten Strategien wird eine deskriptive, JSON-basierte Sprache in PROOF bereitgestellt. Damit werden Strategien in einer Config-Datei vom Merge-Service dargestellt. In Abb. 6.10 wird die grundlegende Struktur der deskriptiven Sprache angezeigt:

- **data_flows**
  Dieses Attribut beschreibt drei wichtige Informationen von jedem Datenfluss, nämlich:
  1. **id**: ID eines Datenflusses

2. **number_of_data_elements**: Anzahl der Datenelemente eines Datenflusses zum Zusammenführen mit anderen Datenflüssen
3. **keys_to_store**: Schlüsselwerte zur Speicherung aller Datenelemente eines Datenflusses

- **is_sequential**
  Diese Anweisung stellt fest, ob das FIFO-Prinzip beim Zusammenführen eingehalten wird. Falls der Wert
  * **1** ist, wird das am längsten wartende Element aus einer Warteschlange als nächstes bearbeitet.
  * **0** ist, wird das FIFO-Prinzip nicht mehr angepasst. Stattdessen beruht das Zusammenführen auf den Merkmalen, die im Attribut *keys_to_combine* vordefiniert werden, von Elementen der zu kombinierenden Datenflüsse.
- **combinations**
  Dieses Attribut stellt vor, wie Datenflüsse miteinander kombiniert werden. Hierzu wird in jeder Kombination festgelegt:
  - **ids_of_data_flows**: IDs der zu kombinierenden Datenflüsse
  - **keys_to_combine**: Schlüsselwerte definieren, wonach sich Datenflüsse zusammenführen lassen

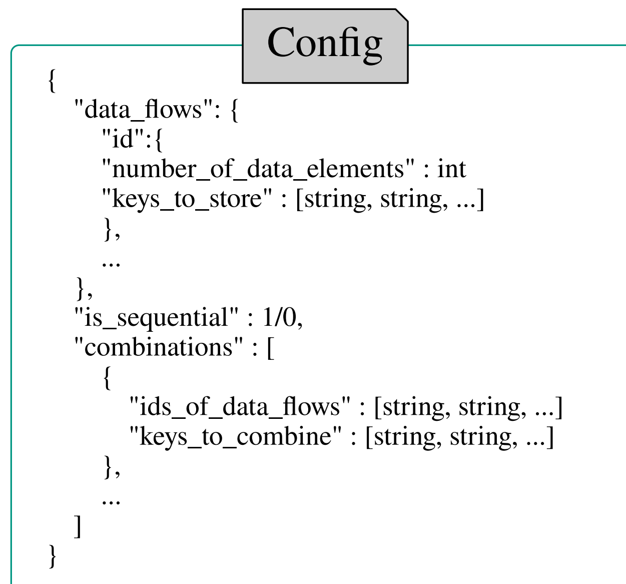

Abb. 6.10.: Deskriptive Sprache zur Definition der Strategien für den Merge-Service

Im Folgenden ist zu klären, wie die im vorherigen Abschnitt dargestellten fünf Strategien mit der deskriptive Sprache formuliert werden, um für den Merge-Service verständlich zu sein.

Definition der Strategie 1

Wie im Abschnitt 6.2.2 beschrieben, stellt Strategie 1 den einfachsten Fall FIFO zum Zusammenführen vor. Jedes Mal wird nur ein Datenelement jedes Datenflusses bearbeitet und ausgegeben. Wie in Abb. 6.11 dargestellt, sind hierzu in der Config-Datei lediglich drei Attributwerte festzustellen:

- Die IDs der drei Datenflüsse **A**, **B** und **C** im Attribut „data_flows“ aufzulisten,
- den Wert von „number_of_data_elements“ jedes Datenflusses auf **1** zu definieren,
- und den Wert von „is_sequential“ auf **1** festzusetzen

Die Werte der anderen Attribute bleiben leer.

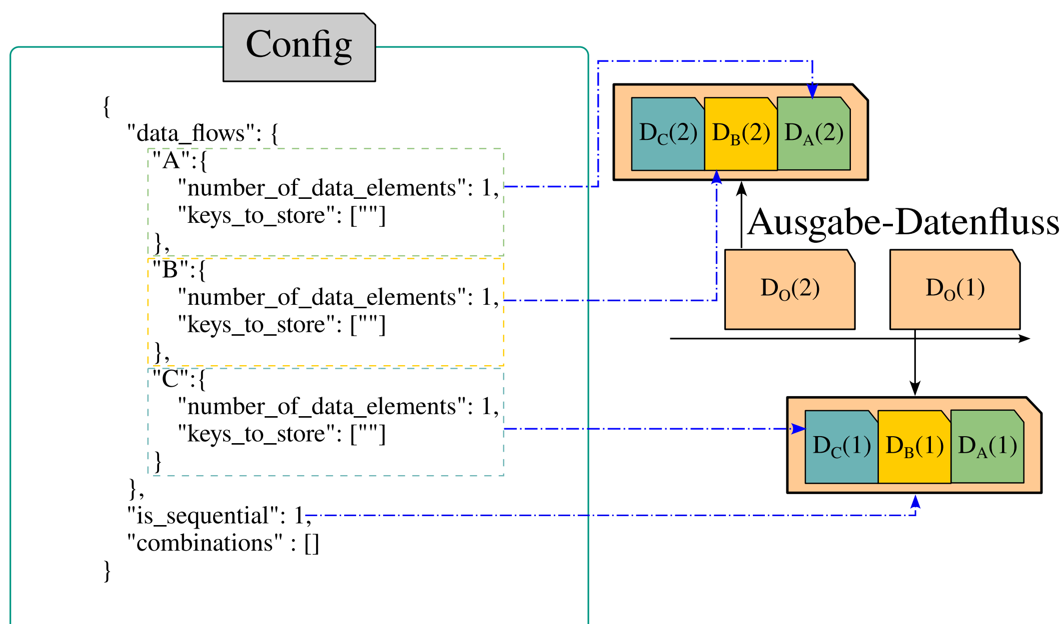

Abb. 6.11.: Definition der Strategie 1

Definition der Strategie 2

Anders als bei Strategie 1 enthalten die Output-Elemente bei Strategie 2 unterschiedlich viele Input-Elemente aus den 3 Datenströmen. Basierend auf der Definition der Strategie 1 ist nur der Wert von „number_of_data_elements“ jedes Datenflusses anzupassen (siehe Abb. 6.12):

- **A**: number_of_data_elements = **2**
- **B**: number_of_data_elements = **3**
- **C**: number_of_data_elements = **1**

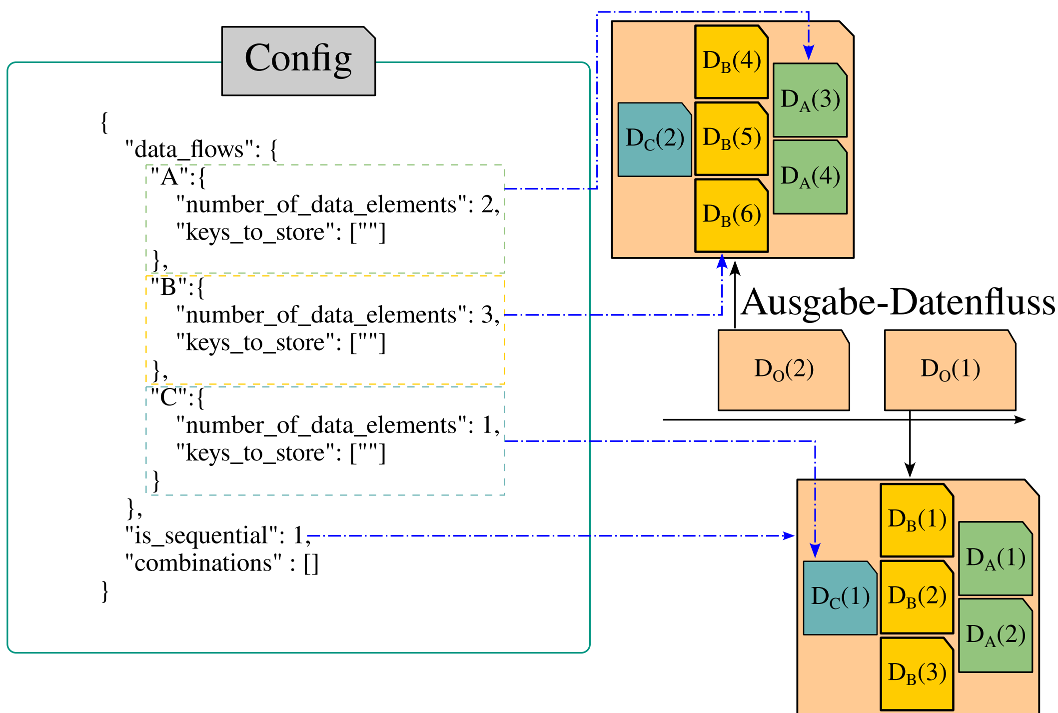

Abb. 6.12.: Definition der Strategie 2

Definition der Strategie 3

Im Abschnitt 6.2.2 wurde dargestellt, dass das FIFO-Prinzip bei Strategie 3 nicht mehr eingehalten wird, sondern es spielt ein Elementmerkmal (ID, Eigenschaft usw.) beim Zusammenführen eine wichtige Rolle. Wie in Abb. 6.13 zu sehen ist, werden die Attribute „keys_to_store", „ids_of_data_flows" und „keys_to_combine" sowie „is_sequential" hierfür auf der Grundlage von der Config-Datei der Strategie 2 neu aufgestellt:

- keys_to_store = [**Stadt**]: **Stadt** als Schlüsselwert zur Speicherung aller Datenelemente jedes Datenflusses
- ids_of_data_flows = [**A**, **B**, **C**]: Liste von IDs aller Datenströme
- keys_to_combine = [**Stadt**]: **Stadt** als Schlüsselwert, wonach Datenelemente von den drei Datenflüssen zusammengeführt werden
- is_sequential = **0**: Das FIFO-Prinzip wird nicht mehr eingesetzt

Definition der Strategie 4

Bei Strategie 4 sind zwei weitere Zusatzbedingungen zur Kombination von Datenelementen im Vergleich zu Strategie 3 zu berücksichtigen. Wie in Abb. 6.14 dargestellt,

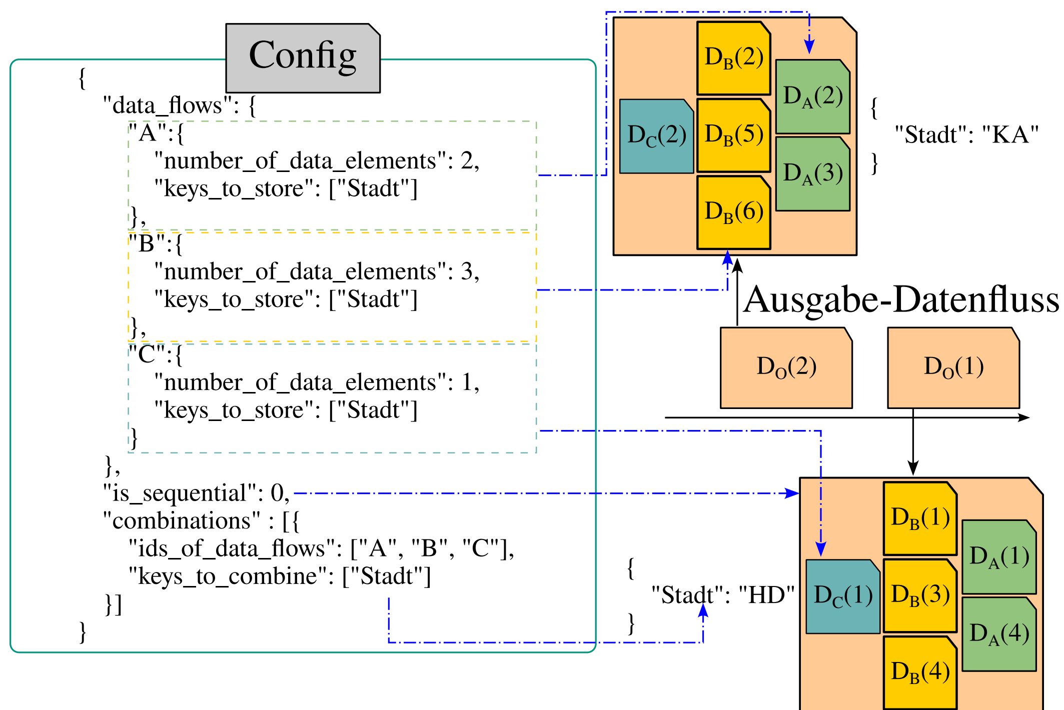

Abb. 6.13.: Definition der Strategie 3

werden die Attribute „keys_to_store“ und „keys_to_combine“ um zwei zusätzliche Eigenschaften ergänzt:

- keys_to_store = [Stadt, **Nationalität**, **Fachrichtung**]: Liste der drei Schlüsselwerte zur Speicherung aller Datenelemente jedes Datenflusses
- keys_to_combine = [Stadt, **Nationalität**, **Fachrichtung**]: Liste der drei Schlüsselwerte, wonach Datenelemente von den drei Datenflüssen zusammengeführt werden

Definition der Strategie 5

Als die komplizierteste Strategie hat Strategie 5 im Kontrast zu Strategie 4 eine weitere Beschränkung, dass nicht alle Datenflüsse, sondern nur teilweise in einem Datenfluss gemeinsam zusammengeführt werden, wie im Abschnitt 6.2.2 präsentiert. Deshalb werden zwei unterschiedliche Kombinationsoptionen in der Config-Datei der Strategie 5 in der Abb. 6.15 definiert:

- ids_of_data_flows = [**A**, **C**]: Elemente von den Datenflüssen **A** und **C** werden zusammengeführt
- ids_of_data_flows = [**B**, **C**]: Elemente von den Datenflüssen **B** und **C** werden zusammengeführt

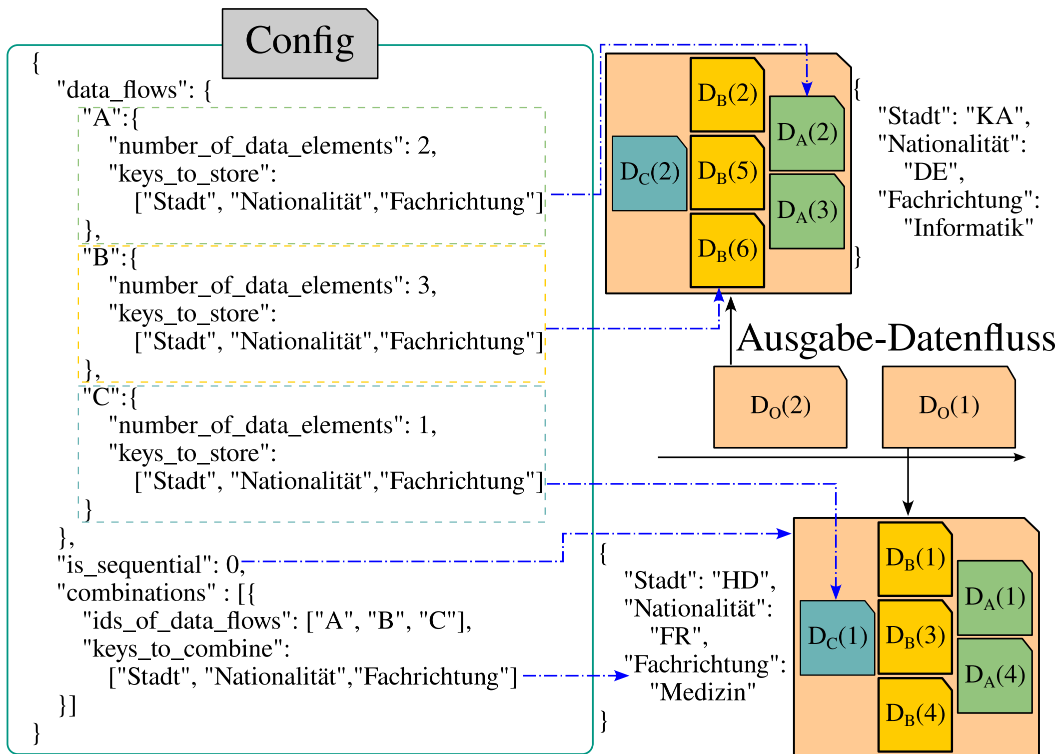

Abb. 6.14.: Definition der Strategie 4

Anhand dieser deskriptiven Sprache können verschiedene Strategien zum Zusammenführen von mehreren Datenflüssen in einem Datenfluss in einer Config-Datei formuliert werden. Eine Vorlage der deskriptiven Sprache wird Nutzern auf der Weboberfläche bereitgestellt, um ihre eigene Kombinationsstrategie zu definieren. Nach der Definition übernimmt der Merge-Service automatisiert die restlichen Aufgaben, die im Ablaufdiagramm 6.4 verdeutlicht wurden, z.B. Strategie analysieren, auflösen, danach Elemente von Datenflüssen empfangen, identifizieren, einsammeln, abspeichern und schließlich ausgeben.

Dieser Merge-Service wird für den Anwendungsfall Energy Hub Gas [Pop+21] verwendet, um sechs Datenflüsse zusammenzuführen. Mehr Details befinden sich im Abschnitt 7.2.3 in Kapitel 7.

6.3 Primary-Secondary-Architektur

Co-Simulation ist ein allgemeiner Ansatz zur Simulation gekoppelter technischer Systeme. Als Co-Simulationsstandards sind High-Level Architecture (HLA) [MP04]

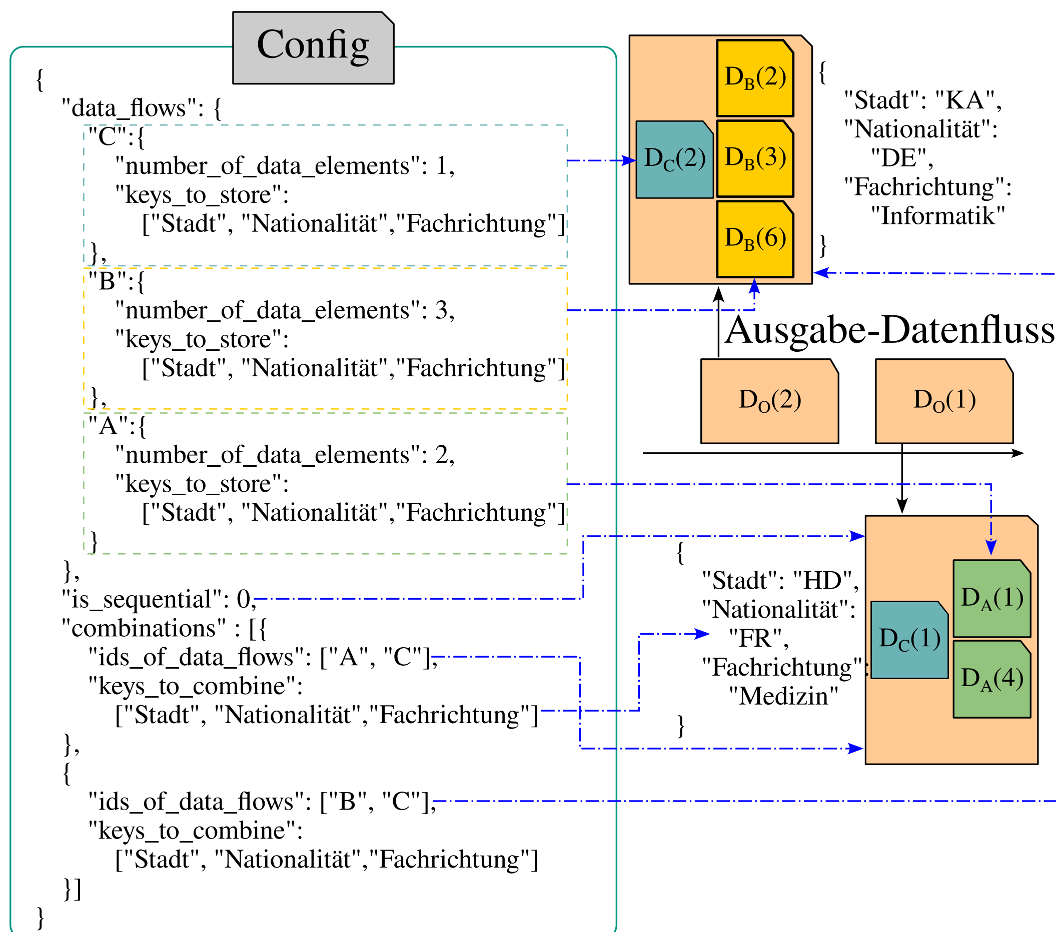

Abb. 6.15.: Definition der Strategie 5

(s. 3.2.7) oder Functional Mock-up Interface (FMI) [Blo+12] (s. 2.1) werkzeugunabhängig und definieren nicht-funktionale Interoperabilität über Metadatenformate für Modellaustausch, Datenmodelle, Protokolle der Anwendungsschicht oder Synchronisationsmuster, um Simulationsmodelle interoperabel zu machen [Alb+21; Gra+20]. Diese Standards verlassen sich häufig auf eine Primary-Secondary-Modellarchitektur, um eine Synchronisation zu erreichen [Ngu+17]. Subsimulatoren sind Secondary-Simulatoren, die von einem Primary-Algorithmus (dem Co-Simulationsalgorithmus) gesteuert werden. Die Subsimulatoren haben keine Informationen übereinander. Sie haben keine Kenntnis oder Kontrolle darüber, mit welchen anderen Subsimulatoren sie gekoppelt sind. Bei einem Primary-Secondary-Konzept simulieren die Secondary-Simulatoren Teilprobleme, während der Primary-Simulator sowohl für die Koordination der Gesamtsimulation als auch für die Datenübertragung zuständig ist.

6.3.1 Übersicht

In Abschnitt 4.3 wurde beschrieben, dass PROOF als eine erweiterte Plattform zur verteilten Ausführung von wissenschaftlichen Workflows konzipiert wurde, die sowohl Co-Simulation als auch verteilte Simulationen von Datenflüssen ausführen kann. Zur Unterstützung der Co-Simulationsstandards HLA und FMI wird eine Primary-Secondary-Architektur in PROOF daher realisiert, um Primary-Secondary-Co-Simulationen zu implementieren.

Hierzu wird ein Primary-Prozess, wie in Abb. 6.16 illustriert, konzipiert. Der Primary-Prozess besteht aus zwei wesentlichen Bestandsservices:

- **Schedule-Service**
  Dieser Service bearbeitet und speichert einen Fahrplan ab, um den Primary-Prozess zu steuern. Der Fahrplan wird in einer Schedule-Config-Datei definiert, die wie in Abb. 6.17 formatiert wird und vier Attribute umfasst:
  - **start_time** repräsentiert die Zeit, wann der Primary-Prozess gestartet werden kann.
  - **time_step** definiert, nach wie vielen festen Makrozeitschritten der Primary-Prozess die Steuerung für seine Secondary-Prozesse wiederholt, z.B. wie in Abb. 6.17 dargestellt, kommuniziert der Primary-Prozess mit seinen Secondary-Prozessen alle 5s zur Steuerung.
  - **end_time** legt die Zeit fest, wann der Primary-Prozess beendet werden kann.
  - **schedules** stellt Fahrpläne von allen Secondary-Prozessen fest. Gemäß den Fahrplänen kann der Primary-Prozess seine untergeordnete Secondary-Prozesse kontrollieren, wann sie ausgeführt oder geschlossen werden sollen und ihre feste Mikroschrittgröße der Operation.
- **Kommunikationsservice**
  Während der Schedule-Service die Informationen des Fahrplans behandelt, ist dieser Kommunikationsservice für den Aufbau der Kommunikation zwischen dem Primary-Prozess und seinen gekoppelten Secondary-Prozessen verantwortlich. Damit kann der Primary-Prozess sowohl die Fahrpläne an den untergeordneten Secondary-Prozessen weiterleiten als auch die Rückgaben ihrer Simulationen erhalten. Diese asynchrone Kommunikation ist zeitgesteuert und wird über Redis realisiert. Weitere Details zur Koordination wird im nächsten Abschnitt beschrieben.

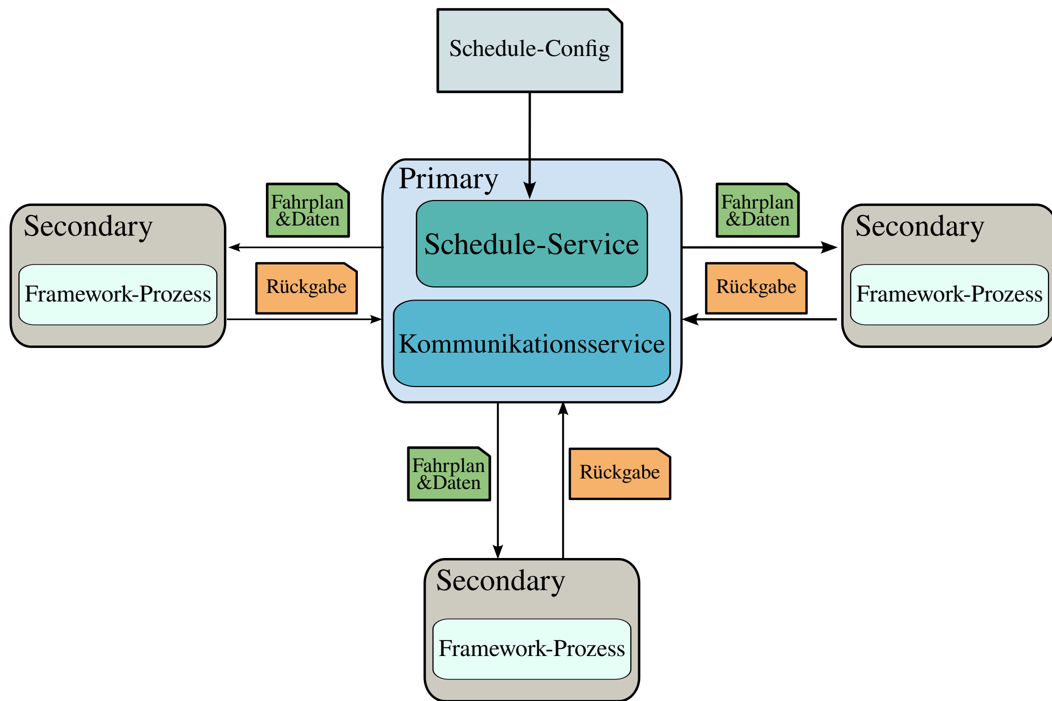

Abb. 6.16.: Primary-Secondary-Architektur

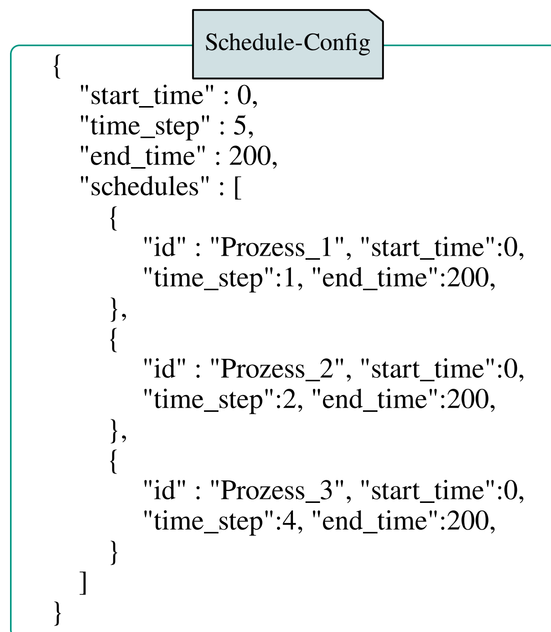

Abb. 6.17.: Schedule-Config

6.3.2 Koordination

Der Primary-Algorithmus geht in Makroschritten fester Schrittweite von der Startzeit bis zur Endzeit vor. Die Berechnung eines Zeitschritts von t_i nach t_{i+1} sowie t_{i+2} innerhalb von Zyklen erfolgt folgendermaßen:

- $\mathbf{t_i}$: Es wird angenommen, dass jeder Secondary-Prozess zum Zeitpunkt t_i vom Primary-Prozess gestartet wird und eine Annahme für seinen Eingangswert zum Zeitpunkt t_{i+1} trifft.
- $\mathbf{t_{i+1}}$: Zum Zeitpunkt t_{i+1} sendet der Primary-Prozess die Steuerungsbefehle an den freien Secondary-Prozessen, die mit ihren Simulationen/Berechnungen fertig sind und ihre Rückgabe bevor t_{i+1} zurückgegeben haben. Somit erfolgt die Synchronisierung und Aktualisierung der ausgetauschten Werte mit der berechneten Rückgabe am Ende des Zeitschritts t_{i+1}.
- $\mathbf{t_{i+2}}$: Die anderen Secondary-Prozesse, die ihre Berechnungen bevor t_{i+1} noch nicht abgeschlossen haben, werden bis zum Ende der Berechnungen weiter ausgeführt. Sobald sie ihre Rückgaben zurücksenden, warten sie ab, bis sie zum nächsten Makrozeitpunkt t_{i+2} die neuen Steuerungsbefehle erhalten und dann vom Primary-Prozess ausgeführt werden.

Dieser Primary-Secondary-Architektur wird für den Anwendungsfall Energy Hub Gas [Pop+21] eingesetzt, um elf Secondary-Prozesse schrittweise zu steuern. Weitere detaillierte Beschreibung des Anwendungsfalls befinden sich im Abschnitt 7.2.3 in Kapitel 7.

6.4 Zusammenfassung

In diesem Kapitel wurde der dritte Beitrag der vorliegenden Arbeit geliefert. Dabei wurden zunächst der Task-Parallelismus und der Daten-Parallelismus vom Parallel-Computing zur parallelen Verarbeitung mehrerer Aufgaben in PROOF vorgestellt. Bezogen auf den Daten-Parallelismus wurde ein Koordinationsservice implementiert, um mehrere Instanzen eines Framework-Prozesses in separaten Containern auf dem Computing-Cluster parallel ausgeführt werden zu können. Darüber hinaus ist der Service befähigt, Eingabedaten aus einer Warteschlange annehmen und in verschiedene Container bzw. Instanzen verteilen sowie Ausgabedaten von Containern sammeln zu können. Mithilfe dieser Services wird sichergestellt, die Ressourcen und Hardware (Computing-Cluster) effizient zu nutzen, um die Leistungsfähigkeit von Aufgaben zu verbessern.

Darauf aufbauend schilderte dieses Kapitel das grundlegende Konzept und das Ablaufdiagramm des Merge-Services im Hinblick auf die Synchronisierung mehrerer Datenflüsse. Dabei wurde ausführlich erläutert, wie mehrere Datenströme auf fünf verschiedene Arten (Strategien) kombiniert werden können. Zur Definition und Realisierung der fünf Strategien wurde eine leicht verständliche deskriptive Sprache entwickelt.

Schließlich widmete sich das Kapitel der Architektur der Primary-Secondary-Co-Simulation. Hierzu wurde der Primary-Prozess zur Steuerung seiner untergeordneten Secondary-Prozesse dargestellt. Innerhalb des Primary-Prozesses wurden zwei wesentliche Services implementiert, nämlich der Schedule-Service zur Verarbeitung der Fahrpläne und der Kommunikationsservice zur Erstellung der Interaktion zwischen dem Primary-Prozess und den Secondary-Prozessen.

Zusammenfassend werden die nachstehenden Anforderungen 3.1.2 vom dritten Beitrag erfüllt:

A8 **Skalierung und Parallelisierung**
Durch die Verwendung des Koordinationsservice sind Workflows in PROOF leicht zu skalieren und parallelisieren, um Performance zu steigern.

A9 **Generische Funktionalitäten**
Der Merge-Service wird als Hilfsbaustein für die generische Funktionalität zur Synchronisierung und zum Zusammenführen von mehreren Datenströmen bereitgestellt. Durch die Verwendung des Primary-Prozesses kann Primary-Secondary-Co-Simulationen aufgebaut werden.

7 Evaluierung

In diesem Kapitel sollen die in vorherigen Kapiteln erarbeiteten Systemarchitekturen und vorgestellten drei Beiträge dieser Arbeit evaluiert werden. PROOF ermöglicht ein systematisches und nachvollziehbares Vorgehen zum Entwerfen, Aufbauen und Ausführen von wissenschaftlichen Workflows. Die Verwendung von Technologien, Architekturen und Softwaretools sowie die Konzeption von Funktionalitäten während der einzelnen Phasen des Entwurfsprozesses gewährleisten, dass PROOF den geforderten Qualitätsaspekten gerecht wird. Dazu zählen beispielsweise die Containerisierung und die Container-Orchestrierungstechnologie, die die Integration der Applikationen und letztlich die Wiederverwendbarkeit, Parallelisierung sowie Skalierung von wissenschaftlichen Applikationen bzw. Workflows fördern.

Die Validierung der Beiträge wird durch die Anwendung von PROOF im Rahmen verschiedener Fallstudien durchgeführt. Dabei werden im Folgenden die Beschreibung, die Implementierung und die Analyse von drei Anwendungsfällen ausführlich dargestellt. Weiterhin wird der Prototyp des Frameworks durch ein Benchmarking in Bezug auf seine Praxistauglichkeit und die Erfüllung der Anforderungen an seine technischen Laufzeiteigenschaften, wie Speichernutzung, Performance, Skalierbarkeit usw. evaluiert. Auf Basis der erläuterten Anwendungsfälle und des Benchmarkings werden Merkmale analysiert, welche die PROOF-Architektur bezüglich der Qualität beurteilen.

Dieses Kapitel ist wie folgt strukturiert: Abschnitt 7.1 erläutert den Fallstudienprozess im Software-Engineering. Danach präsentiert Abschnitt 7.2 die Implementierung der drei Anwendungsfälle in PROOF. Im Abschnitt 7.2.5 werden alle implementierten PROOF-Prozesse in der Prozess-Bibliothek zusammengefasst. Abschnitt 7.3 fokussiert anschließend ein Benchmarking zur Messung der Speichernutzung, der Ausführungszeit im Vergleich zur direkten Ausführung und des Overheads für parallele Aufgaben in PROOF. Abschnitt 7.4 widmet sich der Qualitätsanalyse. Im Abschnitt 7.5 wird die Evaluation der Machbarkeit zur Prüfung auf Vollständigkeit der Softwarefunktionen gegeben. Den Abschluss des Kapitels bildet eine Zusammenfassung der Evaluierung.

7.1 Methodik der Fallstudie

Fallstudien sind eine häufig verwendete Forschungsstrategie in Bereichen wie Psychologie, Soziologie, Politikwissenschaft, Sozialarbeit, Wirtschaft und Gemeindeplanung (z.B. [RM17; Sta95; Yin09]. In diesen Bereichen werden Fallstudien mit dem Ziel durchgeführt, nicht nur das Wissen zu erweitern (z.B. Wissen über Einzelpersonen, Gruppen und Organisationen und über soziale, politische und verwandte Phänomene), sondern auch eine Veränderung des untersuchten Phänomens herbeizuführen (z.B. Verbesserung von Bildung oder Sozialfürsorge). Software-Engineering-Forschung hat ähnliche hochrangige Ziele, nämlich besser zu verstehen, wie und warum Software-Engineering durchgeführt werden sollte, um mit diesem Wissen zu versuchen, den Software-Engineering-Prozess und die resultierenden Softwareprodukte zu verbessern.

In [Run+12] wird eine Fallstudie (Case Study) fürs Software-Engineering definiert:

> „*Die Fallstudie im Software-Engineering ist eine empirische Untersuchung, die sich auf mehrere Beweisquellen stützt, um eine Instanz (oder eine kleine Anzahl von Instanzen) eines zeitgenössischen Software-Engineering-Phänomens in seinem realen Kontext zu untersuchen. Fallstudien zum Software-Engineering tendieren zu einer positivistischen Perspektive, insbesondere für erklärende Studien. Dies hängt mit der pragmatischen Natur empirischer Software-Engineering-Forschung zusammen, bei der die praktischen Implikationen einer bestimmten Praxis relevanter sind als die Fragen nach abstrakten philosophischen Prinzipien.*“

7.1.1 Fallstudienprozess

Beim Fallstudienprozess müssen vier Hauptprozessschritte berücksichtigt werden [Run+12]:

1. **Entwurf**
   Beim Entwurf einer Fallstudie werden eine Reihe von Elementen berücksichtigt:
   - Ziel: Es ist zu klären, was mit der Studie für ein Software-Framework erreicht werden soll.
   - Fallstudientyp: Bei der **Einzelfallstudie** wird ein Anwendungsfall in einem Kontext durch die Verwendung verschiedener Testdaten untersucht,

um ein Software-Framework zu bewerten. Bei der **Mehrfachfallstudie** werden mehrere Anwendungsfälle von unterschiedlichen Szenarien implementiert, um die Qualität eines Software-Frameworks zu analysieren.

- Anwendungsfall: Beschreibung des Anwendungsfalls
- Konzept: Konzept des Anwendungsfalls

2. **Datenerfassung**
   Laut Lethbridge et al. [LSS05] können Datenerfassungstechniken in drei Grade eingeteilt werden:
   - Erster Grad: Dies sind direkte Methoden, bei denen die Forschenden in direktem Kontakt mit den Befragten steht und Daten in Echtzeit sammelt (z.B. persönliches Gespräch).
   - Zweiter Grad: Dies sind indirekte Methoden, bei denen die Forschenden Rohdaten direkt sammelt, ohne während der Datenerfassung tatsächlich mit den Befragten zu interagieren (z.B. Verwendung eines Softwaretools zur Datenerfassung).
   - Dritter Grad: Das sind Methoden, bei denen die Forschenden selbstständig bereits vorhandene Arbeitsartefakte analysiert (wenn z. B. Lastenhefte und Fehlerberichte analysiert werden).
3. **Datenanalyse**
   Die Datenanalyse wird für quantitative und qualitative Daten unterschiedlich durchgeführt. Das grundlegende Ziel der Datenanalyse ist es, Schlussfolgerungen aus den Daten abzuleiten und diese mithilfe der Daten präzise zu belegen. Im Software-Engineering kann durch die Datenanalyse überprüft werden, ob eine Berechnung, Simulation oder Emulation fehlerfrei und richtig ausgeführt wird.
4. **Bericht**
   Basierend auf den Richtlinien von Jedlitschka und Pfahl [JP05] ist die resultierende Struktur eines Berichts in [Run+12] dargestellt:
   - Title
   - Authorship
   - Structured abstract
   - Introduction
   - Related work
   - Case study design: Objektive, Description, Concept etc.
   - Results

- Conclusions and future work
- Acknowledgments
- References
- Appendices

7.1.2 Gegenstand der Fallstudie

In dieser Arbeit werden PROOF und die damit einhergehenden Beiträge mittels des Fallstudienprozesses evaluiert.

1. **Entwurf**
   Zur Mehrfachfallstudie wurden verschiedene Anwendungsfälle in PROOF implementiert. In dem folgenden Abschnitt 7.2 werden das Ziel, die Beschreibung, das Konzept und die Implementierung von drei ausgewählten Anwendungsfällen präsentiert.
2. **Datenerfassung**
   Durch die Interviews, Tagungen oder virtuellen Meetings erfolgte die Datenerfassung direkt (erster Grad) bei den Ansprechpersonen für die entsprechenden Anwendungsfälle.
3. **Datenanalyse**
   Für jedes Projekt bzw. jeden Anwendungsfall wurden Informationen zu geplanten und tatsächlichen Zeitplänen, Phasen, Meilensteinen, Entscheidungen und Neuplanungen identifiziert und organisiert. Die Ergebnisse der Anwendungsfälle wurden in PROOF gesammelt und bei Interviews oder virtuellen Meetings den Ansprechpersonen der Anwendungsfälle zur Analyse auf Korrektheit gegeben.
4. **Bericht**
   Während und nach den Fallstudien in PROOF wurde eine Vielzahl von Berichten erstellt. Diese enthielten:
   - Wissenschaftliche Konferenzpaper, die veröffentlicht wurden
   - Wissenschaftliche Journalpaper, die publiziert wurden
   - Fachberichte, die während der Promotion und seit deren Abschluss fertiggestellt wurden
   - Zwischenberichte, die zur Überprüfung durch Projektansprechpersonen, Forschungsmitarbeitende und das Forschungsteam erstellt wurden

- Feedback-Berichte, die zu den vorläufigen Endergebnissen der jeweiligen Projekte zusammengefasst wurden. Diese boten nicht nur Gelegenheit zur Qualitätssicherung der Fallstudien, sondern auch für die jeweiligen Projekte, ihre Projekte zu reflektieren und aus den Feedback-Berichten und dieser Reflexion zu lernen

7.2 Anwendungsfälle

In den folgenden Studien wurde PROOF bei drei Energiesystemprojekten angewandt, um vor allem die Funktionalitäten von PROOF zu validieren. Anschließend werden alle implementierten PROOF-Prozesse in der Prozess-Bibliothek für verschiedene Anwendungsfälle zusammengefasst.

7.2.1 Welder et al.: Spatio-temporal optimization of a future energy system for power-to-hydrogen applications in Germany

Der Anwendungsfall führte Berechnungen zur optimalen Dimensionierung eines überregionalen Energiesystems unter Berücksichtigung konventioneller und erneuerbarer Energiequellen sowie verschiedener Speicher-, Umwandlungs- und Transporttechnologien.

Ziel

Das Ziel des Anwendungsfalls ist es, die grundlegenden Funktionalitäten von PROOF zum Aufbau und Ausführen von Workflows zu testen, z.B. Parameterkonfiguration auf der Weboberfläche, Modellintegration, Datenaustausch, Automatisierung der Ausführung der Modelle, Wiederverwendbarkeit der integrierten Modelle von der Prozess-Bibliothek, usw. Außerdem werden die Funktionalitäten *Koordination* und *Parallelisierung* untersucht.

Übersicht

In Abb. 7.1 wird die Gestaltung eines wasserstoffbasierten **E**nergy **S**ystem **M**odeling (ESM) in Deutschland dargestellt [Wel+18]. Als Eingabe erhält dieses Modell Annahmen, wie die Produktion aus Windressourcen simuliert werden soll sowie markt- und technoökonomische Eingaben. Alle Eingaben werden in Form verschiedener

CSV-Dateien bereitgestellt. Zu Beginn werden die in den Sub-Workflows gezeigten Modelle für jede der Regionen, die in das Optimierungsschema aufgenommen werden sollen, iteriert. Durch einmaliges Abschließen dieses Sub-Workflows für jede Region werden die vom Modell *Energy System Optimization* (ESO) benötigten Eingaben erstellt.

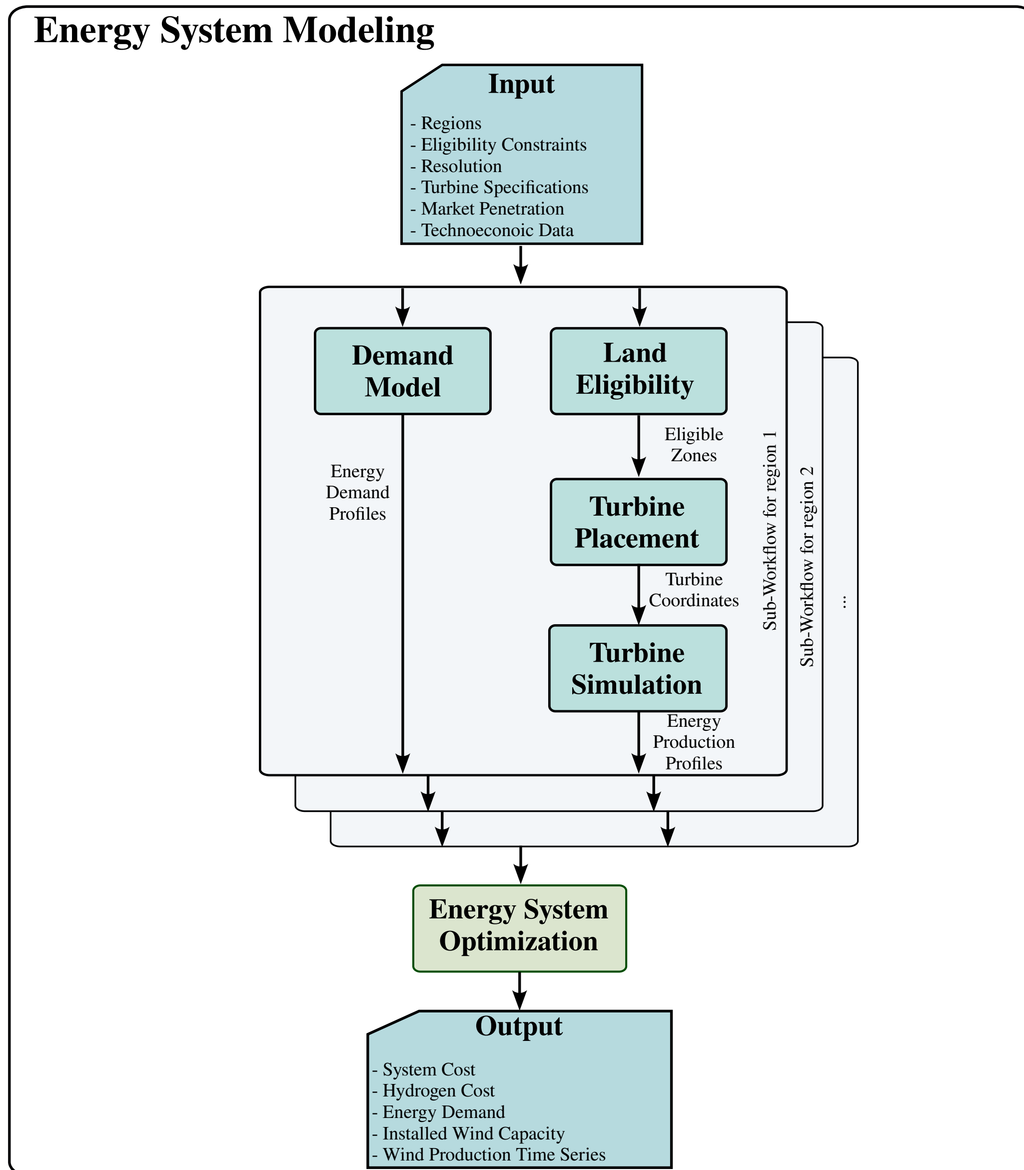

Abb. 7.1.: Übersicht vom ESM [Liu+18]

Das *Demand*-Modell erhält als Eingabe das betrachtete Jahr 2050 sowie einen Faktor, der die Marktdurchdringung von Wasserstoff-Brennstoffzellenautos im Bereich der Verbrauchermobilität darstellt (angenommen 75%). Das Modell projiziert dann die Bevölkerung der Region auf das angegebene Jahr und verwendet Statistiken zum

Mobilitätsenergieverbrauch und zur Leistung von Brennstoffzellenfahrzeugen, um den Gesamtenergiebedarf zu schätzen, der dann als CSV-Datei ausgegeben wird. Eine ausführliche Beschreibung findet sich in [Rob15; Rob+17].

Das *Land-Eligibility*-Modell benötigt als Eingabe soziotechnische Annahmen bezüglich der Gebiete der betreffenden Region, die die Installation von Windkraftanlagen verbieten sollten. Das umfasst die Werte für den Mindestabstand zu Straßen, Siedlungen und Naturschutzgebieten. Als Ergebnis zeigt dieses Modell die verbleibenden Regionen an und gibt diese Informationen als *GeoTiff*-Datei aus. Eine ausführliche Beschreibung und der Open-Source-Code für dieses Modell finden sich in [For20; RRS17].

Mit den verfügbaren, infrage kommenden Gebieten bestimmt das *Turbine-Placement*-Modell die Koordinaten für die maximale Anzahl von Turbinen, die platziert werden können, vorbehaltlich der Eingabe eines minimalen Abstands zwischen Turbinen. Diese Ausgabe wird in Form einer CSV-Datei bereitgestellt. Damit liest das *Turbine-Simulation*-Modell diese Koordinaten und erhält einen Pfad zu Klimadaten, die für die Simulation jeder Turbine verwendet werden können. Nach der Simulation gibt dieses Modell dann die (Energie)Produktion jeder Turbine in Form einer Zeitreihe als binäres Python-Objekt aus. Nachdem diese Modelle für jede der Regionen eingesetzt wurden, in denen die Optimierung durchgeführt wird, wird das ESO ausgeführt.

Das ESO-Modell erwartet als Input stündliche Zeitreihenwerte für den Gesamtenergiebedarf und die durchschnittliche Turbinenproduktion für jede Region sowie Kostenannahmen für Windturbinen, eine Wasserstoffpipeline, Elektrolyseure, Speicheroptionen und neue Stromnetzleitungen.

Als Output liefert dieser Workflow die Gesamtsystemkosten, die Kosten (pro kg) für Wasserstoff, den Gesamtenergiebedarf für jedes Bundesland, die gesamte installierte Windleistung und schließlich die Zeitreihe der Windproduktion. Diese Ausgabe wird in Form eines binären Python-Objekts für die Zeitreihendaten bereitgestellt.

Implementierung

Zur Automatisierung und Evaluierung des beschriebenen ESM-Anwendungsfalls, der im Abb. 7.1 illustriert wird, wurde die Integration des gesamten Worfklows in PROOF durchgeführt. Alle Modelle sind typischerweise Python-Skripte und kommunizieren über Dateien miteinander. Daher wird jedes Modell mit seinem Dateiadapter, Python-Abhängigkeiten und dem Python-Skript mit Ausführungsbefehlen in ein Docker-Image gepackt und dann in Docker-Containern ausgeführt. Fünf neue NiFi-Prozessoren werden dafür implementiert und der Prozess-Bibliothek zur späteren

Verwendung hinzugefügt. Zwei Standard-NiFi-Prozessoren *ConsumeKafka-Prozessor* und *PublishKafka-Prozessor* werden zum Verarbeiten von Eingabedaten und Ergebnissen des Workflows verwendet. Sobald alle benötigten NiFi-Prozessoren auf der PROOF-Oberfläche eingefügt, konfiguriert und miteinander verbunden wurden, wird ein zu untersuchender Workflow für den ESM-Anwendungsfall fertig aufgebaut (s. Abb. A.7 in Anhang A.9.1).

Durch Klicken auf den Start-Button wird der gesamte Workflow automatisch gestartet. Der *ConsumeKafka-Prozessor* generiert Eingabedaten und überträgt sie an den *ESMLandEligibilityProcessor* und den *ESMEnergyDemandProcessor*. Der *ESMEnergyDemandProcessor* berechnet dann die jährlichen Energiebedarfsprofile, die auf Stundenebene verteilt an den *ESMEnergySystemOptimizationprocessor* weitergegeben werden. Unterdessen identifiziert der *ESMLandEligibilityProcessor* die Daten, die alle infrage kommenden Zonen innerhalb einer Region enthalten, die möglicherweise zur Installation von Windkraftanlagen verwendet werden können. Der *ESMTurbinePlacementProcessor* ermittelt anhand der infrage kommenden Flächen die Koordinaten für die maximale Anzahl von platzierbaren Turbinen. Mit den vorliegenden Koordinaten nimmt der *ESMTurbineSimulatingProcessor* an, dass an jedem potenziellen Standort eine Windkraftanlage aufgestellt wird und simuliert jährliche Windzeitreihen-Produktionsprofile jeder Anlage und gibt sie als binäres Python-Objekt aus. Anschließend wird mit den Ausgangsdaten von *ESMEnergyDemandProcessor* und *ESMTurbineSimulatingProcessor* eine Optimierung durch den *ESMEnergySystemOptimizationProcessor* betrieben, um Auskunft über die optimal zu installierenden Windkapazitäten und die besten Profile für deren Auslastung zur Deckung des Energiebedarfs zu geben. Schließlich können die Ergebnisse des gesamten Workflows (z.B. Systemkosten, installierte Windleistung, Windproduktionszeitreihen) vom *PublishKafka-Prozessor* empfangen werden.

Bewertung

Ein Schulungs-Workshop zur Installation und Anwendung von PROOF wurde für die Ansprechpersonen des IEK-3 (Institut für Energie- und Klimaforschung des Forschungszentrums Jülich) durchgeführt. Die Schulung erfolgte entsprechend der Durchführung des geschilderten Szenarios. Dabei wurde gezeigt, wie die Modelle des Szenarios integriert, der Workflow des Szenarios aufgebaut und automatisiert ausgeführt, wie die Eingabe an den Workflow übergeben und wie die Ausgabe des Workflows gesammelt wurde. Insbesondere konnte der Sub-Workflow durch die Konfiguration des Parameters *Concurrent Tasks* für mehrere Regionen parallelisiert werden, um mehrere Eingaben gleichzeitig zu verarbeiten. Am Ende des Workshops

wurde der einfache Erstellungsprozess von Workflows und vor allem dessen vollautomatisierte Ausführung besonders positiv bewertet. Darüber hinaus bringt die Fähigkeit zur Koordination und Parallelisierung den Vorteil, dass die Effizienz des gesamten Workflows signifikant verbessert wird.

Nach dem Workshop wurden ein Konferenzpaper [Liu+18] und ein Journalpaper [Liu+19] als Berichte zur Beschreibung des Konzepts und der Funktionalitäten von PROOF und des Szenarios mit den Teilnehmern veröffentlicht. In Kapitel *Discussion* des Journalpapers wurde die praktische Relevanz dieser Arbeit mit „sehr hoch" bewertet.

7.2.2 González-Ordiano et al.: Probabilistic forecasts of the distribution grid state using data-driven forecasts and probabilistic power flow

Ein neuer Ansatz wurde vorgestellt, der datengetriebene probabilistische Prognosen und probabilistischen Leistungsfluss kombiniert, um probabilistische Vorhersagen über den Zustand eines Verteilnetzes zu erhalten.

Ziel

Neben dem Testen der Grundfunktionen wird in diesem Anwendungsfall gezielt die Funktion zum Extrahieren von Daten aus der eASiMOV-Datenbank genutzt. Darüber hinaus werden drei verschiedene Anwendungstypen integriert, nämlich zwei Python-Skripte, ein Matlab-Modell und zwei Julia-Modelle.

Übersicht

In [Gon+21] wird eine neue Methode vorgestellt, die nicht nur probabilistische Vorhersagen des Netzzustands erhalten kann, sondern auch die Schätzung gemeinsamer Ereigniswahrscheinlichkeiten erlaubt, ohne eine gemeinsame Wahrscheinlichkeitsverteilung zu verwenden. Die Methode kombiniert nichtparametrische probabilistische Vorhersagen der Wirkleistungseinspeisung aller Knoten mit **P**robabilistic **P**ower **F**low (PPF) [Bor74; CCB08; PJ17; Ram+20]. Unter Verwendung von **Q**uantile **R**egressions (QRs) [KH01; Gon+20] werden nichtparametrische probabilistische Vorhersagen der Wirkleistung an jedem Knoten berechnet. Anschließend wird die Leistungsunsicherheit auf die verbleibenden Variablen propagiert, um das PPF-Problem zu lösen.

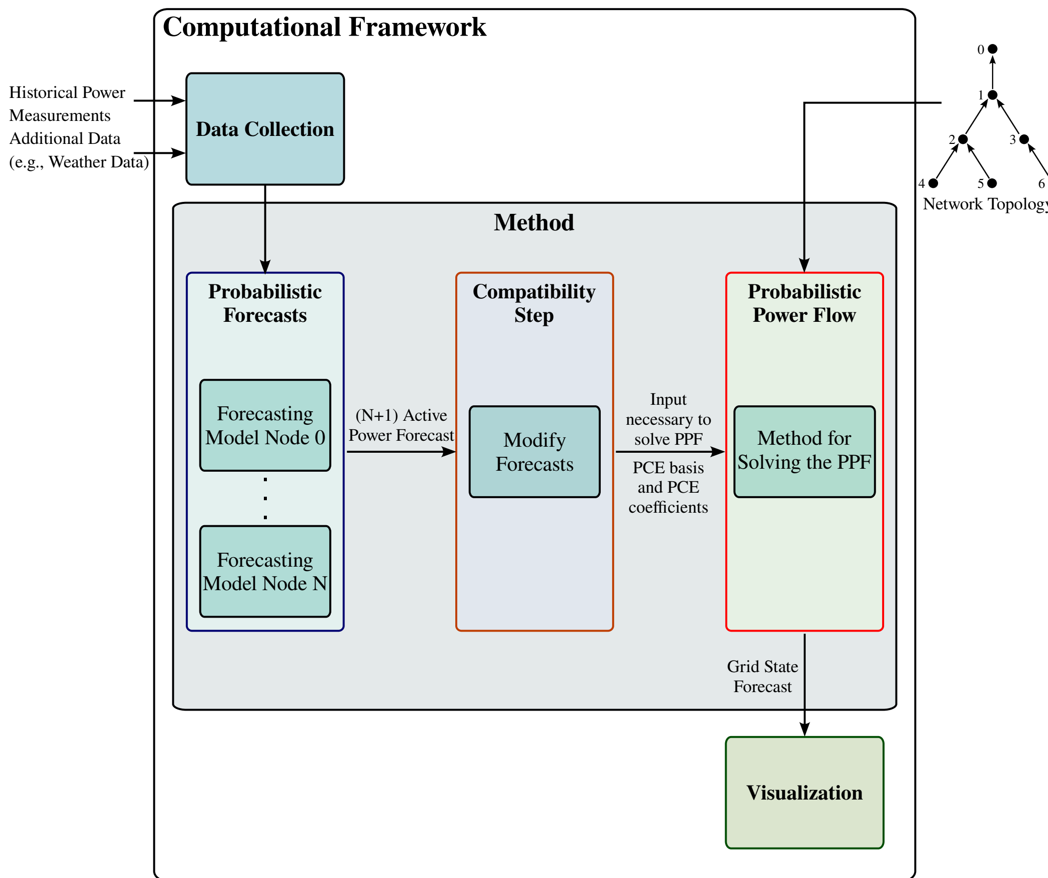

Abb. 7.2.: Übersicht der Methode. PPF: **P**robabilistic **P**ower **F**low; PCE: **P**olynomial **C**haos **E**xpansion [Gon+21]

In Abb. 7.2 wird die Übersicht über den implementierten Workflow für die Methode dargestellt. Dieser Workflow umfasst neben der vorgestellten Methode auch weitere Schritte, die für die Erhebung der Eingabedaten und für die Visualisierung der Ergebnisse notwendig sind. Wie Abb. 7.2 zeigt, sammelt der erste Schritt alle notwendigen Informationen, die die vorgestellte Methode als Eingabe verwendet: z.B. Wirkleistungsmessungen, Wetterdaten usw. Hierzu werden die Daten in Form von Stromverbrauchs-/Erzeugungszeitreihen über das eASiMOV-Framework [Kye+17] extrahiert. Das eASiMOV-Framework bereitet die Daten qualitätssteigernd vor und stellt sie über einen REST-Service in verschiedenen Aggregationsstufen und Zeiträumen zur Verfügung. Anschließend werden die gesammelten Daten an die vorgestellte Methode weitergegeben, die in drei Schritte unterteilt ist. Der erste Schritt verwendet die gesammelten Daten, um probabilistische Vorhersagen der zukünftigen Wirkleistung an jedem Knoten zu erhalten. Diese Prognosen werden mit einem datengetriebenen Ansatz basierend auf QR und einer in [Gon+20] beschriebenen Methode gewonnen.

Der zweite Schritt kann als Adapter bezeichnet werden und stellt sicher, dass die Ausgabe von Schritt eins mit der Eingabe von Schritt zwei kompatibel ist, wofür die Daten teilweise umgewandelt werden müssen. Schließlich löst der dritte Schritt das Problem des PPF unter Verwendung der Netzwerktopologie und des in [App+17] beschriebenen Ansatzes. Das Ergebnis all dieser Schritte ist die gewünschte probabilistische Prognose des Vollzustands des Verteilnetzes. Wie Abb. 7.2 zeigt, werden die Netzzustandsprognosen an einen weiteren Schritt weitergegeben, der ihre Visualisierung ermöglicht. Auch in diesem Schritt kommt wieder das eASiMOV-Framework zum Einsatz, um die Ausgabe abzuspeichern.

Implementierung

Die vorgestellte Methode ist ein komplexer Workflow, der aus einer Kombination von Schritten besteht, die in einer bestimmten Reihenfolge ausgeführt werden müssen und die mit einer Reihe von Softwaretools implementiert werden. Ein Screenshot des Workflows in PROOF wird in A.8 in Anhang A.9.2 angezeigt. Darin wurden sieben PROOF-Prozesse zum Aufbau des Workflows verwendet:

1. Der *ListenHTTP*-Prozess verwendet den eASiMOV-Service (ein Python-Skript) zur Datenerfassung aus der eASiMOV-Datenbank.
2. Der *Energylab_01_Forecast*-Prozess umfasst ein Matlab-Modell zur Berechnung probabilistische Vorhersagen.
3. Der *Energylab_02_Quantile2Distribution*-Prozess beinhaltet ein Matlab-Modell zum Modifizieren der probabilistischen Vorhersagen.
4. Der *Energylab_023_CreateStrainFiles*-Prozess wird als ein zusätzlicher Schritt zwischen den Kompatibilitäts- und den PPF-Schritten betrachtet. Dieser Schritt wurde mit einem Python-Skript implementiert und hat die Aufgabe, auf die Vorhersage mehrerer Knoten zu warten, um sie als Batch an den PPF-Schritt zu übermitteln. Dies wird vom Algorithmus zum Lösen des PPF benötigt.
5. Die nachfolgenden zwei Prozesse, *Energylab_03_CreateStochasticGerm* und *Energylab_04_PCEBFS_05_Postprocessing*, sind zwei Julia-Modelle zum Lösen des PPF-Problems nötig.
6. Der *InvokeHTTP*-Process benutzt auch den eASiMOV-Service zur Verbindung mit der eASiMOV-Datenbank, um die Ausgabe des Workflows für die Visualisierung abzuspeichern.

Bewertung

Für die Implementierung des komplexen Workflows in PROOF und die Bewertung der Anwendbarkeit von PROOF wurden von drei Modellierern und drei Software-Entwicklern mehrmals diskutiert. Dadurch wurden Vorteile von PROOF zum Ausführen des Workflows zusammengefasst.

PROOF verwendet modernste Technologien für die Laufzeitautomatisierung und Koordination von Teilaufgaben, wie z.B. Containervirtualisierung und verteilte nachrichtenorientierte Middleware. Mit anderen Worten, es definiert eine Architektur für die verteilte Prozessausführung und -koordinierung. PROOF verteilt automatisch Softwareanwendungen, die verschiedene Aufgaben ausführen, auf verschiedene Knoten innerhalb eines Clusters. Die Ausführung und der Datenaustausch des Workflows werden in PROOF vollautomatisiert und können mit unterschiedlichen Eingabedaten wiederholt werden. Ein weiterer Vorteil von PROOF besteht darin, dass bestimmte Aufgaben parallel ausgeführt werden können, um Leistungsfähigkeit von Workflows zu verbessern. Somit wird beim Einsatz eines Rechenclusters eine optimale Performance erreicht.

Als Bericht haben alle Teilnehmer ein Journalpaper [Gon+21] zur Beschreibung des Konzepts der Methode und der Implementierung der Methode in PROOF veröffentlicht. Die Vorteile, die PROOF zur Ausführung des Workflows brachte, wurden im Paper vorgestellt, wie z.B. automatisierte, koordinierte und parallelisierte Ausführung sowie effizienter Datenaustausch. Die Anwendbarkeit von PROOF wurde dadurch validiert.

7.2.3 Poppenborg et al.: Energy Hub Gas: A Multi-Domain System Modelling and Co-Simulation Approach

Energy **H**ub (EH) wurde eingesetzt, um netzdienliche Bereitstellung zu untersuchen.

Ziel

Im Anwendungsfall des sogenannten **E**nergy **H**ub wird der Primary-Prozess zum Aufbau von Primary-Secondary-Co-Simulationen genutzt. Weiterhin wird der Merge-Service zur Aggregation und Synchronisierung paralleler Datenflüsse gezielt getestet.

Übersicht

Das EH-Konzept wurde erstmals 2007 von [Gei+07] und [Gei07] eingeführt und ist wie folgt definiert: „*EHs bestehen aus Energieträgern, die mehrere Energieformen umwandeln, aufbereiten und speichern können. Abstrakter formuliert, definieren sie eine Blackbox mit Energieein- und -ausgängen unterschiedlicher Art, die intern zwischen den unterschiedlichen Arten transformiert und schließlich für eine spätere Verwendung gespeichert werden.*“ [Pop+21] nutzt den EH-Ansatz als dezentralen Baustein zum Bereitstellen von Netzdienstleistungen. Der EH ist ein Ansatz zum Glätten und Ausgleichen lokaler Erzeugung und Nachfrage, wie in [Moh+17] angegeben, insbesondere wenn **R**enewable **E**nergy **S**ources (RESs) integriert sind.

Wie Abb. 7.3 veranschaulicht, ist das implementierte und untersuchte EH-Modell für eine Vielzahl von Anwendungsfällen ausgelegt. Dabei koppelt das EH-Modell den Strom- und den Gassektor bidirektional. Die Elektrolyse mit gekoppelter Methanisierung wandelt Strom in Methan um. Das **B**lock**h**eiz**k**raft**w**erk (BHKW) (Combined Heat and Power-CHP) wandelt Methan in Strom um. Ein Gasspeicher speichert chemisch Energie in Form von Methan, und eine Batterie in Form von Elektrizität. Der EH ist über das Methannetz weiterhin an eine nicht-steuerbare Biogasanlage (**B**io **G**as **P**lant, BGP), sowie an ein Methannetz, und lokale Verbraucherinformationen aus der Industrie gekoppelt. Auf der elektrischen Seite ist der EH an einen 20/110kV elektrischen Anschlusspunkt gekoppelt. Ähnlich zum Methannetz bietet auch das Stromnetz erneuerbare Erzeuger in Form von Windkraft- und PV-Anlagen, sowie lokale Verbraucherinformationen aus der Industrie.

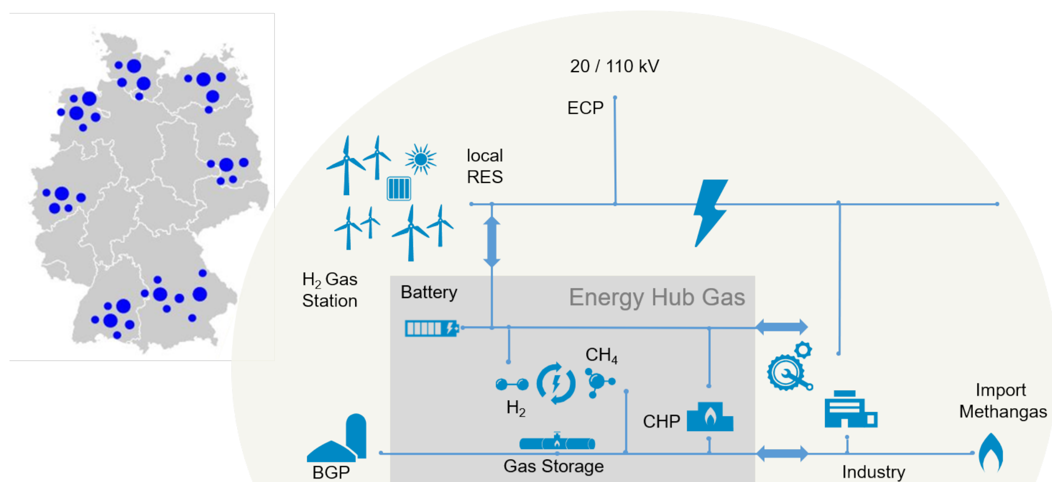

Abb. 7.3.: Übersicht des EH-Systems [Pop+21]

Der Netzbetreiber sendet Fahrpläne für den EH als eine Einheit an das **E**nergy **M**anagement **S**ystem (EMS) basierend auf der lokalen RES-Erzeugung und dem Bedarf. Anschließend muss das EMS den erhaltenen Fahrplan an seinem elektrischen Anschlusspunkt (ECP) erfüllen, indem es die enthaltenen Komponenten aggregiert. Das EMS erhält von einem Marktsimulator weiterhin Preisvorhersagen für die verschiedenen Energieträger. In Kombination kann das EMS den EH dann sowohl netzdienlich als auch kostenoptimiert steuern.

Das Bereitstellen eines Fahrplans für jede Anlage formuliert ein komplexes Optimierungsproblem. Das EMS steuert sechs Anlagen (Batterie, Gasspeicher, BHKW, Elektrolyse, Methanisierung und Biogas) und erhält dabei ständig (Meta-)Daten über deren aktuellen Zustand. Die Fahrplanoptimierung folgt zwei Zielfunktionen: (1) Folgen des Zielwerts des Netzbetreibers am Stromanschlusspunkt, (2) Kosteneffizienz unter Berücksichtigung von CO_2-Emissionen und Betriebskosten. Die Zielfunktionen erfordern zusätzliche Preisinformationen. Das EMS erhält Preisinformationen als (vorhergesagte) Zeitreihendaten aus dem Marktsimulator (historische Zeitreihendaten). Der Anwendungsfall berücksichtigt die Marktpreisinformationen für 2021. Wetterdaten für Karlsruhe, Deutschland, und die Spitzenleistung der installierten erneuerbaren Energien vervollständigen die Szenarioinformationen. Das betrachtete RES akkumuliert bis zu 4MW. Die Netzbetreibersimulation liefert die resultierenden Fahrpläne für den gesamten EH, indem vollständige Informationen aus dem Szenario als Prognose berücksichtigt werden. Das EH-Modell ist auch auf Industriegebiete sowie kleine Städte an den Schnittstellen zum Übertragungsnetz anwendbar. Es besteht aus verschiedenen Umwandlungs- und Speichereinheiten zur flexiblen Kopplung des Gas-, Strom- und Wärmesektors, um das Stromnetz mit Ausgleichsleistungen zu unterstützen.

Implementierung

Zur Evaluation der Praxistauglichkeit des Umsetzungsansatzes wird das EH-Modell als Kombination aus **F**unctional **M**ock-up **U**nits (FMUs) und Python-Modellen implementiert.

Sowohl die Verknüpfung der einzelnen Modelle, als auch das Bereitstellen korrekter Szenariodaten wird unter Verwendung von PROOF umgesetzt. Beispielsweise werden für das Durchführen einer schrittweisen Simulation für das gesamte Szenario zusätzliche funktionale Komponenten von PROOF benötigt (s. Abb. 7.4). Dabei fungiert ein Primary-Prozess als Primary-Scheduler, der die schrittweise Ausführung der Secondary-Simulationen vorantreibt, und ein Merge-Service als Datensynchronisierungskomponente, die auf die Ausgänge der Secondary-Simulatoren wartet und

dem Scheduler mitteilt, dass ein Schritt beendet ist. Der Scheduler verteilt dann Szenariodaten und andere Eingaben an die Secondary-Simulatoren und fordert sie auf, einen Schritt auszuführen.

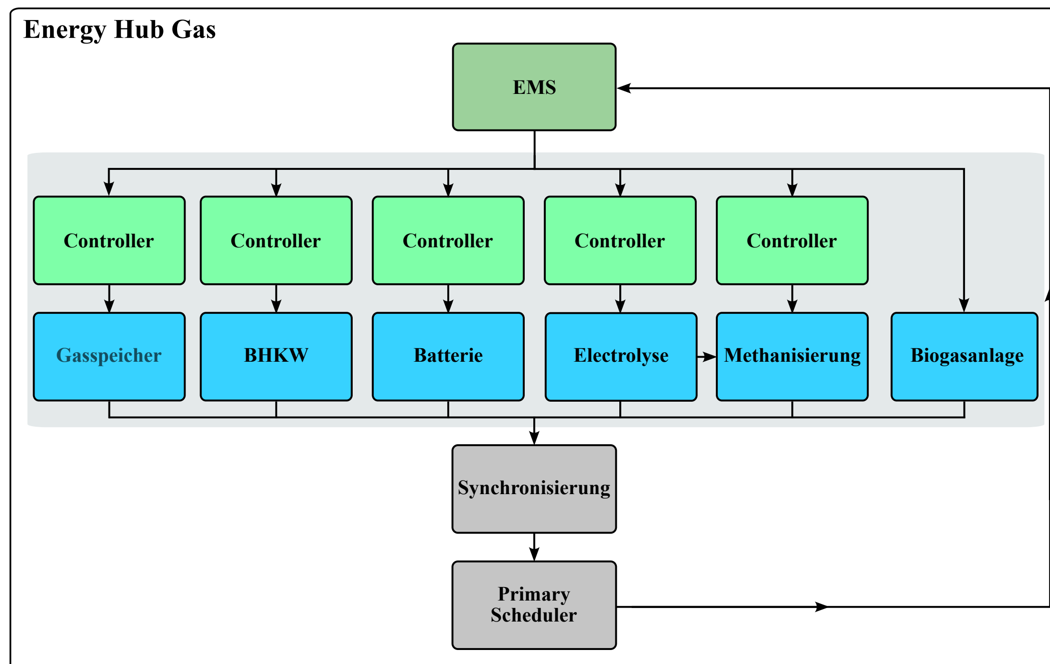

Abb. 7.4.: Aufbau des *EH*

Abb. A.9 in Anhang A.9.3 demonstriert die Verwendung von PROOF zur Implementierung des EH-Systemmodells. Alle Systemmodelle werden als Co-Simulation unter Verwendung eines schrittweisen Primary-Secondary-Ansatzes durch PROOF implementiert. Die Art der verschiedenen Komponenten wird im grafischen Modell durch verschiedenfarbige Symbole der Kästchen unterschieden:

- Das dunkelgrüne Symbol zeigt das EMS.
- Die grünen Symbole stellen Controller-Komponenten der physikalischen Modelle dar.
- Die blauen Symbole markieren physikalische Komponentenmodelle.
- Die grauen Symbole koordinieren die schrittweise Simulation.

Die Scheduler-Komponente in der Mitte des Workflows initiiert Simulationsschritte, indem sie Szenario- und andere Eingabedaten für diesen Schritt an alle beteiligten Secondary-Simulatoren liefert und dann wartet, bis die Synchronisationskomponente die Ausgabe von jedem physikalischen Modell gesammelt hat. In jedem Schritt bestimmt die EMS-Komponente zunächst, ob sie neue Einsatzpläne an die Controller zur dynamischen Einstellung der technischen Anlagen senden soll. Dann führen die physikalischen Simulatoren ihre Berechnungen gemäß der gegebenen

Szenarioeingabe (z.B. Wetterdaten), den definierten Steuerungseinstellungen und anderen verfügbaren Eingabedaten durch und senden ihre Ergebnisse an die Synchronisationskomponente. Danach gibt diese Synchronisationskomponente dem Scheduler eine Rückmeldung, dass alle Berechnungen durchgeführt werden. Dann löst der Scheduler den nächsten Schritt aus, bis alle Schritte ausgeführt werden. Darunter sammelt PROOF alle erforderlichen Daten für die spätere Analyse des Co-Simulationslaufs.

Bewertung

Der Anwendbarkeit von PROOF wurde in Meetings diskutiert und bewertet. Ähnlich wie der vorherige Anwendungsfall macht die Komplexität des EH-Modells eine manuelle Ausführung des Modells praktisch unmöglich, insbesondere dann, wenn der Workflow häufige Eingabedaten verarbeiten muss. Ein weiteres Argument gegen die manuelle Ausführung des Workflows ist die Tatsache, dass die Ausgabedaten aller physikalischen Simulatoren synchronisiert und zusammengefügt werden müssen, bevor sie vom nächsten Schritt verwendet werden können. Außerdem ist es ziemlich mühsam und fehleranfällig, Datenübertragung zwischen den dreizehn Komponenten des EH-Modells direkt ohne PROOF durchzuführen. Um die Ausführungseffizienz zu verbessern, ist es daher notwendig, die Ausführung des gesamten Modells durch PROOF vollständig zu automatisieren.

Als Bericht haben alle Teilnehmer ein Konferenzpaper [Pop+21] zur Beschreibung des EH-Modells und seiner Implementierung in PROOF veröffentlicht. Kapitel *RESULTS AND DISCUSSION* des Konferenzpapers beschrieb, dass die Anwendbarkeit von PROOF durch die Simulation des EH-Modells validiert wurde.

7.2.4 Weitere Anwendungsfälle

Neben den drei vorgestellten Anwendungsfällen wurden noch die folgenden Anwendungsfälle in PROOF umgesetzt und jeweils als Paper veröffentlicht.

- Prognosen für Energiesysteme der Zukunft: Methoden und Tools [MGH20]
- A generic distributed microservices and container based framework for metaheuristic optimization [Kha+18]
- Power System Simulation [Çak+19]

Darüber hinaus wurde PROOF für weitere verschiedene Forschungsprojekte (z.B. ES2050 [HEL22], Energylab 2.0 [KIT22b], IT-Komponenten für die Energiewende

[KIT22c], Sektorenkopplung [KIT22a] usw.) eingesetzt und als Poster auf verschiedenen Konferenzen (z.B. [PLD20; Bra+16; LD16; Red+20; LDH17] usw.) vorgestellt. Weitere Details zu den Einsätzen und Postern werden innerhalb dieser vorliegenden Arbeit nicht dargestellt.

Eine Zusammenfassung über alle PROOF-Prozesse, die für verschiedene Anwendungsfälle von PROOF bereitgestellt wurden, wird im nachfolgenden Abschnitt 7.4 gegeben.

7.2.5 Zusammenfassung der PROOF-Prozesse in der Prozess-Bibliothek

Basierend auf den Anforderungen von verschiedenen Anwendungsfällen wurden viele Standard-Prozesse und -Services in PROOF bereitgestellt und zur Wiederverwendbarkeit in der Prozess-Bibliothek abgespeichert. In Tab. 7.1 werden alle PROOF-Prozesse zusammengefasst. Neben der Beschreibung ihrer Funktionalitäten sind die Anwendungsfälle erfasst, in denen sie eingesetzt werden.

Tab. 7.1.: Standard-Prozesse und -Services in der Prozess-Bibliothek von PROOF

Kategorie	Funktionalität	Anwendungsfall
MATLAB-Prozess	Ausführen von MATLAB-Modellen	[Gon+21]
Julia-Prozess	Ausführen von Julia-Modellen	[Gon+21]
Python-Prozess	Ausführen von Python-Modellen und -Programmen	[Gon+21; LDH17; Liu+18; Liu+19]
Java-Prozess	Ausführen von Java-Applikationen	[Kha+18]
C und C++-Prozess	Ausführen von C- und C++-Programmen	[Kha+18]
eASiMOV-Prozess	Verarbeitung von Daten aus der Datenbank *eASiMOV*[1]	[Gon+21]
FMU-Prozess	Operation von FMU-Modellen	[Pop+21; PLD20]

Auf der nächsten Seite fortgesetzt

[1] iai-easimov.iai.kit.edu

Tab. 7.1.: Standard-Prozesse und -Services in der Prozess-Bibliothek

Kategorie	Funktionalität	Anwendungsfall
Excel-Prozess	Verarbeitung von Daten in Excel-Dateien (z.B. lesen, schreiben, durchsuchen usw.)	Zusammenarbeit mit dem Institut für Technische Chemie
PDF-Prozess	Bearbeitung von PDF-Dateien (z.B. lesen, schreiben, durchsuchen usw.)	Zusammenarbeit mit dem Institut für Technische Chemie
CSV-Prozess	Operation von Daten in CSV-Dateien (z.B. lesen, schreiben, durchsuchen usw.)	[LDH17; Liu+18; Liu+19]
MQTT-Schnittstelle	Kommunikation mit externen Applikationen	Zusammenarbeit mit EBI (Engler-Bunte-Institut)
STANET-Schnittstelle	Einbindung von STANET (Wärme- und Gasnetz Campus Nord)	Zusammenarbeit mit EBI (Engler-Bunte-Institut)
Merge-Prozess	Verbinden mehrerer eingehender Datenflüsse zu einem Ausgabedatenfluss durch vordefinierte Regeln	[Gon+21; Pop+21]
Zeitreihendaten-Prozess	Prozessoren zum Zugriff auf die Datenplattform (z.B. Lesen und Schreiben von Daten in der Zeitreihendatenbank)	[Pop+21; Gon+21]
Wrapper-Service	Steuerung von Simulatoren, Modellen und Applikationen	[Pop+21; Gon+21; LDH17; Liu+18; Liu+19; LD16; Bra+16; PLD20]

Auf der nächsten Seite fortgesetzt

Tab. 7.1.: Standard-Prozesse und -Services in der Prozess-Bibliothek

Kategorie	Funktionalität	Anwendungsfall
Adapter-Service	Datei-, TCP- und HTTP-Adapter als Schnittstelle zur Kommunikation zwischen Redis und Applikationen	Alle Anwendungsfälle
Optimierungsservice	Steuerung von Gleam	[Kha+18]
MatPower-Service	Steuerung von MatPower-Simulationen	[Çak+19]
IEC61850-Service	Datenerfassung bei Messgeräten	IEC 61850
Forecasting-Service	Lastprognose von z.B. Gebäuden am Campus Nord	[KIT22b]
pyWATTS-Service	Ausführen von pyWATTS-Modellen	pyWATTS[2] [Hei+21]
Primary-Prozess	Realisierung von Primary-Secondary-Koordination	[Pop+21]

7.2.6 Zusammenfassung der Anwendungsfälle

Die Ergebnisse der Studien der Anwendungsfälle fassen zusammen, dass die Workflows der Anwendungsfälle wie im Konzept erarbeitet funktioniert. Hiermit kann die Funktionalität von PROOF und der einzelnen Komponenten sowie Services belegt werden. Die zu validierenden Beiträge dieser Arbeit wurden mittels Erfüllung unterschiedlicher Anforderungen der Anwendungsfälle überprüft. Die Forschenden und Ansprechpersonen der Anwendungsfälle sind überzeugt, dass sich wissenschaftliche Workflows durch PROOF schneller integrieren, vollautomatisiert und performanter ausführen lassen, damit ihre Arbeit effizienter gestaltet werden kann.

Zur Evaluierung der Ausführungseffizienz von Workflows im Framework wird ein Benchmarking zur Messung der Speichernutzung, der Ausführungszeit im Vergleich zur direkten Ausführung und des Overheads für parallele Aufgaben in PROOF durchgeführt und im nachfolgenden Abschnitt 7.3 vorgestellt.

[2]https://pywatts.readthedocs.io/en/latest/

7.3 Benchmarking

Um das Verhalten und die Leistung des vorgestellten Frameworks in Bezug auf die Parallelverarbeitung zu bewerten, wurde ein allgemeiner Anwendungsfall innerhalb der Energiesystemanalyse, der von der Parallelisierung profitiert, verwendet.

7.3.1 Use-Case-Workflow

Der entsprechende Workflow des Anwendungsfalls wurde mit einer unterschiedlichen Anzahl paralleler Aufgabeninstanzen ausgeführt. Der Workflow führte Berechnungen zur optimalen Dimensionierung eines überregionalen Energiesystems unter Berücksichtigung konventioneller und erneuerbarer Energiequellen sowie verschiedener Speicher-, Umwandlungs- und Transporttechnologien. Die Ausführung des Workflows kann parallelisiert werden, indem z.B. ein großes geografisches Gebiet in separate Gebiete aufgeteilt wird, in denen Berechnungen parallel durchgeführt werden können. Basierend auf räumlich und zeitlich aufgelösten Wetterdaten und detaillierten Markt- und technoökonomischen Informationen wurden die installierten Gesamtleistungen, Kosten der Energieträger, erzeugte Strommengen, Gesamtenergiebedarf und Gesamtsystemkosten ermittelt.

Der zum Testen des Frameworks verwendete Workflow umfasste vier Python-basierte Energiesystemmodelle. Das Energiebedarfsmodell (Demand Model) schätzte den Gesamtenergiebedarf für eine bestimmte Region und für ein Jahr auf der Grundlage von Bevölkerungsprognosen und Statistiken zum Energieverbrauch im Mobilitätssektor. Das Landeignungsmodell (Land Eligibility) verarbeitete soziotechnische Parameter, um geeignete Flächen für die Erzeugung erneuerbarer Energien zu bestimmen. Die Turbinenplatzierungs- und Turbinensimulationsmodelle (Turbine Placement und Turbine Simulation) wählten spezifische Standorte für Windturbinen aus und simulierten entsprechende zeitabhängige Produktionsraten. Darüber hinaus umfasste der Workflow zwei ausführbare Dateien, Konvertierung 1 und Konvertierung 2 (Conversion 1 und Conversion 2), die die Daten als geeignetes Eingabeformat für ein nachfolgendes Optimierungsmodell anordneten.

Wie in Abb. 7.5 dargestellt, wurde der Workflow in zwei unabhängige Sub-Workflows aufgeteilt, die für jede betrachtete Region ausgeführt werden mussten. Dabei wurde ListenHTTP als HTTP-Server für Eingaben verwendet. Der Koordinationsdienst synchronisierte die Ausgaben von Konvertierung 1 und Konvertierung 2. Für eine optimale Leistung sollte der gesamte Vorgang parallel berechnet werden.

Weitere detaillierte Informationen zu den Modellen und deren Einbindung in eine anspruchsvollere Modellkette finden sich in [Wel+18; Rob15; Rob+17; RRS18; Ryb+19; For20].

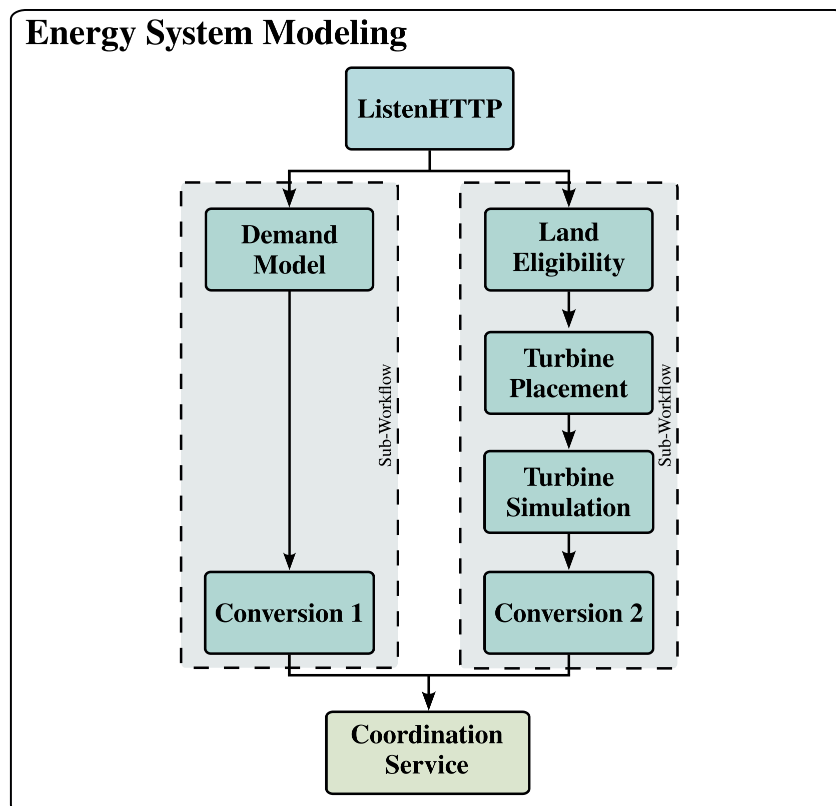

Abb. 7.5.: Use-Case-Workflow zur Gestaltung eines überregionalen Energiesystems

Um die Leistung des Frameworks während der Ausführung des Workflows zu ermitteln, wurden während der Ausführung des Workflows mehrere Benchmarks zur Messung der Speichernutzung, der Ausführungszeit im Vergleich zur direkten Ausführung ohne PROOF und des Overheads für parallele Aufgaben durchgeführt. In den nachfolgenden Abschnitten werden die Benchmarks detailliert diskutiert.

7.3.2 Speichernutzung

Die Speichernutzung mit und ohne PROOF, die für denselben Workflow erforderlich sind, werden in Abb. 7.6 und 7.7 gezeigt.

Im Vergleich zur direkten sequenziellen Ausführung jedes Modells ohne PROOF, wurden sechs Container, die ein Modell jeweils umfasst, gleichzeitig im Workflow-Setup verwendet und in PROOF ausgeführt. Dies benötigte mehr Gesamtspeicher

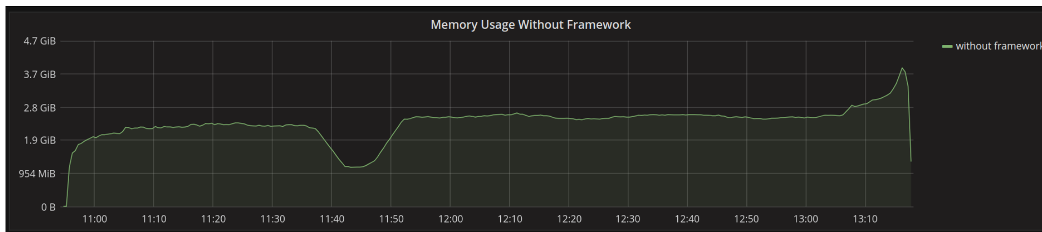

Abb. 7.6.: Speichernutzung ohne PROOF

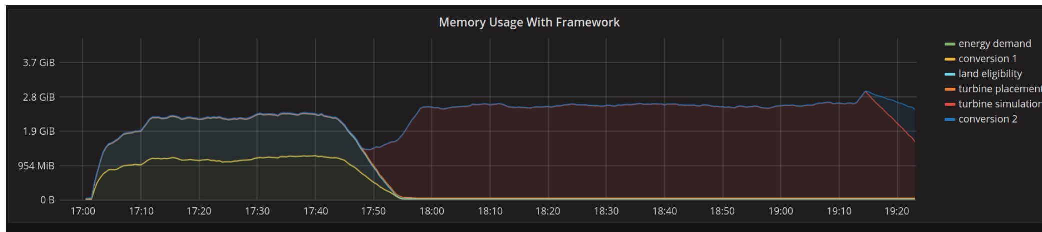

Abb. 7.7.: Speichernutzung mit PROOF

als eine direkte sequentielle Ausführung. Die Speichernutzung wurde für die direkte Ausführung und für jeden Container innerhalb des Frameworks überwacht. Beide resultierenden Kurvenverläufe sehen jedoch ähnlich aus, insbesondere nach den ersten 45 Minuten zum Ausführen der oben dargestellten Modelle. Nachdem die Modelle Landeignung, Turbinenplatzierung, Energiebedarf und Konvertierung 1 fertig gerechnet wurden, wurde fast 2.8 GB Gesamtspeicher verwendet, um die Turbinensimulation für die beiden Situationen auszuführen. Das bedeutet, dass die Ausführung in der Containerumgebung bei der Ausführung der Aufgabe der Turbinensimulation fast keinen zusätzlichen Speicher beanspruchte. Darüber hinaus war in beiden Fällen auch die Ausführungszeit einer Aufgabeninstanz in einem einzelnen Container gleich schnell wie im Standalone-Fall ohne PROOF. Wenn also die vom Container bereitgestellte CPU-Leistung mit der für die eigenständige Ausführung derselben ausführbaren Datei bereitgestellten CPU-Leistung vergleichbar war, war die Verarbeitungszeit fast gleich. Dies bedeutet, dass die Nutzung der Container-Technologie zur Ausführung der Modelle im Vergleich zur direkten Ausführung der Modelle fast keinen Overhead der CPU-Nutzung hinzufügte. Daher benötigte PROOF fast den gleichen Speicherbedarf zum Ausführen einer ausführbaren Datei wie für eine direkte Ausführung, und die Leistung zum Ausführen einzelner Modellinstanzen war in beiden Fällen nahezu gleich. Die Ausführungszeit hing auch von der Leistung des Rechenknotens selbst ab, die für diesen Vergleich als ähnlich angenommen werden konnte.

7.3.3 Ausführungszeit

Tab. 7.2 listet die direkte Ausführungszeit ohne PROOF und die Ausführungszeit mit PROOF für jedes Modell sowie die Gesamtausführungszeit für alle Modelle auf. Aus den Werten ist ersichtlich, dass jedes Modell in beiden Laufzeitumgebungen grundsätzlich die gleiche Ausführungszeit benötigt. Die direkte Ausführung ohne Framework für alle Modelle dauerte etwa $239\,\mathrm{s}$, während die Ausführung des Modells mit dem Framework etwa $240\,\mathrm{s}$ dauerte. Dank der Container-basierten Virtualisierungstechnologie auf der zugrunde liegenden Cluster-Computing-Umgebung und der Microservices-basierten Architektur des Frameworks war der Unterschied in der Ausführungszeit der Modelle nahezu vernachlässigbar. Daher kann geschlussfolgert werden, dass die vom Framework für einzelne Instanzen bereitgestellte Modellausführungsumgebung und -leistung im Wesentlichen mit den direkten identisch sind.

Tab. 7.2.: Vergleich der Ausführungszeit mit und ohne PROOF

Modell	Ausführungszeit (s) ohne PROOF	Ausführungszeit (s) mit PROOF
land eligibility	30,895	31,614
turbine placement	5,245	5,147
turbine simulation	180,357	178,956
conversion 2	22,406	23,874
energy demand	30,988	30,615
conversion 1	0,504	0,517
all models	238,903	239,591

Um den durch das Framework hinzugefügten Kommunikations- und Workflow-Verarbeitungsoverhead zu analysieren, wurden 20 Tests mit unterschiedlichen Parametern durchgeführt. In Abb. 7.8 geben die horizontale Achse und die vertikale Achse jeweils die Gesamtausführungszeit t jedes Tests und das Verhältnis des Overheads o zur Gesamtausführungszeit an, die durch jeden Test erzeugt wird, nämlich o/t. Wie der Kurve in Abb. 7.8 entnommen werden kann, wurde der Anteil des Overheads an der gesamten Prozessausführungszeit sukzessive reduziert. Im Vergleich zur gesamten Ausführungszeit kann dieser geringe Overhead durch die Performance-Vorteile des parallelen Rechnens aufgewogen werden, auf die im nächsten Abschnitt eingegangen wird.

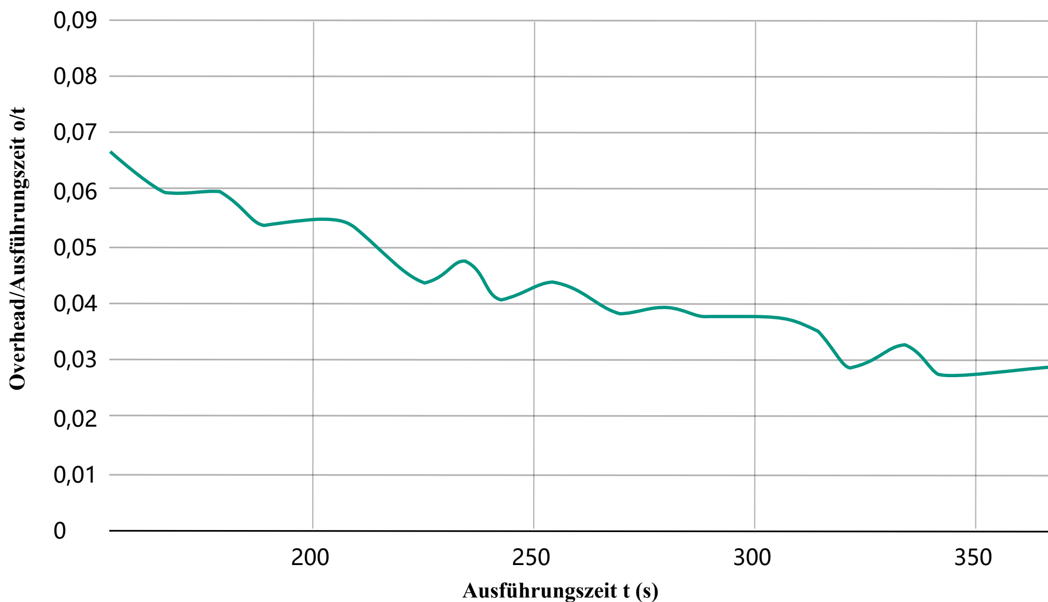

Abb. 7.8.: Verhältnis von Overhead zu der gesamten Ausführungszeit

7.3.4 Parallelisierung

Um die Skalierbarkeit und Daten-Parallelismus (s. 6.1) von PROOF zu untersuchen, wurden 10 verschiedene parallele Tests durchgeführt. In Abb. 7.9 stellen die horizontale und vertikale Achse die Anzahl paralleler Tasks und die Ausführungszeit für jeden Test dar. Bei jedem Test entsprach die Menge der ausgeführten Tasks der Menge der an den Workflow gesendeten Eingaben. Das erwartete Ergebnis war, dass die Ausführungszeit bei allen getesteten Anzahlen von parallelen Tasks konstant ist, da jeder Task eine Eingabe verarbeitet. Abb. 7.9 veranschaulicht den fast gleichen Overhead für jeden Task. Die Gesamtausführungszeit stieg leicht an, als die Anzahl der parallelen Tasks von eins auf zehn stieg. Die Ausführungszeit hängt nicht von der Anzahl der Tasks ab. Bezogen auf die Gesamtausführungszeit (156s) ist der geringe Anstieg (ca. 8s) durch die Parallelität akzeptabel und nahezu vernachlässigbar. Darüber hinaus kann die Parallelität von nur zwei Tasks bereits den im vorherigen Abschnitt beschriebenen geringen Overhead überwinden, da die Ausführung von zwei Tasks nacheinander etwa 310s dauerte, während die Zeit für die parallele Ausführung von zwei Tasks mit PROOF immer noch etwa 156s betrug.

Die durch PROOF bereitgestellte Parallelität kann auf viele Anwendungsfälle angewendet werden, um Laufzeit erheblich zu sparen und die Betriebseffizienz zu verbessern, insbesondere bei komplexen Aufgaben wie der iterativen Netzoptimierung. Aus den Ergebnissen der Tests lässt sich daher auf eine hohe Skalierbarkeit und erweiterte Flexibilität von PROOF schließen.

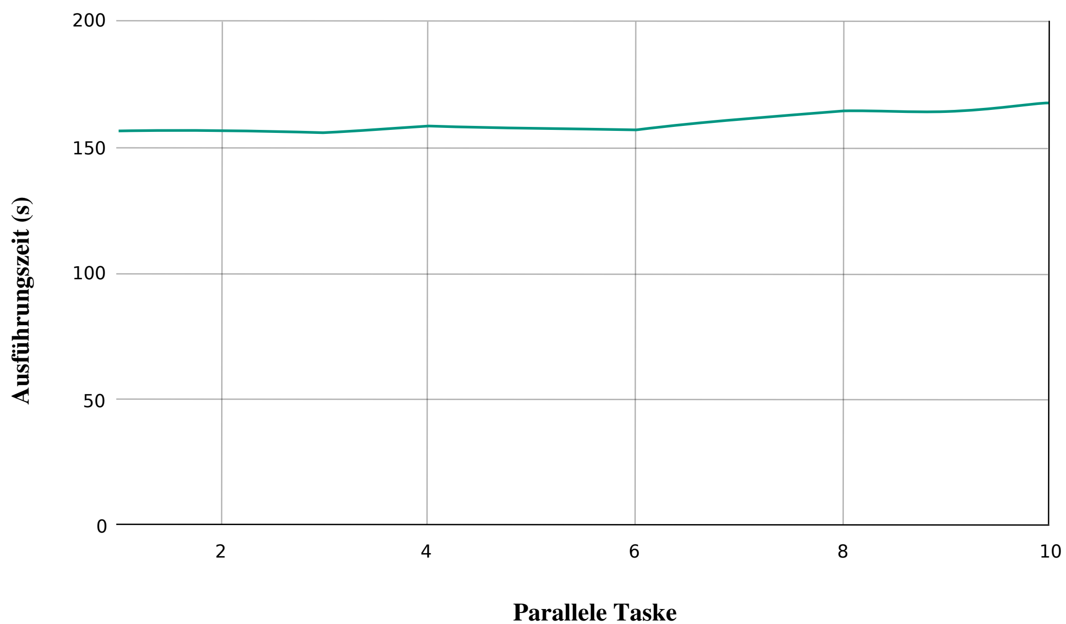

Abb. 7.9.: Ausführungszeit für mehrere parallele Tasks

Für die weitere Evaluierung werden im nachfolgenden Abschnitt Qualitätsmerkmale 7.4 herausgearbeitet und auf das Konzept angewandt.

7.4 Analyse von Qualitätsmerkmalen

In diesem Abschnitt wird der Prototyp bzw. das Konzept von PROOF analysiert, indem verschiedene Qualitätsmerkmale als Grundlage genutzt werden. Die Qualitätsmerkmale werden in Anlehnung an die ISO/IEC 25010:2011 -Norm [13] gewählt. Die Norm wird weiter in verschiedene Unterqualitätsmerkmale unterteilt, die eine optimale Grundlage für eine detaillierte Analyse bieten. Im Folgenden wird jedes Qualitätsmerkmal separat diskutiert, um das PROOF-Konzept zu bewerten.

7.4.1 Funktionalität (Functional Suitablity)

Funktionalität ist die Fähigkeit der Softwarearchitektur, die erforderlichen Funktionen bereitzustellen. Wie in der Norm [13] definiert, verfolgt der Aspekt Funktionalität mehrere wichtige Ziele. Zum einen ist es das Prüfen auf Vollständigkeit hinsichtlich der Softwarefunktionen. Im vorliegenden Fall hängt die Funktionalität hauptsächlich von der Erfüllung der Anforderungen ab, welche bereits in Abschnitt

3.1.2 zusammengefasst wurden. Die Erfüllung dieser Anforderungen wird in Abschnitt 7.5 nachfolgend detailliert beschrieben, weshalb die Anforderungen hier nicht weiter ausgeführt werden.

Zwei weitere Aspekte der Funktionalität sind die Korrektheit und die Angemessenheit. Unter der Korrektheit wird das Liefern der richtigen oder vereinbarten Ergebnisse verstanden. Die PROOF-Bausteine selbst stellen eine Black Box dar, weshalb die Funktionalität der Bausteine als das richtige Setzen der Ein- und Ausgabewerte definiert wird. Die Angemessenheit bezieht sich darauf, wie gut Funktionen bestimmte Aufgaben und Ziele erfüllen können. Wie in Abschnitt 7.2 beschrieben, wurden verschiedene Anwendungsfälle in PROOF implementiert. Die Ein- und Ausgabedaten für jeden Anwendungsfall wurden von den Ansprechpersonen analysiert und überprüft, dass die Simulationen aller Anwendungsfälle fehlerfrei und richtig ausgeführt wurden. Dies dient zur Einhaltung der Korrektheit. PROOF stellt die grundlegenden Funktionalitäten, wie z.B. automatisierte, koordinierte und parallelisierte Ausführung von Workflows, und verschiedene Hilfsbausteine für generische Funktionalitäten, wie z.B. der PROOF-Wrapper, der Merge-Service, der Primary-Prozess, bereit, um bestimmte Anforderungen und Ziele der Anwendungsfälle zu erfüllen. Somit wird die Angemessenheit gewährleistet.

7.4.2 Kompatibilität (Compatibility)

Dieses Qualitätsmerkmal ist eng mit dem Einsatz der Dienste und der Infrastruktur verbunden. Unter dem Merkmal wird verstanden, verschiedene Infrastrukturen und die Interoperabilität der Bereitstellung von Softwarekomponenten für diese Infrastrukturen zu unterstützen. Es ist ein häufiger Anwendungsfall, dass ein System in einer Infrastruktur sowohl zu Testzwecken als auch für die Produktion bereitgestellt wird. Daher ist es wichtig, dass das System aus den verschiedenen Diensten und Komponenten auf beiden Infrastrukturen einsetzbar ist.

Der Prototyp von PROOF erfüllt diesen Bedarf. Anhand der Wahl, auf Containervirtualisierung und Microservices-Architektur zu setzen, werden alle PROOF-Komponenten, -Services und -Prozesse als unterschiedliche Container bereitgestellt. Wissenschaftliche Applikationen beruhen auf diversen Programmiersprachen, -Umgebungen bzw. Bibliotheken. Durch die Verwendung der Containervirtualisierung können die Applikationen jeweils in einem Image gekapselt werden. Dies ermöglicht, die wissenschaftlichen Applikationen unabhängig von ihrer Programmiersprache einzubinden und als PROOF-Prozesse in PROOF zu integrieren, ohne dabei ihre

Funktionalität zu beeinträchtigen. Mit der Containervirtualisierung als Abstraktionsschicht ist PROOF mit jeder Infrastruktur kompatibel, die Container-Virtualisierung unterstützt. Dies ermöglicht eine hohe Kompatibilität der PROOF-Architektur mit unterschiedlichen Infrastrukturen.

Darüber hinaus ist eine zweite Form der Kompatibilität zu berücksichtigen, wie Anwendungen von Drittanbietern mit PROOF kommunizieren können. Dies wird durch den Mqtt-Service und die REST-APIs begünstigt, die PROOF als Schnittstelle mit externen Anwendungen/Modellen/Anlagen bereitstellen (s. 5.4). Eine Anwendung muss lediglich das HTTP(S)-Protokoll oder Mqtt-Protokolle unterstützen. Da es für beide Protokolle zahlreiche Bibliotheken für viele Programmiersprachen gibt, sind Anwendungen mit den Kommunikationsschnittstellen von PROOF kompatibel.

7.4.3 Portabilität (Portability)

Portabilität ist die Eigenschaft eines Computerprogramms, in anderen Betriebssystemen betrieben werden zu können, ohne dass dies Änderungen oder viel Programmieraufwand erfordert. Dies hängt hauptsächlich vom Einsatz der Software und ihrer Kompatibilität mit verschiedenen Infrastrukturen ab. Die Portabilität hängt eng mit dem ersten Aspekt zusammen, der für das Kompatibilitätsmerkmal zuvor diskutiert wurde. Bereits dieser Aspekt zeigte, dass die Container-Virtualisierung jeder Komponente von PROOF zu einer hohen Kompatibilität mit unterschiedlichen Infrastrukturen und damit auch zu einer hohen Portierbarkeit auf unterschiedliche Umgebungen führt. Durch die Portabilität lassen sich die Container auf einem Computing-Cluster ausführen und somit von höherer Rechenleistung oder mehr Speicher profitieren.

7.4.4 Benutzbarkeit (Usability)

PROOF bietet eine Unterstützung zum Aufbau von Workflows für Simulationen und Co-Simulationen. Dabei handelt es sich um zwei Aspekte der Benutzbarkeit, nämlich der Backend- und Frontend-Benutzbarkeit.

Zur Backend-Benutzbarkeit bietet PROOF verschiedene generische Framework-Prozesse als Hilfsbausteine, um unterschiedliche Applikationen zu integrieren. Dabei werden eine einheitliche Struktur des E/A-Adapters, PROOF-Wrappers und Datenformats (JSON) verwendet. Dies schafft eine hohe Synergie zwischen den

PROOF-Prozessen und somit beinhaltet eine flache Lernkurve zwischen Benutzern und Prozessen.

Der zweite Aspekt, der zu einer besseren Frontend-Benutzbarkeit führt, ist das Integrieren der grafischen Webbenutzeroberfläche. Der Aufbau, die Konfiguration und die Ausführung von Workflows sind visuell realisierbar auf der Weboberfläche. Für die Anwendung von PROOF werden keinerlei Programmierkenntnisse benötigt. Dies beeinflusst die Benutzbarkeit positiv.

7.4.5 Verlässlichkeit (Reliability)

Unter Verlässlichkeit sind vier Unterkategorien zu verstehen: der Reifegrad einer Software, deren Skalierbarkeit, die Fehlertoleranz und Wiederherstellbarkeit.

Die Ausgereiftheit verlangt, dass der Prototyp die Anforderungen zuverlässig im Normalbetrieb erfüllt. Der PROOF-Prototyp, der in der vorliegenden Arbeit vorgestellt wurde, erfüllt die definierten Anforderungen vollständig. Eine detaillierte Beschreibung der Anforderungsabdeckung befindet sich in Abschnitt 7.5.

PROOF wird auf einem Kubernetes-Computing-Cluster eingerichtet, um genügend Ressourcen zur Bewältigung der spezifischen Belastung zu haben. Mithilfe des Kubernetes-Clusters ermöglicht PROOF die horizontale Skalierung von Microservices, indem mehrere Instanzen desselben Services parallel ausgeführt werden. Die Services werden auf die verschiedenen Cluster-Knoten verteilt, um die Leistung des Clusters optimal zu nutzen. Außerdem verfügt PROOF über integrierte Lastausgleichsstrategien des Kubernetes-Clusters, die Anforderungen an die verschiedenen Instanzen eines Dienstes weiterleiten. Um die Skalierbarkeit zu untersuchen, wurden 10 verschiedene parallele Tests durchgeführt (s. Abschnitt 7.3.4). In Abbildung 7.9 zeigte die Ergebnisse des Benchmarkings, dass die Zeit für die parallele Ausführung von mehreren Tasks in PROOF immer noch etwa 156s betrug und die Zeit für Overheads nahezu vernachlässigbar war. Daher kann aus den Testergebnissen auf eine hohe Skalierbarkeit und erweiterte Flexibilität des Frameworks geschlossen werden.

Die beiden Unterkategorien Fehlertoleranz und Wiederherstellbarkeit lassen sich aufgrund der Realisierung im Prototyp bzw. Konzept zusammenfassen. Die Fehlertoleranz fordert, dass der Prototyp funktioniert, trotz Vorhandensein von Hardware- oder Softwarefehlern. Die Wiederherstellbarkeit beschreibt den Grad, zu dem ein Produkt oder System im Falle einer Unterbrechung oder eines Ausfalls die betroffenen Daten und den gewünschten Zustand des Systems wiederherstellen kann. Beide

Aspekte werden durch die Container-Orchestrierung (Kubernetes) unterstützt. Bei einem Absturz oder Fehler eines Containers ermöglicht Kubernetes einen Neustart durchzuführen, bis der Container wieder funktionsfähig ist. Somit kann beim Ausfall eines oder mehrerer Container der gewünschte Zustand wiederhergestellt werden. Die Daten in Redis wären nach einem Absturz zwar weg, jedoch werden sie bei einem Neustart des Workflows wieder erzeugt. Durch die separate Ausführung der Container auf dem Computing-Cluster profitiert die Fehlertoleranz ebenfalls. Ein Fehlerfall innerhalb eines Containers hat keinen Einfluss auf die anderen Container. Außerdem kann ein Container falsche Daten liefern, die den nächsten Container zum Absturz bringen können. Für diesen Fall kann PROOF nicht für Fehler „verantwortlich" gemacht werden, wenn die Modellierer (Programmierer) schwere Fehler machen.

7.4.6 Wartbarkeit (Maintainability)

Laut [13] beschreibt Wartbarkeit die Fähigkeit, die Software zu modifizieren und den Aufwand, welcher erforderlich ist, um die Änderung an der Software vorzunehmen. Änderungen sind beispielsweise Korrekturen, Verbesserungen, Anpassungen an die Umgebung, neue Anforderungen oder eine neue funktionale Spezifikation. Um eine Software nachhaltig zu modifizieren, ist es für eine Software-Architektur sehr wichtig, bestimmte Qualitätsmerkmale sicherzustellen.

Eines dieser Merkmale bezieht sich auf die Modularität, d.h. dass verschiedene Services und Komponenten unabhängig voneinander sind. Dies ist ein sehr entscheidender Aspekt, da ein stark gekoppeltes Netzwerk von Services und Komponenten auf lange Sicht schwer wartbar ist. Die PROOF-Architektur basiert auf einer Microservices-Architektur. Wie Tab. 2.1 liefert, weist die Microservice-basierte Architektur im Vergleich zur monolithischen Architektur nützliche Eigenschaften auf, um die Ziele wie z.B: hohe Flexibilität, Modularität und Skalierbarkeit zu erreichen. Das Gesamtprojekt wird in einzelne Microservices geteilt, um die Modularität zu gewährleisten. Wie in Abschnitt 4.4 konzipiert, wird jede Applikation unter Berücksichtigung ihrer Laufzeit- und Arbeitsumgebung als ein PROOF-Prozess in einem Docker-Image gekapselt und als ein Microservice in Docker-Containern automatisiert ausgeführt. Sie besitzen lediglich Ein- und Ausgaben, wissen aber nichts über deren Verknüpfung zueinander. Die Verknüpfung wird durch die Apache-NiFi-Engine auf der Weboberfläche grafisch verwaltet und mithilfe der Kommunikationsinfrastruktur und des Prozessmanagements aufgebaut. Die Änderung eines Services hat wenige Auswirkungen auf andere Services in PROOF. Diese geringe Abhängigkeit der ver-

schiedenen PROOF-Prozesse bzw. Microservices zeigt, dass PROOF modular und die Services lose gekoppelt sind.

Ein zweites Qualitätsmerkmal neben der Modularität ist die Wiederverwendbarkeit. Es hilft bei der Wartung der Software, da Komponenten wiederverwendet werden können, anstatt nur neue Komponenten zu implementieren. Damit PROOF wiederverwendbar ist, ist es wichtig, dass es nicht an eine bestimmte Domäne (z. B. Energie- oder Umweltdomäne) gebunden ist. Das wurde im Konzept 4.4 anhand der abstrakten Beschreibung des PROOF-Prozesses umgesetzt. Unabhängig von der Domäne lassen sich anhand der Containervirtualisierung wissenschaftliche Applikationen in Docker-Images packen und als PROOF-Prozesse überführen. Darüber hinaus werden die implementierten PROOF-Prozesse in der Prozess-Bibliothek abgespeichert, um eine Wiederverwendbarkeit für verschiedene Szenarien weiterhin zu unterstützen.

Analysierbarkeit ist ein drittes Unterqualitätsmerkmal, das die Fähigkeit beschreibt, das System zu protokollieren und zu überwachen. Dies führt zu messbaren Metriken und Protokollen, die analysiert werden können, um die Wartung zu erleichtern. Insbesondere Fehler oder Probleme mit der Software lassen sich mit den entsprechenden Informationen schneller finden und beheben. Neben dem Beheben von Fehlern soll eine Architektur auch aktualisiert und geändert werden, um neue Funktionen bereitzustellen. Das entsprechende Unterqualitätsmerkmal, durch das sich ändernde Software definiert, ist die Modifizierbarkeit. Da die PROOF-Architektur auf Microservices basiert, wird die Modifizierbarkeit erleichtert. Jeder Service kann separat aktualisiert oder sogar ersetzt werden.

7.4.7 Effizienz (Performance Efficiency)

Das Merkmal Effizienz stellt die Leistung im Verhältnis zur Menge der eingesetzten Ressourcen und der Zeit dar. Zur Einhaltung dieses Kriteriums wurde das Benchmarking bezüglich der Speichernutzung (s. Abschnitt 7.3.2), der Ausführungszeit im Vergleich zur direkten Ausführung (s. Abschnitt 7.3.3) und des Overheads für parallele Aufgaben (s. Abschnitt 7.3.4) durchgeführt. Die Ergebnisse des Benchmarkings zeigten, dass sich die Gesamtausführungszeit jeder parallelen Aufgabe leicht erhöhte, als die Anzahl der parallelen Aufgaben von eins auf zehn stieg. Bezogen auf die Gesamtausführungszeit (156s) ist die geringe Verlängerung (ca. 8s) durch die Parallelität akzeptabel und nahezu vernachlässigbar. Die Ausführungszeit der parallelen Tasks in PROOF blieb immer konstant, d.h., durch die vorhandene Skalierbarkeit und Parallelisierung, welche mit der Container-Orchestrierung einhergeht, kann die Leistungsfähigkeit von Workflows signifikant verbessert werden.

7.4.8 Sicherheit (Security)

Ein weiteres Qualitätsmerkmal, welches als eine neue Ergänzung zu den Qualitätsmerkmalen der Norm [13] betrachtet wird, ist die Sicherheit. Softwaresicherheit ist das Konzept der Implementierung von Mechanismen in die Sicherheitskonstruktion, die dazu beitragen, dass sie funktional (oder resistent) gegen Angriffe bleibt. In der Norm werden drei Arten von Softwaresicherheit umfasst: Sicherheit der Software selbst, Sicherheit der von der Software verarbeiteten Daten und Sicherheit der Kommunikation mit anderen Systemen über Netzwerke. Bei der vorliegenden Arbeit wurde der Sicherheitsaspekt nicht intensiv betrachtet und auf die gängigen Sicherheitskriterien wie Authentifizierung und Zugangsbeschränkungen für das Rechner-Cluster beschränkt.

7.5 Evaluation der Machbarkeit

Für die Evaluation der Machbarkeit wurde der Anforderungskatalog aus dem Abschnitt 3.1.2 herangezogen, mit dem die Arbeiten in Abschnitt 3.2 bewertet sowie der Handlungsbedarf und die Motivation für diese Arbeit ermittelt wurden. Das Ergebnis dieser Bewertung ist Tabelle 3.3 zu entnehmen. Wird die vorliegende Arbeit nun auf Basis dieses Anforderungskatalogs bewertet und mit ähnlichen Ansätzen verglichen, bringt das den Vergleich für die Machbarkeit zur Prüfung auf Vollständigkeit hinsichtlich der Erfüllung der Anforderungen. Als Grundlage dienen die Anforderungen:

A1: Hohe Konfigurierbarkeit,
A2: Einfache Integration,
A3: Performanter Datenaustausch,
A4: Unterstützung zum Ausführen großer und komplexer Workflows,
A5: Automatisierung,
A6: Plattformunabhängige grafische Oberfläche,
A7: Wiederverwendbarkeit,
A8: Skalierung und Parallelisierung,
A9: Generische Funktionalitäten.

Das Ergebnis der Bewertung dieser Arbeit zeigt Tabelle 7.3, während die einzelnen Anforderungen (A1-A9) nachfolgend im Detail erörtert werden. Abschließend werden diese Anforderungen mit den Beiträgen der vorliegenden Arbeit verknüpft, um auch dahingehend eine logische Zugehörigkeit zu erreichen.

7.5.1 A1: Hohe Konfigurierbarkeit

Die hohe Konfigurierbarkeit stellt sicher, dass Softwareapplikationen zum Aufbau eines wissenschaftlichen Workflows problemlos konfigurierbar und verknüpfbar sind, ohne dass zusätzliche Schnittstellen zur Konfiguration fest codiert und angelegt werden müssen. Nach der Integration einer Applikation in PROOF können Parameter der Applikation unter dem Tab *PROPERTIES* im Konfigurationsdialog der Weboberfläche flexibel konfiguriert werden, die mithilfe von Redis und eines Wrappers an die entsprechende Applikation im Container weitergegeben werden können. Anders ausgedrückt, die Aufgaben zum Aufbau eines Workflows in PROOF sind für Nutzer nur die Konfiguration und die Verknüpfung der Framework-Prozesse auf der Weboberfläche. Die restlichen Aufgaben zur Unterstützung konfigurierbarer Kommunikation zwischen Framework-Prozessen werden von PROOF automatisiert übernommen. Dabei ist keine weitere Schnittstelle zu implementieren, keine neue Datei anzulegen und kein Quellcode der Applikation anzupassen. Darüber hinaus bieten die konfigurierbaren Services verschiedene Funktionalitäten, z.B. der MQTT-Service zur Interaktion mit externen Applikationen/Modellen/Anlagen, der TimeseriesService zum Lesen und Schreiben der Zeitreihendaten und die Dateioperations-Services zur Verarbeitung von xlsx-, CSV-, PKL- und PDF-Dateien.

7.5.2 A2: Einfache Integration

Durch die Verwendung der generischen Framework-Prozesse, z.B. Matlab-, FMU-, Python-, Java-Prozesse usw., kann der Integrationsaufwand stark reduziert werden. Alle Applikationen/Modelle einer Software oder Programmiersprache können sich ihren generischen Framework-Prozess teilen und damit in PROOF integriert werden, ohne dass ein spezifisches Docker-Image für jede Applikation erstellt und die Applikation selbst dafür erweitert werden muss. Auch von der Containervirtualisierung profitiert die Integration von Applikationen, die auf diversen Programmiersprachen, -umgebungen bzw. Bibliotheken basieren. Durch die Verwendung der Containervirtualisierung können die Applikationen jeweils in einem Image gekapselt und danach in Containern ausgeführt werden. Die Applikationen werden unabhängig von ihrer Programmiersprache eingebunden und entkoppelt als separate PROOF-Prozesse in PROOF integriert, ohne dabei ihre Funktionalität zu beeinflussen.

Dieser kurze Setup-Prozess zur Integration von Applikationen in PROOF kann von Benutzern mit wenig Programmiererfahrung abgeschlossen werden, unabhängig davon, ob es sich bei der ausführbaren Datei um eine Eigenentwicklung handelt oder nicht. Da nur Schnittstellenkenntnisse erforderlich sind, schneidet PROOF bei

der Modellintegration deutlich besser ab als die Verwendung eines Ansatzes mit engerer Kopplung, der immer anspruchsvolle und zeitaufwändige Änderungen des Quellcodes beinhaltet. Darüber hinaus bietet PROOF den Vorteil, dass der Integrationsaufwand nur einmal pro Modell bzw. Applikation und nicht einmal pro Workflow wie bei den Tight-and-Loose-Coupling-Ansätzen [WSR17] anderer Frameworks erfolgen muss, da die erstellten Docker-Images im Docker-Image-Repository gespeichert werden und in allen zukünftigen Iterationen des Workflow-Entwicklungszyklus in verschiedenen Simulationskontexten wiederverwendet werden können, was besonders im dynamischen Forschungsbereich von Vorteil ist.

7.5.3 A3: Performanter Datenaustausch

Als ein generischer nachrichtenorientierter In-Memory-basierte Datenspeicher wird Redis zum Datenaustausch in PROOF eingesetzt. Da die Verarbeitungszeit beim Lesen und Schreiben von Daten in Redis viel schneller als andere festplattenbasierte Kommunikationsschnittstellen ist, werden kleine Datenpakete zwischen ausführbaren Applikationen mithilfe von Redis in PROOF performant übertragen. Für den Austausch großer Datensätze wird ein gemeinsames Volume in PROOF integriert. Im Volume werden große Daten abgespeichert und von allen Framework-Prozessen bzw. Container geteilt. Die großen Datensätze werden innerhalb PROOF nicht mehr übermittelt, sondern von Containern direkt benutzt, um die Verarbeitungseffizienz von großen Daten zu verbessern.

7.5.4 A4: Unterstützung zum Ausführen großer und komplexer Workflows

PROOF wird auf einem Kubernetes-Computing-Cluster eingerichtet, um genügend Ressourcen zur Bewältigung der spezifischen Belastung zu haben. Der Kubernetes-Cluster ermöglicht die horizontale Skalierung von Microservices, indem mehrere Instanzen desselben Services oder Workflows parallel ausgeführt werden. Die Services werden auf die verschiedenen Cluster-Knoten verteilt, um die Leistung des Clusters optimal zu nutzen. Unter Verwendung der Containervirtualisierung und Container-Orchestrierungs-technologie kann das Prozessmanagement umfangreiche Framework-Prozesse in Containern in Pods als Microservices auf dem Computing-Cluster verteilen und ausführen. Dies unterstützt die Ausführung großer und komplexer Workflows und Simulationen aus einer Vielzahl von Komponentenmodellen, die parallel zueinander arbeiten können.

7.5.5 A5: Automatisierung

Durch die Verwendung der PROOF-Weboberfläche können wissenschaftliche Workflows von Nutzern aufgebaut und ausgeführt werden, die nicht zu berücksichtigen brauchen, wie entsprechende Simulationen bzw. Applikationen ausgeführt und benötigte Datenflüsse in den Simulationen realisiert werden. Die Ausführung von Workflows wird in PROOF vollautomatisiert. Das Framework kümmert sich um die Laufzeitinfrastrukturen sowie Laufzeitbibliotheken für die Applikationen.

Basierend auf einer containerisierten Microservices-Architektur werden alle Applikationen mit ihren Laufzeitinfrastrukturen, Bibliotheken, Solvern, Konfigurationsdateien, usw. jeweils in einem Docker-Image gekapselt und als Framework-Prozesse in PROOF betrachtet. Die Framework-Prozesse werden vom Prozessmanagement bei Bedarf dynamisch erstellt, vollautomatisiert ausgeführt und gemanagt, um Laufzeitautomatisierung und Plattformunabhängigkeit zu gewährleisten.

7.5.6 A6: Plattformunabhängige grafische Oberfläche

Durch die Erweiterung und Integration des Web-Frameworks Apache NiFi ist PROOF plattformunabhängig verwendbar über Webbrowser. Im Vergleich zu Workflows, die basierend auf Konfigurationsdateien aufgebaut werden, brauchen Nutzer keine Programmierkenntnis für die Implementierung zusätzlicher Schnittstellen oder Konfigurationsdateien zur Erstellung von Workflows in PROOF.

Auf der Weboberfläche von PROOF repräsentieren NiFi-Prozessoren Framework-Prozesse visuell und können konfiguriert und gekoppelt werden, um wissenschaftliche Workflows flexibel abzubilden. Darüber hinaus befinden sich viele Operationsfunktionen auf der Weboberfläche, um die aufgebauten Workflows zu starten, zu stoppen, zu steuern, zu verwalten und als eine XML-Datei zur Wiederherstellung der Workflows zu exportieren.

7.5.7 A7: Wiederverwendbarkeit

Wiederverwendbarkeit ist die Fähigkeit, Framework-Prozesse und Workflows von PROOF wiederzuverwenden. Zur Wiederverwendung der Framework-Prozesse umfasst die Prozess-Bibliothek von PROOF alle integrierten Framework-Prozesse, die

wissenschaftliche Applikationen einschließen und zum Aufbau von Workflows in verschiedenen Simulationskontexten angewendet werden können. Zur Wiederverwendung der aufgebauten Workflows stellt PROOF eine XML-Vorlage zur Speicherung der Workflows bereit, auf deren Basis Workflows für andere Szenarien neu erstellt werden können.

7.5.8 A8: Skalierung und Parallelisierung

Als Container-Orchestrierungstechnologie unterstützt Kubernetes nützliche Funktionen zur Skalierung und Parallelisierung von Microservices auf dem Computing-Cluster, um die genannten Ziele, wie hohe Flexibilität, Modularität und Skalierbarkeit, zu erreichen. Jeder Microservice erfüllt einen speziellen Task und verwendet seine eigene Technologie und Programmiersprache. Dies gilt als ein wertvolles Merkmal, um parallele und skalierbare Lösungen für PROOF aufzubauen, da jeder Microservice die am besten geeigneten Technologien verwendet und horizontal skaliert werden kann.

7.5.9 A9: Generische Funktionalitäten

PROOF bietet verschiedene Hilfsbausteine für generische Funktionalitäten:

- Der PROOF-Wrapper zur dynamischen Steuerung des gekoppelten Bausteins, um die Ausführungsflexibilität von Bausteinen in Workflows zu erhöhen.
- Die generischen Framework-Prozesse zur Integration von Applikationen, um den Integrationsaufwand zu reduzieren.
- Die konfigurierbaren Services zur Interaktion mit externen Applikationen/-Modellen/Anlagen, zum Lesen und Schreiben der Zeitreihendaten und zur Operation von Dateien mit verschiedenen Datenformaten.
- Der Merge-Service zur Synchronisierung und zum Zusammenführen von mehreren Datenströmen.
- Der Primary-Prozess als Koordinator zur Steuerung seiner untergeordneten Secondary-Prozesse, um Primary-Secondary Co-Simulationen zu erstellen.

Tab. 7.3.: Bewertung dieser Arbeit im Vergleich zu bestehenden Arbeiten basierend auf den neun Anforderungen: ○-nicht erfüllt; ◐-teilweise erfüllt; ●-vollständig erfüllt.

Framework	A1	A2	A3	A4	A5	A6	A7	A8	A9
OOCoSim CoSimulating Communication Networks and Electrical System for Performance Evaluation in Smart Grid [Kim+18]	○	○	●	○	○	○	◐	◐	◐
FNCS Open-source framework for power system transmission and distribution dynamics co-simulation [Hua+17]	◐	○	●	●	○	○	◐	○	◐
HELICS Design of the HELICS high-performance transmission-distribution-communication-market co-simulation framework [Pal+17b]	○	○	●	●	●	○	◐	○	◐
Snakemake A scalable bioinformatics workflow engine [KR12]	◐	○	◐	●	●	○	◐	◐	◐
PowerNet Smart grid communication and co-simulation [LA11]	○	○	○	○	○	○	○	○	◐
Mosaik A framework for modular simulation of active components in Smart Grids [SST11]	◐	○	●	○	●	○	◐	◐	◐
HLA High Level Architecture [DKW98]	○	●	●	●	○	○	●	●	○
PROOF PROcess Operation Framework	●	●	●	●	●	●	●	●	●

7.5.10 Zuordnungen der Anforderungen zu den Beiträgen

Die hier untersuchten Anforderungen beeinflussen die Beiträge von PROOF, wie in Abschnitt 1.3 dargelegt wurde. Zusammengefasst ergibt sich die Verknüpfung der Anforderungen mit den Beiträgen dieser Arbeit, welcher Tabelle 7.4 zu entnehmen ist. Darin steht + für eine positive Relation zwischen Anforderung und aus ihr resultierenden Beiträgen, welche die identifizierten Lücken (s. Abschnitt 3.3) bestehender Arbeiten geschlossen haben.

Tab. 7.4.: Verknüpfung der Anforderungen mit den Beiträgen dieser Arbeit

Beitrag	A1	A2	A3	A4	A5	A6	A7	A8	A9
B1: Automatisierung von wissenschaftlichen Workflows	+			+	+	+	+		+
B2: Konfigurierbare Kommunikation zwischen Framework-Bausteinen	+	+	+						+
B3: Parallelisierung und Koordination von Workflows								+	+

7.6 Zusammenfassung

Dieses Evaluierungskapitel analysiert das PROOF-Konzept auf Grundlage des entwickelten Prototyps. Als Erstes wurde der Fallstudienprozess im Bereich Software-Engineering vorgestellt. Danach wurden die Einführung, das Konzept, die Implementierung und die Analyse der drei Anwendungsfälle gemäß dem Fallstudienprozess detailliert dargestellt. Im Anschluss erfolgte eine Zusammenfassung aller Framework-Prozesse, die für verschiedene Forschungsszenarien implementiert und in der Prozess-Bibliothek abgespeichert wurden.

Zur Analyse der Praxistauglichkeit fokussierte dieses Kapitel anschließend den Einsatz und die Untersuchung eines Anwendungsfalls innerhalb der Energiesystemanalyse, um das Verhalten und die Leistung von PROOF hinsichtlich der Parallelverarbeitung zu bewerten. Hierzu wurde ein Benchmarking zur Analyse der Speichernutzung, der Ausführungszeit und des Overheads für parallele Tests in PROOF geschildert.

Darauffolgend widmete sich das Kapitel der Qualitätsbewertung, bei der die acht Qualitätsmerkmale Funktionalität, Kompatibilität, Portabilität, Benutzbarkeit, Ver-

lässlichkeit, Wartbarkeit, Effizienz und Sicherheit berücksichtigt wurden. Daraus wurde geschlossen, dass der Prototyp bzw. das Konzept von PROOF eine Reihe von weiteren nicht-funktionalen Anforderungen erfüllt.

Zur Prüfung auf Vollständigkeit bezüglich der Softwarefunktionen wurde dann die Machbarkeit basierend auf der Anforderungsanalyse von Abschnitt 3.1.2 bewertet. Das Ergebnis der Bewertung dieser Arbeit wurde in Tabelle 7.3 dargestellt, aus der ersichtlich wird, dass PROOF die in der vorliegenden Arbeit enthaltenen Anforderungen vollständig abdeckt.

8 Fazit und Ausblick

Das Schlusskapitel dieser Arbeit fasst die Beiträge des Entwurfsansatzes zusammen. Am Ende steht ein Ausblick auf weitere Fragestellungen, die zeigen, wie sich diese Arbeit fortführen bzw. daran anknüpfen lässt.

8.1 Zusammenfassung

Wissenschaftler und Ingenieure, die mit dem Entwurf und der Implementierung komplexer Systemlösungen befasst sind, verwenden wissenschaftliche Workflows für ihre Simulationen, Analysen und Bewertungen. Mit zunehmender Systemkomplexität steigt auch die Komplexität dieser Workflows. Ohne Integrationswerkzeuge sind Wissenschaftler und Ingenieure jedoch oft sehr damit beschäftigt, wie zusätzliche Schnittstellen implementiert werden können, um Softwarewerkzeuge und Modellsätze zu integrieren, was ihre ursprünglichen Forschungs- oder Konstruktionsziele behindert. Daher sind die effiziente Automatisierung und parallele Simulation komplexer Workflows von zunehmender Bedeutung für die Durchführung von Systemlösungen in vielen Wissenschaftsbereichen wie z.B. Energie- und Umweltinformatik. Bei der Kopplung von heterogenen Modellen und anderen ausführbaren Dateien müssen verschiedenste Anforderungen an die Software-Infrastruktur berücksichtigt werden, um die Kompatibilität der Workflow-Komponenten zu gewährleisten. Die konsequente Nutzung fortschrittlicher Rechenkapazitäten und die Implementierung nachhaltiger Softwareentwicklungskonzepte, die maximale Effizienz und Wiederverwendbarkeit garantieren, sind weitere Themen, denen sich Wissenschaftler in Forschungsorganisationen regelmäßig stellen müssen.

Um diese Herausforderungen anzugehen, wird PROOF für die effiziente Kopplung und automatisierte Ausführung komplexer wissenschaftlicher Workflows vorgestellt. Zunächst werden in der vorliegenden Arbeit spezifische Probleme vorgestellt, die im Zusammenhang mit der effizienten Durchführung wissenschaftlicher Aufgaben zur Bereitstellung einer solchen wissenschaftlichen Workflow-Umgebung als wesentliche Software-Infrastruktur identifiziert wurden. Der Fokus der Workflows liegt auf den Energieforschungsprojekten, die die Forschung mehrerer Forschungszentren

im Zusammenhang mit intelligenten Energiesystemen bündeln, die Referenzarchitektur wird jedoch generisch konzipiert. Die Probleme werden weiter detailliert analysiert, indem unterschiedliche Anforderungen zu deren Lösung definiert werden. In einem nächsten Schritt werden diese Anforderungen verwendet, um die relevantesten Frameworks auf der Basis von wissenschaftlichen Artikeln und Veröffentlichungen zu analysieren. Für die aufgelisteten Anforderungen sind in der Literatur bereits zahlreiche relevante Arbeiten beschrieben, die jedoch nur die Teilaspekte der gewünschten Lösung adressieren. So gibt es z.B. eine ganze Reihe von Co-Simulations-Frameworks, mit der sich getrennte Simulationsanwendungen zu einer integrierten Systemsimulation zusammenfassen lassen. Auch existieren viele Automatisierungs-Frameworks, die für spezielle Anwendungsbereiche (z.B. Bildverarbeitung oder Datenanalyse) mehrere Softwarewerkzeuge zu integrierten Lösungen zusammenfassen. Das Ergebnis einer detaillierten wissenschaftlichen Recherche und Analyse zeigt aber auch, dass nicht alle geforderten Anforderungen von einer existierenden Lösung bereits vollständig erfüllt werden. Daher wird im Rahmen der vorliegenden Arbeit ein neues generisches, modulares und skalierbares Framework zur Automatisierung wissenschaftlicher Workflows, nämlich **PRO**cess **O**peration **F**ramework (PROOF), als „Proof of Concept"-Lösung für die Evaluation der Machbarkeit einer solchen Lösung konzipiert und prototypisch realisiert. Das Framework integriert einige bereits vorhandene Ansätze unter Ergänzung eigener Lösungsbausteine zu einer Gesamtlösung. Es werden drei wissenschaftliche Beiträge zu dieser Gesamtlösung vorgestellt, die auch bereits in Publikationen veröffentlicht und mit der Fachwelt diskutiert wurden, und die den Kern der vorliegenden Arbeit bilden. Die Validierung dieser Beiträge und ihrer Vorzüge erfolgt durch Vergleich mit bestehenden Ansätzen, durch Umsetzung von konkreten wissenschaftlichen Workflows, wie sie in verschiedenen Projekten zur systemischen Energieforschung in konkreten Forschungsprojekten benötigt werden. Anschließend wird der Prototyp des Frameworks dadurch in Bezug auf seine Praxistauglichkeit, Erfüllung der Anforderungen seiner technischen Laufzeiteigenschaften, wie Speichernutzung, Performanz, Skalierbarkeit etc. evaluiert.

8.2 Beiträge dieser Arbeit

Im Kontext des im Vorfeld beschriebenen Rahmens dieser Arbeit sowie der betrachteten Arbeiten in Abschnitt 3.2, sind folgende Beiträge entstanden, welche die identifizierten Lücken bestehender Ansätze schließen sollen.

B1: Konzeption und Umsetzung eines Grundkonzeptes für die Beschreibung und Ausführung von wissenschaftlichen Workflows und Co-Simulationen

Der erste wissenschaftliche Beitrag dieser Arbeit stellt die Konzeption und Umsetzung der PROOF-Architektur dar. Dieser Beitrag fokussiert vor allem Problemstellung P1 und P3.

Hierzu wurde die PROOF-Architektur basierend auf einer Multiplayer-Simulations-Architektur konzipiert, um die Ausführung von wissenschaftlichen Workflows unter Verwendung modular hinzufügbarer Prozessoren auf Computing-Umgebungen zu automatisieren. Durch die Überführung des PROOF-Konzepts in eine Microservices-Architektur, den Einsatz der Docker-Container-Virtualisierung und der Kubernetes-Container-Orchestrierung, können umfangreiche wissenschaftliche Applikationen als Framework-Prozesse in Containern bzw. in Pods als Microservices auf dem Computing-Cluster verteilt, skaliert und vollautomatisiert ausgeführt werden. Dies unterstützt die Ausführung großer und komplexer Workflows und Simulationen aus einer Vielzahl von Komponentenmodellen, die parallel zueinander arbeiten können. Durch Integration des neuen Konzepts PROOF-Wrapper wurde die dynamische Steuerung des gekoppelten PROOF-Prozesses bereitgestellt, um die Ausführungsflexibilität von PROOF-Prozessen in Workflows zu erhöhen. Basierend auf der Apache NiFi-Schnittstelle ist PROOF plattformunabhängig verwendbar über eine benutzerfreundliche Webbenutzeroberfläche, mit der Workflows aufgebaut, konfiguriert und ausgeführt werden können. Eine erweiterbare Prozessbibliothek, die die integrierten wissenschaftlichen Applikationen mit ihren Abhängigkeiten als Framework-Prozesse speichert, ist auf der Oberfläche verfügbar. Daraus können Framework-Prozesse zum Aufbau von Workflows und Simulationen in verschiedenen Applikationskontexten wiederverwendet werden.

B2: Konfigurierbare Kommunikation zwischen Framework-Prozessen

Dieser Beitrag widmet sich konfigurierbarer Datenübertragung zwischen Framework-Prozessen, um das Problem P2 zu lösen.

Durch Verwendung von Redis und Erweiterung der Prozessorbasisklasse können Parameter einer Applikation unter dem Tab *PROPERTIES* im Konfigurationsdialog der Weboberfläche flexibel konfiguriert und an die entsprechende Applikation weitergegeben werden, ohne eine zusätzliche Schnittstelle zu implementieren, Dateien anzulegen oder den Quellcode anzupassen. In PROOF wurden die generischen Framework-Prozesse, z.B. Matlab-, FMU-, Python-, Java-Prozesse usw., bereitgestellt,

um Integrationsaufwand stark zu reduzieren. Alle Applikationen/Modelle einer Software oder Programmiersprache können sich ihren generischen Framework-Prozess teilen und damit in PROOF integriert werden, ohne ein spezifisches Docker-Image für jede Applikation zu erstellen. Ein Volume wurde außerdem in PROOF zur Speicherung aller großen Daten, die nicht über Redis transportiert, sondern von allen Framework-Prozessen direkt im Volume verarbeitet werden, integriert. Den Abschluss des Beitrags bildete die Vorstellung verschiedener konfigurierbarer Services von PROOF für unterschiedliche Anwendungsfälle. Dabei wurde der MQTT-Service als Schnittstelle mit externen Applikationen/Modellen/Anlagen präsentiert. Danach wurde der Zeitreihendaten-Service zum Zugriff der entsprechenden Daten beschrieben. Letztlich wurden verschiedene Dateioperation-Services zur Verarbeitung der xlsx-, CSV-, PKL- und PDF-Datei in diesem Unterabschnitt dargestellt.

B3: Parallelisierung und Koordination von Workflows

Mit dem dritten Beitrag der vorliegenden Arbeit wird geliefert, wie Parallelisierung und Koordination von Workflows zur Leistungssteigerung in PROOF realisiert wird. Es handelt sich um die Lösung für das Problem P4.

Dabei wurden zunächst das Task-Parallelismus und das Daten-Parallelismus vom Parallel-Computing zur parallelen Verarbeitung mehrerer Aufgaben in PROOF auf verteilten Cluster-Knoten vorgestellt. Um mehrere Instanzen eines Framework-Prozesses in separaten Containern parallel auszuführen, ist nur der Parameter „concurrent Tasks" zur Festlegung der Anzahl der Container auf der Weboberfläche zu konfigurieren, ohne zusätzliche Schnittstellen zu schreiben oder Quellcode anzupassen. Die restlichen Aufgaben zur Steuerung der Container werden vom Koordinationsservice automatisiert übernommen, z.B. Eingabedaten aus einer Warteschlange anzunehmen und in verschiedene Container bzw. Instanzen zu verteilen sowie Ausgabedaten von Containern zu sammeln. Darüber hinaus schilderte Kapitel 6 das grundlegende Konzept und das Ablaufdiagramm des Merge-Services bezüglich der Synchronisierung mehrerer Datenflüsse. Dabei wurde detailliert präsentiert, wie mehrere Datenströme auf fünf verschiedene Arten (Strategien) kombiniert werden können. Die fünf Strategien zum Zusammenführen von Datenströmen können mit der in PROOF definierten deskriptiven Sprache beschrieben werden. Dieser Beitrag endet mit der Architektur der Primary-Secondary-Co-Simulation. Hierzu wurde der Primary-Prozess zur Steuerung seiner untergeordneten Secondary-Prozesse implementiert.

8.3 Ausblick

Die hier bearbeiteten Problemstellungen (vgl. Abschnitt 1.2) und die Validierung haben weitere Fragestellungen ergeben, die sich als Basis für die Weiterführung dieser Arbeit anbieten.

Automatisierung des Integrationsprozesses von neuen Applikationen

Durch die Nutzung der generischen Prozesse und des Volumens konnte die Implementierungseffizienz zur Integration von Applikationen in PROOF stark verbessert werden. Jedoch ist diese Integration von den Entwicklern in PROOF manuell durchgeführt worden. Im Rahmen der Weiterentwicklung von PROOF sollte der Integrationsprozess automatisiert werden. Hierzu sollte ein Dialog entwickelt und auf der Oberfläche bereitgestellt werden, damit Applikationen problemlos in PROOF hochgeladen werden können. Außerdem sollte ein neuer Integrationsservice dabei mitwirken und sicherstellen, Applikationen im Volumen zu speichern, eine UUID für jede Applikation zur Identifizierung in PROOF zu generieren. Daher müssen Benutzer weder mit den Entwicklern des Frameworks kommunizieren, noch über Programmierkenntnisse verfügen.

Visualisierung von Simulationsergebnissen

Diese Arbeit betrachtet in Bezug auf systematische Simulationen nur den Aufbau und die Automatisierung von Workflows, keine Visualisierung von Simulationsergebnissen. Um Simulationsprozesse zu vervollständigen und die Ergebnisdatenverwaltung, -analyse, -validierung und -evaluation zu vereinfachen, könnte eine weitere Verbesserung darin liegen, Funktionalitäten für die Datenvisualisierung von Simulationsworkflows hinzuzufügen. Hierzu sollte PROOF Benutzern Möglichkeiten auf der Weboberfläche bieten, Workflow-Ergebnisse grafisch darzustellen. Beispielsweise könnten Zeitreihendaten als Simulationsergebnisse geplottet, als Abbildungen gespeichert, teilweise gefiltert und ausgewählt werden.

Implementierung einer neuen Weboberfläche

Basierend auf Apache NiFi hat PROOF die Weboberfläche integriert und bereitgestellt, um wissenschaftliche Workflows veranschaulicht zu bilden und auszuführen. Jedoch ist diese Oberfläche wegen vieler unveränderter Definitionen im Quellcode beschränkt und schwer zu erweitern. Beispielsweise darf nur ein Datenfluss zwischen

zwei Prozessblöcken in Apache NiFi realisiert werden, nämlich 1:1 Datenübertragung. In komplexen Anwendungsfällen ist es möglich, dass ein Modell mehrere verschiedene Eingabedaten für ein anderes hat, nämlich m:n Datenübertragung, die in Apache NiFi unmöglich realisierbar ist. Deshalb sollte eine umfassendere Benutzeroberfläche implementiert werden, um eine flexible Erstellung von Workflows zu ermöglichen [LS22; Liu+23]. Damit könnten verschiedene Eingaben und Ausgaben für jeden Prozessblock definiert werden. Darüber hinaus erfordert die Verbindung zwischen Prozessblöcken ein Verständnis der rechnerischen Kausalität und der Datenformate/Standards. Daher sollte Lösungen für die Automatisierung der Verbindungsüberprüfung entwickelt werden, um die Vielzahl von Verbindungen komplexer Systeme zu erleichtern. Die Automatisierung von Verknüpfungen erfordert konsequente Metadatenbeschreibungen, um sie richtig und effizient zu nutzen.

Umsetzung weiterer Anwendungsfälle in PROOF

Im Abschnitt 7.2 wurde beschrieben, dass PROOF für viele verschiedene wissenschaftliche Energiesystemlösungen eingesetzt wurde. Jedoch für den Aufbau weiterer komplizierter Energiesysteme sollte zukünftig eine größere Prozess-Bibliothek in PROOF integriert werden, die die im Abschnitt 3.2.8 analysierten Simulationsumgebungen, z.B. Simulink, OpenDSS, DigSilent, PSLF usw., instrumentieren könnte. Darüber hinaus sollten mehr neue Anwendungsfälle aus nicht nur dem Energiesystembereich, sondern auch interdisziplinären domänenübergreifenden Forschungen in PROOF implementiert werden, um die generische Anwendbarkeit von PROOF weiter zu verbessern.

Literatur

[Abd+18] Leila Abdollahi Vayghan, Mohamed Aymen Saied, Maria Toeroe und Ferhat Khendek. “Deploying Microservice Based Applications with Kubernetes: Experiments and Lessons Learned”. In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. 2018, S. 970–973. DOI: 10.1109/CLOUD.2018.00148 (zitiert auf Seite 29).

[Alb+21] Karsten Albers, Benjamin Bolte, Max-Arno Meyer, Axel Terfloth und Anna Wißdorf. “Tool Support for Co-Simulation-Based Analysis”. In: *Model-Based Engineering of Collaborative Embedded Systems: Extensions of the SPES Methodology*. Hrsg. von Wolfgang Böhm, Manfred Broy, Cornel Klein et al. Cham: Springer International Publishing, 2021, S. 269–282. DOI: 10.1007/978-3-030-62136-0_13 (zitiert auf Seite 131).

[All+21] Ammar Allaoua, Toufik Madani Layadi, Ilhami Colak und Khaled Rouabah. “Design and Simulation of Smart-Grids using OMNeT++/Matlab-Simulink Co-simulator”. In: *2021 10th International Conference on Renewable Energy Research and Application (ICRERA)*. 2021, S. 141–145. DOI: 10.1109/ICRERA52334.2021.9598799 (zitiert auf Seite 56).

[Ama+15a] Marcelo Amaral, Jordà Polo, David Carrera et al. “Performance Evaluation of Microservices Architectures Using Containers”. In: *2015 IEEE 14th International Symposium on Network Computing and Applications*. 2015, S. 27–34. DOI: 10.1109/NCA.2015.49 (zitiert auf den Seiten 26, 27).

[Ama+15b] Bhagya Amarasekara, Chathurika Ranaweera, Ampalavanapillai Nirmalathas und Rob Evans. “Co-simulation platform for smart grid applications”. In: *2015 IEEE Innovative Smart Grid Technologies - Asia (ISGT ASIA)*. 2015, S. 1–6. DOI: 10.1109/ISGT-Asia.2015.7387091 (zitiert auf Seite 9).

[Ana17] Anaconda. *CONDA*. 2017. URL: https://docs.conda.io/projects/conda/en/latest/ (besucht am 6. Mai 2022) (zitiert auf Seite 45).

[AN11] Kyle Anderson und Amit Narayan. “Simulating integrated volt/var control and distributed demand response using GridSpice”. In: *2011 IEEE First International Workshop on Smart Grid Modeling and Simulation (SGMS)*. 2011, S. 84–89. DOI: 10.1109/SGMS.2011.6089204 (zitiert auf Seite 56).

[App+17] Riccardo R. Appino, Tillmann Mühlpfordt, Timm Faulwasser und Veit Hagenmeyer. “On solving probabilistic load flow for radial grids using polynomial chaos”. In: *2017 IEEE Manchester PowerTech*. 2017, S. 1–6. DOI: 10.1109/PTC.2017.7981264 (zitiert auf Seite 147).

[Ass22a] Modelica Association. *fmi: Functional Mock-up Interface*. 2022. URL: https://fmi-standard.org/ (besucht am 6. Mai 2022) (zitiert auf Seite 42).

[Ass22b] Modelica Association. *THYME: Toolkit for HYbrid Modeling of Electric power systems*. 2022. URL: https://web.ornl.gov/~nutarojj/thyme/docs/ (besucht am 6. Mai 2022) (zitiert auf den Seiten 56, 57).

[Ass22c] The Modelica Association. *The Modelica Association*. 2022. URL: https://modelica.org/index.html (besucht am 12. Mai 2022) (zitiert auf Seite 80).

[Bak17] Kapil Bakshi. "Microservices-based software architecture and approaches". In: März 2017, S. 1–8. DOI: 10.1109/AERO.2017.7943959 (zitiert auf den Seiten 26, 27).

[Bar+22] Luca Barbierato, Pietro Rando Mazzarino, Marco Montarolo et al. "A comparison study of co-simulation frameworks for multi-energy systems: the scalability problem". In: *Energy Informatics* 5 (Dez. 2022), S. 53. DOI: 10.1186/s42162-022-00231-6 (zitiert auf Seite 5).

[BDL21] Ulrich Bergmann, Lachlan Deer und Julian Langer. *Reproducible Data Analytic Workflows with Snakemake and 'R': An Extended Tutorial for Researchers in Business, Economics and the Social Sciences*. 2021. URL: https://lachlandeer.github.io/snakemake-econ-r-tutorial (zitiert auf Seite 44).

[Ber14] David Bernstein. "Containers and Cloud: From LXC to Docker to Kubernetes". In: *IEEE Cloud Computing* 1.3 (2014), S. 81–84. DOI: 10.1109/MCC.2014.51 (zitiert auf Seite 29).

[Bez+12] Jeff Bezanson, Stefan Karpinski, Viral B. Shah und Alan Edelman. "Julia: A Fast Dynamic Language for Technical Computing". In: *CoRR* abs/1209.5145 (2012). arXiv: 1209.5145. URL: http://arxiv.org/abs/1209.5145 (zitiert auf Seite 80).

[BAS14] Dhananjay Bhor, Kavinkadhirselvan Angappan und Krishna M. Sivalingam. "A co-simulation framework for Smart Grid wide-area monitoring networks". In: *2014 Sixth International Conference on Communication Systems and Networks (COMSNETS)*. 2014, S. 1–8. DOI: 10.1109/COMSNETS.2014.6734880 (zitiert auf Seite 9).

[Bia+15] D. Bian, M. Kuzlu, M. Pipattanasomporn, S. Rahman und Y. Wu. "Real-time co-simulation platform using OPAL-RT and OPNET for analyzing smart grid performance". In: *2015 IEEE Power Energy Society General Meeting*. 2015, S. 1–5. DOI: 10.1109/PESGM.2015.7286238 (zitiert auf den Seiten 9, 56).

[Blo+12] T. Blochwitz, Martin Otter, Johan Åkesson et al. "Functional Mockup Interface 2.0: The Standard for Tool independent Exchange of Simulation Models". In: Sep. 2012. DOI: 10.3384/ecp12076173 (zitiert auf Seite 131).

[Bor74] Barbara Borkowska. "Probabilistic Load Flow". In: *IEEE Transactions on Power Apparatus and Systems* PAS-93.3 (1974), S. 752–759. DOI: 10.1109/TPAS.1974.293973 (zitiert auf Seite 145).

[Bra+16] Eric Braun, Jianlei Liu, Clemens Düpmeier und Karl U. Stucky. *A service oriented IT architecture and its core components for the Energiewende*. Englisch. Poster präsentiert auf 5. Jahrestagung des KIT-Zentrums Energie, Blick nach Vorne - Energiesystem 2050, Karlsruhe, 15.Juni 2016. 37.98.10; LK 01. 2016 (zitiert auf den Seiten 153, 154).

[Bra+17] Eric Braun, Thorsten Schlachter, Clemens Düpmeier, Karl-Uwe Stucky und Wolfgang Suess. "A Generic Microservice Architecture for Environmental Data Management". Englisch. In: *Environmental Software Systems : Computer Science for Environmental Protection, Proceedings of the 12th IFIP WG 5.11 International Symposium, ISESS 2017, Zadar, Croatia, 10th - 12th May 2017. Ed.: J. Hřebíček*. Bd. 507. IFIP Advances in Information and Communication Technology. 37.98.10; LK 01. Springer, 2017, S. 383–394. DOI: 10.1007/978-3-319-89935-0_32 (zitiert auf Seite 109).

[BSS18] Juraj Brenkuš, Viera Stopjaková und Miroslav Slávik. "Energy Monitoring Platform for Smart Grid Applications". In: *2018 International Conference on Applied Electronics (AE)*. 2018, S. 1–2. DOI: 10.23919/AE.2018.8501445 (zitiert auf Seite 56).

[Çak+18] H. K. Çakmak, M. Kyesswa, U. G. Kühnapfel und V. Hagenmeyer. *eASiMOV - A New Framework for Power Grid Analysis*. Englisch. Poster präsentiert auf 7. Annual Conference of the KIT Energy Center (2018), Karlsruhe, Deutschland, 26. Juni 2018. 37.06.01; LK 01. 2018. DOI: 10.13140/RG.2.2.15952.20480 (zitiert auf Seite 222).

[Çak+19] Hüseyin Çakmak, Anselm Erdmann, Michael Kyesswa, Uwe Kühnapfel und Veit Hagenmeyer. "A new distributed co-simulation architecture for multi-physics based energy systems integration: Analysis of multimodal energy systems". In: *at - Automatisierungstechnik* 67.11 (2019), S. 972–983. DOI: doi:10.1515/auto-2019-0081 (zitiert auf den Seiten 152, 155).

[Cel+14] Gianni Celli, Paolo Atillio Pegoraro, Fabrizio Pilo, Giuditta Pisano und Sara Sulis. "DMS Cyber-Physical Simulation for Assessing the Impact of State Estimation and Communication Media in Smart Grid Operation". In: *IEEE Transactions on Power Systems* 29.5 (2014), S. 2436–2446. DOI: 10.1109/TPWRS.2014.2301639 (zitiert auf Seite 56).

[CSG08] D. P. Chassin, K. Schneider und C. Gerkensmeyer. "GridLAB-D: An open-source power systems modeling and simulation environment". In: *2008 IEEE/PES Transmission and Distribution Conference and Exposition*. 2008, S. 1–5. DOI: 10.1109/TDC.2008.4517260 (zitiert auf Seite 56).

[CCB08] P. Chen, Z. Chen und B. Bak-Jensen. "Probabilistic load flow: A review". In: *2008 Third International Conference on Electric Utility Deregulation and Restructuring and Power Technologies*. 2008, S. 1586–1591. DOI: 10.1109/DRPT.2008.4523658 (zitiert auf Seite 145).

[Chl+22] Malte Chlosta, Jianlei Liu, Rafael Poppenborg et al. "An adapter-based architecture for evaluating candidate solutions in energy system scheduling". Englisch. In: *Energy Informatics* 5.S4 (2022). 37.12.02; LK 01, S. 56. DOI: 10.1186/s42162-022-00246-z (zitiert auf Seite 77).

[Cho+13] Samira Chouikhi, Ines KORBI, Yacine Ghamri-Doudane und Leila Saidane. "A Comparative Study of Smart Grid Co-Simulation Platforms for Wireless Sensor Networks". In: *International Journal of Digital Information and Wireless Communications* 3 (Dez. 2013), S. 93–102 (zitiert auf Seite 56).

[Cir+14] Selim Ciraci, Jeff Daily, Jason Fuller et al. "FNCS: A Framework for Power System and Communication Networks Co-Simulation". In: *Proceedings of the Symposium on Theory of Modeling & Simulation - DEVS Integrative*. DEVS '14. Tampa, Florida: Society for Computer Simulation International, 2014 (zitiert auf Seite 40).

[Col16] Ilhami Colak. "Introduction to smart grid". In: *2016 International Smart Grid Workshop and Certificate Program (ISGWCP)*. 2016, S. 1–5. DOI: 10.1109/ISGWCP.2016.7548265 (zitiert auf Seite 7).

[Com22a] Angular Community. *Angular-the mordern web developer's platform*. 2022. URL: https://angular.io/ (besucht am 12. Mai 2022) (zitiert auf Seite 67).

[Com22b] Python Community. *Windpowerlib*. 2022. URL: https://pypi.org/project/windpowerlib/ (besucht am 12. Mai 2022) (zitiert auf Seite 207).

[Com22c] Typescript Community. *TypeScript is JavaScript with syntax for types.* 2022. URL: https://www.typescriptlang.org/ (besucht am 12. Mai 2022) (zitiert auf Seite 67).

[DM98] J.S. Dahmann und K.L. Morse. "High Level Architecture for simulation: an update". In: *Proceedings. 2nd International Workshop on Distributed Interactive Simulation and Real-Time Applications (Cat. No.98EX191)*. 1998, S. 32–40. DOI: 10.1109/DISRTA.1998.694563 (zitiert auf Seite 52).

[DKW98] Judith S. Dahmann, Frederick Kuhl und Richard Weatherly. "Standards for Simulation: As Simple As Possible But Not Simpler The High Level Architecture For Simulation". In: *SIMULATION* 71.6 (1998), S. 378–387. DOI: 10.1177/003754979807100603 (zitiert auf den Seiten 52, 59, 172).

[Dra+16] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente et al. *Microservices: yesterday, today, and tomorrow*. 2016. DOI: 10.48550/ARXIV.1606.04036 (zitiert auf Seite 26).

[Düp+16] Clemens Düpmeier, Karl-Uwe Stucky, Ralf Mikut und V. Hagenmeyer. "A Concept for the Control, Monitoring and Visualization Center in Energy Lab 2.0". In: Bd. 9424. Jan. 2016, S. 83–94. DOI: 10.1007/978-3-319-25876-8_8 (zitiert auf Seite 7).

[EO12] Erik Ela und Mark O'Malley. "Studying the Variability and Uncertainty Impacts of Variable Generation at Multiple Timescales". In: *IEEE Transactions on Power Systems* 27.3 (2012), S. 1324–1333. DOI: 10.1109/TPWRS.2012.2185816 (zitiert auf Seite 42).

[EOM98] Hilding Elmqvist, Martin Otter und Sven Erik Mattsson. "MODELICA — THE NEW OBJECT-ORIENTED MODELING LANGUAGE". In: 1998 (zitiert auf Seite 80).

[Eyi+12] Emeka Eyisi, Jia Bai, Derek Riley et al. "NCSWT: An integrated modeling and simulation tool for networked control systems". In: *Simulation Modelling Practice and Theory* 27 (2012), S. 90–111. DOI: `https://doi.org/10.1016/j.simpat.2012.05.004` (zitiert auf Seite 56).

[Fel+15] Wes Felter, Alexandre Ferreira, Ram Rajamony und Juan Rubio. "An updated performance comparison of virtual machines and Linux containers". In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2015, S. 171–172. DOI: `10.1109/ISPASS.2015.7095802` (zitiert auf Seite 27).

[FT00] Roy Thomas Fielding und Richard N. Taylor. "Architectural Styles and the Design of Network-Based Software Architectures". AAI9980887. Diss. 2000 (zitiert auf Seite 32).

[For20] Jülich Forschungszentrum. *Geospatial Land Availability for Energy Systems (GLAES)*. 2020. URL: `https://github.com/FZJ-IEK3-VSA/glaes` (besucht am 12. Mai 2022) (zitiert auf den Seiten 143, 157).

[Fou22a] Apache Software Foundation. *Apache Kafka*. 2022. URL: `https://kafka.apache.org/` (besucht am 6. Mai 2022) (zitiert auf Seite 30).

[Fou22b] Apache Software Foundation. *Apache NiFi*. 2022. URL: `https://nifi.apache.org/` (besucht am 6. Mai 2022) (zitiert auf Seite 31).

[Fou22c] Cloud Native Computing Foundation. *A Distributed, Reliable Key-Value Store for the Most Critical Data of a Distributed System*. 2022. URL: `https://etcd.io/` (besucht am 6. Mai 2022) (zitiert auf Seite 30).

[Fou23] Software Foundation. *Welcome to Apache ZooKeeper*. 2023. URL: `https://zookeeper.apache.org/` (besucht am 16. Feb. 2023) (zitiert auf Seite 30).

[Fou22d] The Linux Foundation. *Kubernetes*. 2022. URL: `https://kubernetes.io/` (besucht am 6. Mai 2022) (zitiert auf Seite 29).

[Fou22e] The R Foundation. *The R Project for Statistical Computing*. 2022. URL: `https://www.r-project.org/` (besucht am 12. Mai 2022) (zitiert auf Seite 80).

[Fre22] Free Software Foundation, Inc. *Docker Inc*. 2022. URL: `https://docs.docker.com/engine/swarm/` (besucht am 6. Mai 2022) (zitiert auf Seite 29).

[FF04] Eric Freeman und Elisabeth Freeman. *Head first design patterns - your brain on design patterns.* Jan. 2004 (zitiert auf Seite 77).

[Gam+15] Erich [VerfasserIn] Gamma, Richard Helm, Ralph [VerfasserIn] Johnson et al., Hrsg. *Design Patterns : Entwurfsmuster als Elemente wiederverwendbarer objektorientierter Software*. 1. Aufl. Authorized translation from the English language edition, 1. ed., 1995"; *Der Bestseller von Gamma und Co. in komplett neuer Übersetzung*. [Frechen]: mitp-Verl., 2015. URL: `http://d-nb.info/1048583325/04` (zitiert auf Seite 78).

[GB12] Anita Garhwal und Partha Bhattacharya. “A review on WiMAX Technology”. In: *International Journal of Advances in Computing and Information Technology* 1 (Apr. 2012), S. 167–173. DOI: 10.6088/ijacit.12.10021 (zitiert auf Seite 56).

[Gei07] Martin Geidel. “Integrated Modeling and Optimization of Multi-Carrier Energy Systems”. In: *ETH Zürich* (2007). DOI: https://doi.org/10.3929/ethz-a-005377890 (zitiert auf Seite 149).

[Gei+07] M. Geidl, G. Koeppel, P. Favre-Perrod et al. “Energy hubs for the future”. In: *IEEE Power and Energy Magazine* 5.1 (2007), S. 24–30. DOI: 10.1109/MPAE.2007.264850 (zitiert auf Seite 149).

[Geo+12] Hanno Georg, Christian Wietfeld, Sven Christian Müller und Christian Rehtanz. “A HLA based simulator architecture for co-simulating ICT based power system control and protection systems”. In: *2012 IEEE Third International Conference on Smart Grid Communications (SmartGridComm)*. 2012, S. 264–269. DOI: 10.1109/SmartGridComm.2012.6485994 (zitiert auf Seite 53).

[GCM22] Federico M. Giorgi, Carmine Ceraolo und Daniele Mercatelli. “The R Language: An Engine for Bioinformatics and Data Science”. In: *Life* 12.5 (2022). DOI: 10.3390/life12050648 (zitiert auf Seite 80).

[Gmb22] GAMS Software GmbH. *General Algebraic Modeling System*. 2022. URL: https://www.gams.com/ (besucht am 12. Mai 2022) (zitiert auf Seite 80).

[God+10] Tim Godfrey, Sara Mullen, David W. Griffith et al. “Modeling Smart Grid Applications with Co-Simulation”. In: *2010 First IEEE International Conference on Smart Grid Communications*. 2010, S. 291–296. DOI: 10.1109/SMARTGRID.2010.5622057 (zitiert auf den Seiten 56, 57).

[GM15] Lipika Bose Goel und Rana Majumdar. “Handling mutual exclusion in a distributed application through Zookeeper”. In: *2015 International Conference on Advances in Computer Engineering and Applications*. 2015, S. 457–460. DOI: 10.1109/ICACEA.2015.7164748 (zitiert auf Seite 30).

[Gom+18] Cláudio Gomes, Casper Thule, David Broman, Peter Larsen und Hans Vangheluwe. “Co-Simulation: A Survey”. In: *ACM Computing Surveys* 51 (Mai 2018). DOI: 10.1145/3179993 (zitiert auf Seite 63).

[Gon+20] Jorge Ángel González Ordiano, Lutz Gröll, Ralf Mikut und Veit Hagenmeyer. “Probabilistic energy forecasting using the nearest neighbors quantile filter and quantile regression”. In: *International Journal of Forecasting* 36.2 (2020), S. 310–323. DOI: https://doi.org/10.1016/j.ijforecast.2019.06.003 (zitiert auf den Seiten 145, 146).

[Gon+21] Jorge Ángel González-Ordiano, Tillmann Mühlpfordt, Eric Braun et al. “Probabilistic forecasts of the distribution grid state using data-driven forecasts and probabilistic power flow”. In: *Applied Energy* 302 (2021), S. 117498. DOI: https://doi.org/10.1016/j.apenergy.2021.117498 (zitiert auf den Seiten 109, 145, 146, 148, 153, 154, 223, 227).

[Gou+17] Sebastien Gougeaud, Soraya Zertal, Jacques-Charles Lafoucriere und Philippe Deniel. “Using ZeroMQ as communication/synchronization mechanisms for IO requests simulation”. In: *2017 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*. 2017, S. 1–8. DOI: 10.23919/SPECTS.2017.8046773 (zitiert auf Seite 51).

[Gra+20] Christian Granrath, Max-Arno Meyer, Jakob Andert et al. “EleMA: A reference simulation model architecture and interface standard for modeling and testing of electric vehicles”. In: *eTransportation* 4 (2020), S. 100060. DOI: https://doi.org/10.1016/j.etran.2020.100060 (zitiert auf Seite 131).

[Gup+17] Adity Gupta, Swati Tyagi, Nupur Panwar, Shelly Sachdeva und Upaang Saxena. “NoSQL databases: Critical analysis and comparison”. In: *2017 International Conference on Computing and Communication Technologies for Smart Nation (IC3TSN)*. 2017, S. 293–299. DOI: 10.1109/IC3TSN.2017.8284494 (zitiert auf Seite 68).

[Güt+20] Moritz Gütlein, Wojciech Baron, Christopher Renner und Anatoli Djanatliev. “Performance Evaluation of HLA RTI Implementations”. In: *2020 IEEE/ACM 24th International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*. 2020, S. 1–8. DOI: 10.1109/DS-RT50469.2020.9213641 (zitiert auf Seite 55).

[HP17] Irene Hafner und Niki Popper. “On the terminology and structuring of co-simulation methods”. In: Dez. 2017, S. 67–76. DOI: 10.1145/3158191.3158203 (zitiert auf den Seiten 3, 16).

[Hag+16] Veit Hagenmeyer, Hüseyin Kemal Cakmak, Clemens Düpmeier et al. “Information and communication technology in energy lab 2.0: Smart energies system simulation and control center with an open-street-map-based power flow simulation example”. Englisch. In: *Energy Technology* 4.1 (2016). 37.98.10; LK 01, S. 145–162. DOI: 10.1002/ente.201500304 (zitiert auf Seite 7).

[Al-12] Ahmad T. Al-Hammouri. “A comprehensive co-simulation platform for cyber-physical systems”. In: *Computer Communications* 36.1 (2012), S. 8–19. DOI: https://doi.org/10.1016/j.comcom.2012.01.003 (zitiert auf Seite 56).

[ABL08] Ahmad T. Al-Hammouri, Michael S. Branicky und Vincenzo Liberatore. “Co-simulation Tools for Networked Control Systems”. In: *Hybrid Systems: Computation and Control*. Hrsg. von Magnus Egerstedt und Bud Mishra. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, S. 16–29 (zitiert auf Seite 56).

[Han+17] Jacob Hansen, Thomas Edgar, Jeff Daily und Di Wu. “Evaluating transactive controls of integrated transmission and distribution systems using the Framework for Network Co-Simulation”. In: *2017 American Control Conference (ACC)*. 2017, S. 4010–4017. DOI: 10.23919/ACC.2017.7963570 (zitiert auf Seite 40).

[HS17] Wilhelm Hasselbring und Guido Steinacker. “Microservice Architectures for Scalability, Agility and Reliability in E-Commerce”. In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. 2017, S. 243–246. DOI: 10.1109/ICSAW.2017.11 (zitiert auf Seite 26).

[Hei+21] Benedikt Heidrich, Andreas Bartschat, Marian Turowski et al. *pyWATTS: Python Workflow Automation Tool for Time Series*. 2021. DOI: 10.48550/ARXIV.2106.10157 (zitiert auf Seite 155).

[HEL22] HELMHOLTZ. *Energie System 2050*. 2022. URL: https://www.helmholtz.de/forschung/forschungsbereiche/energie/energie-system-2050/ (besucht am 12. Mai 2022) (zitiert auf den Seiten 7, 152).

[Hic22] Lan Hickson. *HTML5*. 2022. URL: https://dev.w3.org/html5/spec-LC/ (besucht am 12. Mai 2022) (zitiert auf Seite 67).

[HOS23] HOSTINGER. *Cron Job: A Comprehensive Guide for Beginners 2023*. 2023. URL: https://www.hostinger.com/tutorials/cron-job (besucht am 16. Feb. 2023) (zitiert auf Seite 90).

[Hua+17] Renke Huang, Rui Fan, Jeff Daily, Andrew Fisher und Jason Fuller. "An Open-source Framework for Power System Transmission and Distribution Dynamics Co-simulation". In: *IET Generation, Transmission & Distribution* 11 (Juni 2017). DOI: 10.1049/iet-gtd.2016.1556 (zitiert auf den Seiten 40, 41, 59, 172).

[01] "IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) - HLA Object Model Template (OMT) Specification". In: *IEEE Std 1516.2-2000* (2001), S. 1–140. DOI: 10.1109/IEEESTD.2001.92423 (zitiert auf Seite 53).

[Inc22] Docker Inc. *Docker*. 2022. URL: https://www.docker.com/ (besucht am 6. Mai 2022) (zitiert auf den Seiten 27, 28).

[Inf22] InfluxData. *influxdata*. 2022. URL: https://www.influxdata.com/ (besucht am 12. Mai 2022) (zitiert auf Seite 68).

[13] "ISO / IEC 25010 : 2011 Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models". In: 2013 (zitiert auf den Seiten 161, 165, 167).

[JP05] A. Jedlitschka und D. Pfahl. "Reporting guidelines for controlled experiments in software engineering". In: *2005 International Symposium on Empirical Software Engineering, 2005.* 2005, S. 10. DOI: 10.1109/ISESE.2005.1541818 (zitiert auf Seite 139).

[Jul22] JuliaLang. *The Julia Programming Language*. 2022. URL: https://julialang.org/ (besucht am 12. Mai 2022) (zitiert auf Seite 80).

[KLZ04] F. Kargl, E. Lawrence und G.V. Zaruba. "Introduction to the minitrack on wireless personal area networks (WPANs)". In: *37th Annual Hawaii International Conference on System Sciences, 2004. Proceedings of the*. 2004, S. 1. DOI: 10.1109/HICSS.2004.1265716 (zitiert auf Seite 56).

[Kel+15] Brian M. Kelley, Philip Top, Steven G. Smith, Carol S. Woodward und Liang Min. "A federated simulation toolkit for electric power grid and communication network co-simulation". In: *2015 Workshop on Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES)*. 2015, S. 1–6. DOI: 10.1109/MSCPES.2015.7115406 (zitiert auf Seite 42).

[kh22] Sadie kh. *Smart Grid*. 2022. URL: https://sites.google.com/site/smartgriditbp103/home (besucht am 6. Mai 2022) (zitiert auf Seite 7).

[Kha+18] Hatem Khalloof, Wilfried Jakob, Jianlei Liu et al. "A generic distributed microservices and container based framework for metaheuristic optimization". Englisch. In: *Proceedings of the Genetic and Evolutionary Conference Companion, Kyoto, J, July 15-19, 2018*. 2018 Genetic and Evolutionary Computation Conference. GECCO 2018 (Kyōto, Japan, 15.–19. Juli 2018). 37.98.11; LK 01. Association for Computing Machinery (ACM), 2018, S. 1363–1370. DOI: 10.1145/3205651.3208253 (zitiert auf den Seiten 152, 153, 155).

[Kha+16] Hamzeh Khazaei, Cornel Barna, Nasim Beigi-Mohammadi und Marin Litoiu. "Efficiency Analysis of Provisioning Microservices". In: *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. 2016, S. 261–268. DOI: 10.1109/CloudCom.2016.0051 (zitiert auf Seite 26).

[Kim+18] Hwantae Kim, Kangho Kim, Seongjoon Park, Hyunsoon Kim und Hwangnam Kim. "CoSimulating Communication Networks and Electrical System for Performance Evaluation in Smart Grid". In: *Applied Sciences* 8.1 (2018). DOI: 10.3390/app8010085 (zitiert auf den Seiten 38, 39, 59, 172).

[KPK13] Hwantae Kim, Wonkyun Park und Hwangnam Kim. "Towards CoSimulating network and electrical systems for performance evaluation in Smart Grid". English. In: *28th Annual ACM Symposium on Applied Computing, SAC 2013*. Proceedings of the ACM Symposium on Applied Computing. 28th Annual ACM Symposium on Applied Computing, SAC 2013 ; Conference date: 18-03-2013 Through 22-03-2013. 2013, S. 686–687. DOI: 10.1145/2480362.2480493 (zitiert auf Seite 38).

[KIT22a] KIT. *Energiewende: Neue Technologien für die Sektorenkopplung*. 2022. URL: https://www.kit.edu/kit/pi_2021_093_energiewende-neue-technologien-fur-die-sektorenkopplung.php (besucht am 12. Mai 2022) (zitiert auf Seite 153).

[KIT22b] KIT. *Energy Lab 2.0*. 2022. URL: https://www.elab2.kit.edu/ (besucht am 12. Mai 2022) (zitiert auf den Seiten 152, 155).

[KIT22c] KIT-IAI. *IT-Komponenten für die Energiewende*. 2022. URL: https://www.iai.kit.edu/IT-Komponenten_Energiewende.php (besucht am 12. Mai 2022) (zitiert auf Seite 153).

[KH01] Roger Koenker und Kevin F. Hallock. "Quantile Regression". In: *Journal of Economic Perspectives* 15.4 (Dez. 2001), S. 143–156. DOI: 10.1257/jep.15.4.143 (zitiert auf Seite 145).

[KR12] Johannes Köster und Sven Rahmann. "Snakemake—a scalable bioinformatics workflow engine". In: *Bioinformatics* 28.19 (Aug. 2012), S. 2520–2522. DOI: 10.1093/bioinformatics/bts480. eprint: https://academic.oup.com/bioinformatics/article-pdf/28/19/2520/819790/bts480.pdf (zitiert auf den Seiten 44, 59, 172).

[KS10] R. Kübler und Werner Schiehlen. “Two Methods of Simulator Coupling”. In: *Mathematical and Computer Modelling of Dynamical Systems* 6 (Aug. 2010), S. 93–113. DOI: 10.1076/1387-3954%28200006%296%3A2%3B1-M%3BFT093 (zitiert auf Seite 4).

[Kye+17] Michael Kyesswa, Hüseyin Çakmak, Uwe Kühnapfel und Veit Hagenmeyer. “A Matlab-Based Dynamic Simulation Module for Power System Transients Analysis in the eASiMOV Framework”. In: *2017 European Modelling Symposium (EMS)*. 2017, S. 157–162. DOI: 10.1109/EMS.2017.36 (zitiert auf Seite 146).

[LSS05] Timothy Lethbridge, Susan Sim und Janice Singer. “Studying Software Engineers: Data Collection Techniques for Software Field Studies”. In: *Empirical Software Engineering* 10 (Juli 2005), S. 311–341. DOI: 10.1007/s10664-005-1290-x (zitiert auf Seite 139).

[Lév+12] Martin Lévesque, Da Qian Xu, Géza Joós und Martin Maier. “Co-Simulation of PEV coordination schemes over a FiWi Smart Grid communications infrastructure”. In: *IECON 2012 - 38th Annual Conference on IEEE Industrial Electronics Society*. 2012, S. 2901–2906. DOI: 10.1109/IECON.2012.6389434 (zitiert auf Seite 56).

[LX16] Yunchun Li und Yumeng Xia. “Auto-scaling web applications in hybrid cloud based on docker”. In: *2016 5th International Conference on Computer Science and Network Technology (ICCSNT)*. 2016, S. 75–79. DOI: 10.1109/ICCSNT.2016.8070122 (zitiert auf Seite 28).

[LA11] Vincenzo Liberatore und Ahmad Al-Hammouri. “Smart grid communication and co-simulation”. In: *IEEE 2011 EnergyTech*. 2011, S. 1–5. DOI: 10.1109/EnergyTech.2011.5948542 (zitiert auf den Seiten 46, 56, 57, 59, 172).

[Lin+11] Hua Lin, Santhoshkumar Sambamoorthy, Sandeep Shukla, James Thorp und Lamine Mili. “Power system and communication network co-simulation for smart grid applications”. In: *ISGT 2011*. 2011, S. 1–6. DOI: 10.1109/ISGT.2011.5759166 (zitiert auf den Seiten 21, 56, 57).

[Liu+12] Buquan Liu, Yiping Yao, Zhiwen Jiang et al. “HLA-Based Parallel Simulation: A Case Study”. In: *2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation*. 2012, S. 65–67. DOI: 10.1109/PADS.2012.21 (zitiert auf Seite 55).

[Liu+18] Jianlei Liu, Eric Braun, Clemens Düpmeier et al. “A Generic and Highly Scalable Framework for the Automation and Execution of Scientific Data Processing and Simulation Workflows”. In: *2018 IEEE International Conference on Software Architecture (ICSA)*. 2018, S. 145–14510. DOI: 10.1109/ICSA.2018.00024 (zitiert auf den Seiten 27, 28, 32, 111, 142, 145, 153, 154, 225).

[Liu+19] Jianlei Liu, Eric Braun, Clemens Düpmeier et al. “Architectural Concept and Evaluation of a Framework for the Efficient Automation of Computational Scientific Workflows: An Energy Systems Analysis Example”. In: *Applied Sciences* 9 (Feb. 2019). DOI: 10.3390/app9040728 (zitiert auf den Seiten 3, 111, 145, 153, 154).

[Liu+23] Jianlei Liu, Malte Chlosta, Nicolas Schaber et al. “Introducing PROOF - A PROcess Orchestration Framework for the Automation of Computational Scientific Workflows and Co-Simulations”. In: *2023 Open Source Modelling and Simulation of Energy Systems (OSMSES)*. 2023, S. 1–6. DOI: 10.1109/OSMSES58477.2023.10089680 (zitiert auf Seite 180).

[LD16] Jianlei Liu und Clemens Düpmeier. *Basic concepts of the energy system 2050 co-simulation platform*. Englisch. Poster präsentiert auf Doktorandenworkshop der Initiative EnergieSystem 2050, Friedrichsdorf, 7.-8. Dezember 2016. 37.98.10; LK 01. 2016 (zitiert auf den Seiten 153, 154).

[LDH17] Jianlei Liu, Clemens Düpmeier und V. Hagenmeyer. “A New Concept of a Generic Co-Simulation Platform for Energy Systems Modeling”. In: *FTC 2017 - Future Technologies Conference 2017*. Nov. 2017 (zitiert auf den Seiten 30, 153, 154).

[LS22] Jianlei Liu und Thorsten Schlachter. *PROOF - Process Orchestration Framework, ICT Platform for Energy System Research*. Englisch. Poster präsentiert auf Exkursion des Instituts für Automation und angewandte Informatik (2022), Karlsruher Institut für Technologie, 20.–21. Juli 2022. 37.12.03; LK 01. 2022 (zitiert auf Seite 180).

[Ltd22a] Redis Ltd. *Redis*. 2022. URL: https://redis.io/ (besucht am 6. Mai 2022) (zitiert auf Seite 31).

[Ltd22b] Redis Ltd. *Redis Enterprise and Kafka*. 2022. URL: https://redis.com/comparisons/redis-enterprise-and-kafka/ (besucht am 6. Mai 2022) (zitiert auf Seite 96).

[MI04] D. Mahrenholz und S. Ivanov. “Real-Time Network Emulation with ns-2”. In: *Eighth IEEE International Symposium on Distributed Simulation and Real-Time Applications*. 2004, S. 29–36. DOI: 10.1109/DS-RT.2004.34 (zitiert auf Seite 46).

[MR09] Lindokuhle Malinga und Herman le Roux. “HLA RTI performance evaluation”. In: (Juli 2009) (zitiert auf Seite 55).

[McD23] Gavin McDonald. *Apache ZooKeeper*. 2023. URL: https://cwiki.apache.org/confluence/display/ZOOKEEPER/ (besucht am 16. Feb. 2023) (zitiert auf Seite 30).

[Mer14] Dirk Merkel. “Docker: lightweight Linux containers for consistent development and deployment”. In: *Linux Journal* 2014 (März 2014) (zitiert auf Seite 28).

[MOD14] Kevin Mets, Juan Aparicio Ojea und Chris Develder. “Combining Power and Communication Network Simulation for Cost-Effective Smart Grid Analysis”. In: *IEEE Communications Surveys Tutorials* 16.3 (2014), S. 1771–1796. DOI: 10.1109/SURV.2014.021414.00116 (zitiert auf den Seiten 16, 18, 25).

[MGH20] Ralf Mikut, Jorge Angel Ludwig González Ordiano Nicole und Veit Hagenmeyer. *Prognosen für Energiesysteme der Zukunft: Methoden und Tools*. Poster präsentiert auf ES2050-Abschlusskonferenz (2020), Berlin, Deutschland, 28.–30. September 2020. 37.98.11; LK 01. 2020 (zitiert auf Seite 152).

[MK20] Biswajeeban Mishra und Attila Kertesz. “The Use of MQTT in M2M and IoT Systems: A Survey”. In: *IEEE Access* 8 (2020), S. 201071–201086. DOI: 10.1109/ACCESS.2020.3035849 (zitiert auf Seite 108).

[Mod+18] Arsh Modak, S. D. Chaudhary, P. S. Paygude und S. R. Ldate. “Techniques to Secure Data on Cloud: Docker Swarm or Kubernetes?” In: *2018 Second International Conference on Inventive Communication and Computational Technologies (ICICCT)*. 2018, S. 7–12. DOI: 10.1109/ICICCT.2018.8473104 (zitiert auf Seite 29).

[Moh+17] Mohammad Mohammadi, Younes Noorollahi, Behnam Mohammadi-ivatloo und Hossein Yousefi. “Energy hub: From a model to a concept – A review”. In: *Renewable and Sustainable Energy Reviews* 80 (2017), S. 1512–1527. DOI: https://doi.org/10.1016/j.rser.2017.07.030 (zitiert auf Seite 149).

[Möl+21] Felix Mölder, Kim Jablonski, Brice Letcher et al. “Sustainable data analysis with Snakemake”. In: *F1000Research* 10 (Apr. 2021), S. 33. DOI: 10.12688/f1000research.29032.2 (zitiert auf Seite 45).

[MP04] Katherine L. Morse und Mikel D. Petty. “High Level Architecture Data Distribution Management migration from DoD 1.3 to IEEE 1516”. In: *Concurrency and Computation: Practice and Experience* 16 (2004) (zitiert auf Seite 130).

[MQT22] MQTT.org. *MQTT: The Standard for IoT Messaging*. 2022. URL: https://mqtt.org/ (besucht am 12. Mai 2022) (zitiert auf Seite 108).

[Nai16] Nitin Naik. “Building a virtual system of systems using docker swarm in multiple clouds”. English. In: *ISSE 2016 - 2016 International Symposium on Systems Engineering - Proceedings Papers*. ISSE 2016 - 2016 International Symposium on Systems Engineering - Proceedings Papers. United States: IEEE, Nov. 2016. DOI: 10.1109/SysEng.2016.7753148 (zitiert auf Seite 26).

[Nai17] Nitin Naik. “Docker container-based big data processing system in multiple clouds for everyone”. English. In: *2017 IEEE International Systems Engineering Symposium (ISSE)*. 2017 IEEE International Symposium on Systems Engineering ; Conference date: 11-10-2017 Through 13-10-2017. IEEE, Okt. 2017. DOI: 10.1109/SysEng.2017.8088294 (zitiert auf Seite 28).

[Ngu+17] Van Hoa Nguyen, Yvon Besanger, Quoc Tuan Tran et al. “Using power-hardware-in-the-loop experiments together with co-simulation for the holistic validation of cyber-physical energy systems”. In: *2017 IEEE PES Innovative Smart Grid Technologies Conference Europe (ISGT-Europe)*. 2017, S. 1–6. DOI: 10.1109/ISGTEurope.2017.8260122 (zitiert auf Seite 131).

[NRS09] Eric Noulard, Jean-Yves Rousselot und Pierre Siron. “CERTI, an Open Source RTI, why and how”. In: *Spring Simulation Interoperability Workshop*. San Diego, US, 2009, S. 1–11. URL: https://oatao.univ-toulouse.fr/2056/ (zitiert auf Seite 53).

[Nut+07] James Nutaro, Phani Teja Kuruganti, Laurie Miller, Sara Mullen und Mallikarjun Shankar. “Integrated Hybrid-Simulation of Electric Power and Communications Systems”. In: *2007 IEEE Power Engineering Society General Meeting*. 2007, S. 1–8. DOI: 10.1109/PES.2007.386202 (zitiert auf den Seiten 56, 57).

[Nut22] Jim Nutaro. *Adevs (A Discrete EVent System simulator)*. 2022. URL: https://web.ornl.gov/~nutarojj/adevs/ (besucht am 6. Mai 2022) (zitiert auf Seite 56).

[OAS23] OASIS. *AMQP-Advanced Message Queuing Protocol*. 2023. URL: https://www.amqp.org/ (besucht am 16. Feb. 2023) (zitiert auf Seite 30).

[OFF22a] OFFIS. *Easy-to-use GUI for mosaik available*. 2022. URL: https://mosaik.offis.de/2017/06/21/maverig_installation/ (besucht am 6. Mai 2022) (zitiert auf Seite 51).

[OFF22b] OFFIS. *ZeroMQ: An open-source universal messaging library*. 2022. URL: https://zeromq.org/ (besucht am 6. Mai 2022) (zitiert auf Seite 51).

[Ope22] OpenTSDB. *The Scalable Time Series Database*. 2022. URL: http://opentsdb.net/ (besucht am 12. Mai 2022) (zitiert auf Seite 68).

[Ove09] Maarten Overdijk. “Appropriation of technology for collaboration : From mastery to utilisation”. In: (Jan. 2009) (zitiert auf Seite 3).

[Pal+17a] Peter Palensky, Arjen A. Van Der Meer, Claudio David Lopez, Arun Joseph und Kaikai Pan. “Cosimulation of Intelligent Power Systems: Fundamentals, Software Architecture, Numerics, and Coupling”. In: *IEEE Industrial Electronics Magazine* 11.1 (2017), S. 34–50. DOI: 10.1109/MIE.2016.2639825 (zitiert auf den Seiten 15, 16, 19, 21–25).

[Pal+17b] Bryan Palmintier, Dheepak Krishnamurthy, Philip Top et al. “Design of the HELICS high-performance transmission-distribution-communication-market co-simulation framework”. In: *2017 Workshop on Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES)*. 2017, S. 1–6. DOI: 10.1109/MSCPES.2017.8064542 (zitiert auf den Seiten 42, 43, 59, 172).

[Pau+14] Shuva Paul, Md Sajed Rabbani, Ripon Kumar Kundu und Sikdar Mohammad Raihan Zaman. “A review of smart technology (Smart Grid) and its features”. In: *2014 1st International Conference on Non Conventional Energy (ICONCE 2014)*. 2014, S. 200–203. DOI: 10.1109/ICONCE.2014.6808719 (zitiert auf Seite 7).

[PBR13] Fidelis Perkonigg, Djordje Brujic und Mike Ristic. “MAC-Sim: A multi-agent and communication network simulation platform for smart grid applications based on established technologies”. In: *2013 IEEE International Conference on Smart Grid Communications (SmartGridComm)*. 2013, S. 570–575. DOI: 10.1109/SmartGridComm.2013.6688019 (zitiert auf Seite 56).

[PK17] Hazael Phiri und Douglas Kunda. “A Comparative Study of NoSQL and Relational Database”. In: *Zambia ICT Journal* 1 (Dez. 2017), S. 1–4. DOI: 10.33260/zictjournal.v1i1.8 (zitiert auf Seite 68).

[PLD20] Rafael Poppenborg, Jianlei Liu und Clemens Düpmeier. *Using a scientific workflow platform for simulating the operation of a composed distributed energy resource*. Englisch. Poster präsentiert auf ES2050-Abschlusskonferenz (2020), Berlin, Deutschland, 28.–30. September 2020. 37.98.11; LK 01. 2020 (zitiert auf den Seiten 153, 154).

[Pop+21] Rafael Poppenborg, Johannes Ruf, Malte Chlosta et al. "Energy Hub Gas : A Multi-Domain System Modelling and Co-Simulation Approach". Englisch. In: *Proceedings of the 9th Workshop on Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES '21)*. MSCPES 2021 (Online). 37.12.03; LK 01. Association for Computing Machinery (ACM), 2021, Art.Nr. 12. DOI: 10.1145/3470481.3472712 (zitiert auf den Seiten 109, 111, 130, 134, 149, 152–155, 229).

[PJ17] B Rajanarayan Prusty und Debashisha Jena. "A critical review on probabilistic load flow studies in uncertainty constrained power systems with photovoltaic generation and a new approach". In: *Renewable and Sustainable Energy Reviews* 69 (2017), S. 1286–1302. DOI: https://doi.org/10.1016/j.rser.2016.12.044 (zitiert auf Seite 145).

[Ram+20] Umar Hanif Ramadhani, Mahmoud Shepero, Joakim Munkhammar, Joakim Widén und Nicholas Etherden. "Review of probabilistic load flow approaches for power distribution systems with photovoltaic generation and electric vehicle charging". In: *International Journal of Electrical Power & Energy Systems* 120 (2020), S. 106003. DOI: https://doi.org/10.1016/j.ijepes.2020.106003 (zitiert auf Seite 145).

[Red+20] Florian Redder, Eric Braun, Clemens Düpmeier et al. *ICT-Platform for Designing, Testing and Evaluating Energy System Solutions in Real-World Laboratories*. Englisch. Poster präsentiert auf ES2050-Abschlusskonferenz (2020), Berlin, Deutschland, 28.–30. September 2020. 37.98.11; LK 01. 2020 (zitiert auf Seite 153).

[Ric22] Chris Richardson. *Microservices*. 2022. URL: https://microservices.io/ (besucht am 3. Jan. 2019) (zitiert auf Seite 26).

[RH10] George F. Riley und Thomas R. Henderson. "The ns-3 Network Simulator". In: *Modeling and Tools for Network Simulation*. Hrsg. von Klaus Wehrle, Mesut Güneş und James Gross. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, S. 15–34. DOI: 10.1007/978-3-642-12331-3_2 (zitiert auf Seite 42).

[Rob15] Martin Robinius. "Strom- und Gasmarktdesign zur Versorgung des deutschen Straßenverkehrs mit Wasserstoff". Dissertation. Jülich: RWTH Aachen, 2015, 1 Online–Ressource (VI, 271 Seiten) : Illustrationen, Diagramme, Karten. URL: https://publications.rwth-aachen.de/record/565873 (zitiert auf den Seiten 143, 157).

[Rob+17] Martin Robinius, Alexander Otto, Konstantinos Syranidis et al. "Linking the Power and Transport Sectors—Part 2: Modelling a Sector Coupling Scenario for Germany". In: *Energies* 10.7 (2017). DOI: 10.3390/en10070957 (zitiert auf den Seiten 143, 157).

[RM17] Colin Robson und Kieran McCartan. *Real World Research, 4th Edition*. Dez. 2017 (zitiert auf Seite 138).

[Ros22] Richard E. Rosenthal. *A GAMS TUTORIAL*. 2022. URL: https://www.gams.com/docs/pdf/Tutorial.PDF (zitiert auf Seite 80).

[Run+12] Per Runeson, Martin Höst, Austen Rainer und Björn Regnell. *Case Study Research in Software Engineering – Guidelines and Examples*. Feb. 2012. DOI: 10.1002/9781118181034 (zitiert auf den Seiten 138, 139).

[RRS17] David Ryberg, Martin Robinius und Detlef Stolten. "Methodological Framework for Determining the Land Eligibility of Renewable Energy Sources". In: (Dez. 2017) (zitiert auf Seite 143).

[Ryb+19] David Severin Ryberg, Dilara Gulcin Caglayan, Sabrina Schmitt et al. "The future of European onshore wind energy potential: Detailed distribution and simulation of advanced turbine designs". In: *Energy* 182 (2019), S. 1222–1238. DOI: https://doi.org/10.1016/j.energy.2019.06.052 (zitiert auf Seite 157).

[RRS18] David Severin Ryberg, Martin Robinius und Detlef Stolten. "Evaluating Land Eligibility Constraints of Renewable Energy Sources in Europe". In: *Energies* 11.5 (2018). DOI: 10.3390/en11051246 (zitiert auf Seite 157).

[Sad+15] Mohammad Sadi, Dipankar Dasgupta, Mohd Hasan Ali und Robert Abercrombie. "OPNET/Simulink Based Testbed for Disturbance Detection in the Smart Grid". In: Apr. 2015. DOI: 10.1145/2746266.2746283 (zitiert auf Seite 38).

[Sad+18] Severin Sadjina, Lars Kyllingstad, Martin Rindarøy et al. "Distributed Co-Simulation of Maritime Systems and Operations". In: (Sep. 2018). DOI: 10.1115/1.4040473 (zitiert auf Seite 4).

[Sar14] S.T. Sarena. *A Framework for Microgrid Analysis Using OpenDSS*. Lap Lambert Academic Publishing GmbH KG, 2014. URL: https://books.google.de/books?id=ze2KoAEACAAJ (zitiert auf Seite 38).

[SBS17] M. Sarnovsky, P. Bednar und M. Smatana. "Data integration in scalable data analytics platform for process industries". In: *2017 IEEE 21st International Conference on Intelligent Engineering Systems (INES)*. 2017, S. 000187–000192. DOI: 10.1109/INES.2017.8118553 (zitiert auf den Seiten 31, 32).

[Sch+15] Florian Schloegl, Sebastian Rohjans, Sebastian Lehnhoff et al. "Towards a classification scheme for co-simulation approaches in energy systems". In: *2015 International Symposium on Smart Electric Distribution Systems and Technologies (EDST)*. 2015, S. 516–521. DOI: 10.1109/SEDST.2015.7315262 (zitiert auf den Seiten 15, 17, 18, 24, 25).

[SSS12] Steffen Schütte, S. Scherfke und Michael Sonnenschein. "mosaik - Smart Grid Simulation API". In: Apr. 2012. URL: https://mosaik.offis.de/downloads/mosaik_SimAPI_SmartGreens2012.pdf (zitiert auf Seite 48).

[SST11] Steffen Schütte, Stefan Scherfke und Martin Tröschel. “Mosaik: A framework for modular simulation of active components in Smart Grids”. In: *2011 IEEE First International Workshop on Smart Grid Modeling and Simulation (SGMS)*. 2011, S. 55–60. DOI: 10.1109/SGMS.2011.6089027 (zitiert auf den Seiten 48, 59, 172).

[Sch+19] G. Schweiger, C. Gomes, G. Engel et al. “An empirical survey on co-simulation: Promising standards, challenges and research needs”. In: *Simulation Modelling Practice and Theory* 95 (2019), S. 148–163. DOI: https://doi.org/10.1016/j.simpat.2019.05.001 (zitiert auf Seite 63).

[Shu+18a] Dewu Shu, Xiaorong Xie, Qirong Jiang, Gaopeng Guo und Ke Wang. “A Multirate EMT Co-Simulation of Large AC and MMC-Based MTDC Systems”. In: *IEEE Transactions on Power Systems* 33.2 (2018), S. 1252–1263. DOI: 10.1109/TPWRS.2017.2734690 (zitiert auf Seite 9).

[Shu+22] Dharma Shukla, Muthian Sivathanu, Srinidhi Viswanatha et al. *Singularity: Planet-Scale, Preemptive and Elastic Scheduling of AI Workloads*. 2022. DOI: 10.48550/ARXIV.2202.07848 (zitiert auf Seite 45).

[Shu+18b] Chong Shum, Wing-Hong Lau, Tian Mao et al. “Co-Simulation of Distributed Smart Grid Software Using Direct-Execution Simulation”. In: *IEEE Access* 6 (2018), S. 20531–20544. DOI: 10.1109/ACCESS.2018.2824341 (zitiert auf Seite 56).

[Sia14] Pierluigi Siano. “Demand response and smart grids—A survey”. In: *Renewable and Sustainable Energy Reviews* 30.C (2014), S. 461–478. URL: https://EconPapers.repec.org/RePEc:eee:rensus:v:30:y:2014:i:c:p:461-478 (zitiert auf Seite 6).

[SP17] Vindeep Singh und Sateesh Kumar Peddoju. “Container-based microservice architecture for cloud applications”. In: Mai 2017, S. 847–852. DOI: 10.1109/CCAA.2017.8229914 (zitiert auf Seite 26).

[Sol+07] Stephen Soltesz, Herbert Pötzl, Marc Fiuczynski, Andy Bavier und Larry Peterson. “Container-basedOperatingSystemVirtualization: A Scalable, High-performance Alternative to Hypervisors”. In: Bd. 41. Jan. 2007, S. 275–287 (zitiert auf Seite 27).

[Son+07] Heng-Jie Song, Zhi-Qi Shen, Chun-Yan Miao, Ah-Hwee Tan und Guo-Peng Zhao. “The Multi-Agent Data Collection in HLA-Based Simulation System”. In: *21st International Workshop on Principles of Advanced and Distributed Simulation (PADS’07)*. 2007, S. 61–69. DOI: 10.1109/PADS.2007.30 (zitiert auf Seite 55).

[Sta95] Robert Stake. *The art of Case Study Research*. Bd. 80. Apr. 1995. DOI: 10.2307/329758 (zitiert auf Seite 138).

[Ste+18] Cornelius Steinbrink, A.A. Meer, Milos Cvetkovic et al. “Smart grid co-simulation with MOSAIK and HLA: a comparison study”. In: *Computer Science - Research and Development* 33 (Feb. 2018). DOI: 10.1007/s00450-017-0379-y (zitiert auf den Seiten 20, 49, 51, 53, 55).

[SGZ19] Hao Sun, Yingqing Guo und Wanli Zhao. “Co-simulation of Aircraft Engine Fuel and Oil System”. In: *2019 IEEE 10th International Conference on Mechanical and Aerospace Engineering (ICMAE)*. 2019, S. 404–408. DOI: 10.1109/ICMAE.2019.8880958 (zitiert auf Seite 56).

[Sun+14] Xinwei Sun, Ying Chen, Jiatai Liu und Shaowei Huang. “A co-simulation platform for smart grid considering interaction between information and power systems”. In: *ISGT 2014*. 2014, S. 1–6. DOI: 10.1109/ISGT.2014.6816423 (zitiert auf Seite 9).

[Sys22] ©2020 Dassault Systèmes. *Simulate Functional Mockup Units (FMUs) in Python*. 2022. URL: https://github.com/CATIA-Systems/FMPy (besucht am 12. Mai 2022) (zitiert auf Seite 80).

[TMA12] Yingying Tang, Xiaolin Mao und Raja Ayyanar. “Distribution system modeling using CYMDIST for study of high penetration of distributed solar photovoltaics”. In: Sep. 2012, S. 1–6. DOI: 10.1109/NAPS.2012.6336408 (zitiert auf Seite 42).

[Tea22] Apache NiFi Team. *NiFi Developer’s Guide*. 2022. URL: https://nifi.apache.org/developer-guide.html (besucht am 6. Mai 2022) (zitiert auf Seite 82).

[TAV21] Teodor-Dorin Tripon, Gianina Adela Gabor und Elisa Valentina Moisi. “Angular and Svelte Frameworks: a Comparative Analysis”. In: *2021 16th International Conference on Engineering of Modern Electric Systems (EMES)*. 2021, S. 1–4. DOI: 10.1109/EMES52337.2021.9484119 (zitiert auf Seite 67).

[Tur14] J. Turnbull. *The Docker Book: Containerization Is the New Virtualization*. James Turnbull, 2014. URL: https://books.google.de/books?id=4xQKBAAAQBAJ (zitiert auf den Seiten 27, 28).

[UPA19] Luis F. Ugarte, Fransk A. Puma und Madson C. de Almeida. “Co-Simulation Architecture: A Tool to Enable the State Estimator Application in Smart Grid Environment”. In: *2019 International Conference on Smart Energy Systems and Technologies (SEST)*. 2019, S. 1–6. DOI: 10.1109/SEST.2019.8849025 (zitiert auf Seite 56).

[Van+12] Herman Van der Auweraer, J. Anthonis, Stijn Bruyne und Jan Leuridan. “Virtual engineering at work: The challenges for designing mechatronic products”. In: *Engineering with Computers* 29 (Juli 2012). DOI: 10.1007/s00366-012-0286-6 (zitiert auf Seite 64).

[Vil+16] Mario Villamizar, Oscar Garcés, Lina Ochoa et al. “Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures”. In: *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. 2016, S. 179–182. DOI: 10.1109/CCGrid.2016.37 (zitiert auf Seite 26).

[VMW23] VMWare. *RabbitMQ*. 2023. URL: https://www.rabbitmq.com/ (besucht am 16. Feb. 2023) (zitiert auf Seite 30).

[w3s22] w3sDesign. *The GoF Design Patterns Reference*. 2022. URL: http://w3sdesign.com/?gr=s01&ugr=proble (besucht am 6. Mai 2022) (zitiert auf Seite 78).

[WSR17] Kunpeng Wang, Peer-Olaf Siebers und Darren Robinson. “Towards Generalized Co-simulation of Urban Energy Systems”. In: *Procedia Engineering* 198 (Dez. 2017), S. 366–374. DOI: 10.1016/j.proeng.2017.07.092 (zitiert auf Seite 169).

[Wan+15] Yanbo Wang, Ke Li, Haiping Zhou et al. “Dynamic analysis and co-simulation ADAMS-SIMULINK for a space manipulator joint”. In: *2015 International Conference on Fluid Power and Mechatronics (FPM)*. 2015, S. 984–989. DOI: 10.1109/FPM.2015.7337258 (zitiert auf Seite 56).

[Wel+18] Lara Welder, D.Severin Ryberg, Leander Kotzur et al. “Spatio-temporal optimization of a future energy system for power-to-hydrogen applications in Germany”. In: *Energy* 158 (2018), S. 1130–1149. DOI: https://doi.org/10.1016/j.energy.2018.05.059 (zitiert auf den Seiten 141, 157).

[Wid+15] Edmund Widl, Wolfgang Müller, Daniele Basciotti et al. “Simulation of multi-domain energy systems based on the functional mock-up interface specification”. In: *2015 International Symposium on Smart Electric Distribution Systems and Technologies (EDST)*. 2015, S. 510–515. DOI: 10.1109/SEDST.2015.7315261 (zitiert auf Seite 56).

[Wik22a] Wikipedia. *Co-simulation*. 2022. URL: https://en.wikipedia.org/wiki/Co-simulation (besucht am 6. Mai 2022) (zitiert auf Seite 63).

[Wik22b] Wikipedia. *General Algebraic Modeling System*. 2022. URL: https://en.wikipedia.org/wiki/General_Algebraic_Modeling_System (besucht am 12. Mai 2022) (zitiert auf Seite 80).

[XCW11] Qun Xiang, Gang Chen und Yao Wang. “Distributed simulation based on web enabling HLA”. In: *2011 2nd International Conference on Artificial Intelligence, Management Science and Electronic Commerce (AIMSEC)*. 2011, S. 2062–2064. DOI: 10.1109/AIMSEC.2011.6011137 (zitiert auf Seite 55).

[Yin09] Robert Yin. *Case study research: design and methods*. Jan. 2009, S. 219 (zitiert auf Seite 138).

[Zha+10] Zhi Zhang, Zhonghai Lu, Qiang Chen, Xiaolang Yan und Li-Rong Zheng. “COSMO: CO-Simulation with MATLAB and OMNeT++ for Indoor Wireless Networks”. In: *2010 IEEE Global Telecommunications Conference GLOBECOM 2010*. 2010, S. 1–6. DOI: 10.1109/GLOCOM.2010.5683583 (zitiert auf Seite 56).

Abbildungsverzeichnis

1.1 Entwicklungszyklus von Workflows im Kontext von Computational Science [Liu+19; Ove09] . . . 3
1.2 Smart Grid [Pau+14; Col16] . . . 7

2.1 Simulationskategorien [Pal+17a; HP17] . . . 16
2.2 Kommunikationsreihenfolge – nicht iterativ [Pal+17a] . . . 19
2.3 Kommunikationsreihenfolge – iterativ, parallele Sequenz [Pal+17a] . . . 21
2.4 Synchronisation durch feste Zeitpunkte [Pal+17a] . . . 22
2.5 Ereignisgetriebene Synchronisation [Pal+17a] . . . 23
2.6 Primary-Secondary-Synchronisation [Pal+17a] . . . 24
2.7 Stand der Technik zur Implementierung des PROOF-Konzepts . . . 26
2.8 Docker Container [Tur14] . . . 28

3.1 Architektur von OOCoSim [Kim+18] . . . 39
3.2 Architektur von *FNCS* [Hua+17] . . . 41
3.3 Architektur von *HELICS* [Pal+17b] . . . 43
3.4 Die Lese- und Schreiboperationen zwischen *Modelica* und *ns-2* [LA11] 46
3.5 Einordnung von PowerNet . . . 47
3.6 Architektur von Mosaik [Ste+18] . . . 49
3.7 Einordnung von *Mosaik* . . . 50
3.8 Architektur von *HLA* [Ste+18] . . . 53
3.9 Einordnung von *HLA* . . . 54

4.1 Co-Simulation im Bereich Smart Grids . . . 64
4.2 Co-Simulations-Architektur . . . 66
4.3 Web-basierte grafische Oberfläche . . . 67
4.4 Co-Simulation mit drei Simulatoren . . . 69
4.5 Kommunikation zwischen einem Simulator und Apache Kafka . . . 70
4.6 PROOF-Architektur . . . 73
4.7 Die konzeptionelle Microservices-Architektur von PROOF . . . 75
4.8 Integration einer Applikation in Docker-Container . . . 77
4.9 PROOF-Wrapper . . . 78
4.10 Ablaufsteuerung vom PROOF-Wrapper . . . 79
4.11 Hinzufügen eines eigenen NiFi-Prozessors zur Prozess-Bibliothek . . . 84
4.12 Apache NiFi Weboberfläche . . . 88

4.13 Prozess-Bibliothek 89
4.14 Ablaufplan Konfiguration 90
4.15 Konfiguration der Parameter 91
4.16 Operation zum Erstellen und Importieren einer XML-Workflow-Vorlage 92

5.1 Datenfluss eines Workflows für zwei NiFi-Prozessoren 99
5.2 Konfigurationsdialogfenster 100
5.3 Parameterübertragung 101
5.4 Kopplung von zwei Prozessoren 102
5.5 Datenübertragung zwischen Framework-Prozessen 103
5.6 Austausch großer Datenmengen im Volume 104
5.7 Ablaufdiagramm zur Datenverarbeitung innerhalb eines Framework-Prozesses 105
5.8 Bereitstellung von Applikationen im Volume 107
5.9 Konfigurationsdialogfenster für Eingabe des Applikationsnamens . . . 107

6.1 Konzept der Parallelisierung und Koordination 117
6.2 Parameter „Concurrent Tasks“ zur Definition der Anzahl der Container 118
6.3 Merge-Service zur Synchronisierung mehrerer Datenflüsse 121
6.4 Ablaufdiagramm vom Merge-Service 121
6.5 Strategie 1: FIFO & die Anzahl der Elemente von den Datenflüssen A, B und C ist gleich 1 122
6.6 Strategie 2: FIFO & die Anzahl der Elemente von den Datenflüssen A, B und C ist ungleich 123
6.7 Strategie 3: Basierend auf einem Attribut, z.B. Stadt & die Anzahl der Elemente von den Datenflüssen A, B und C ist ungleich 124
6.8 Strategie 4: Basierend auf mehreren Attributen, z.B. Stadt, Nationalität, Fachrichtung & die Anzahl der Elemente von den Datenflüssen A, B und C ist ungleich 124
6.9 Strategie 5: Basierend auf mehreren Attributen, z.B. Stadt, Nationalität, Fachrichtung & verschiedene Kombinationen von Elementen aus Datenflüssen 125
6.10 Deskriptive Sprache zur Definition der Strategien für den Merge-Service126
6.11 Definition der Strategie 1 127
6.12 Definition der Strategie 2 128
6.13 Definition der Strategie 3 129
6.14 Definition der Strategie 4 130
6.15 Definition der Strategie 5 131
6.16 Primary-Secondary-Architektur 133
6.17 Schedule-Config 133

7.1 Übersicht vom ESM [Liu+18] 142

7.2 Übersicht der Methode. PPF: **P**robabilistic **P**ower **F**low; PCE: **P**olynomial **C**haos **E**xpansion [Gon+21] . . . 146
7.3 Übersicht des EH-Systems [Pop+21] . . . 149
7.4 Aufbau des *EH* . . . 151
7.5 Use-Case-Workflow zur Gestaltung eines überregionalen Energiesystems 157
7.6 Speichernutzung ohne PROOF . . . 158
7.7 Speichernutzung mit PROOF . . . 158
7.8 Verhältnis von Overhead zu der gesamten Ausführungszeit . . . 160
7.9 Ausführungszeit für mehrere parallele Tasks . . . 161

A.1 Simulation einer Windkraftanlage als Beispiel einer Co-Simulation . . 208
A.2 Kompilieren der Matlab-Applikation . . . 209
A.3 Laden der Main-Datei der Matlab-Applikation . . . 209
A.4 Verpacken der Matlab-Applikation . . . 210
A.5 Integration der Matlab-Applikation im Docker-Image . . . 210
A.6 Ein aufgebauter Workflow auf der Oberfläche . . . 216
A.7 Integration des ESM in PROOF [Liu+18] . . . 225
A.8 Implementierung des Frameworks in PROOF: Sowohl die Datenerfassungs- als auch die Visualisierungsschritte werden durch die Blöcke dargestellt, die eine Verbindung zu easimov herstellen, um Daten zu extrahieren oder Daten für die Visualisierung zu speichern. [Gon+21] . . . 227
A.9 Implementierung des EH in PROOF [Pop+21] . . . 229

Tabellenverzeichnis

2.1 Vergleich von monolithischen und Microservices-Architekturen (Erweiterung von [Bak17] . . . 27

3.1 Anforderungskatalog für die Bewertung relevanter Arbeiten sowie der eigenen Beiträge . . . 38
3.2 Liste der Co-Simulations-Frameworks mit drahtloser Netzwerktechnologie 57
3.3 Übersicht der Anforderungen, die durch verwandte Arbeiten erfüllt werden:
○-nicht; ◐-teilweise; ●-vollständig . . . 59

5.1 Vergleich verschiedener Kommunikationsschnittstellen basierend auf Anforderungen A3.1-A3.4: ○-nicht erfüllt; ◐-teilweise erfüllt; ●-vollständig erfüllt. . . . 97

7.1 Standard-Prozesse und -Services in der Prozess-Bibliothek von PROOF 153
7.2 Vergleich der Ausführungszeit mit und ohne PROOF . . . 159
7.3 Bewertung dieser Arbeit im Vergleich zu bestehenden Arbeiten basierend auf den neun Anforderungen: ○-nicht erfüllt; ◐-teilweise erfüllt; ●-vollständig erfüllt. . . . 172
7.4 Verknüpfung der Anforderungen mit den Beiträgen dieser Arbeit . . . 173

Auflistungsverzeichnis

3.1 Beispiel von Snakefile . . . 44
3.2 Beispiel von config.yaml . . . 45

4.1 Dockerfile zur Integration eines Python-Modells . . . 81
4.2 Befehl zum Erstellen eines Docker-Images . . . 81
4.3 Die init-Funktion eines NiFi-Prozessors . . . 85
4.4 Die onScheduled-Funktion . . . 85
4.5 Ein Task zur Aktivierung eines Prozessors . . . 85
4.6 Die onTrigger-Funktion . . . 86
4.7 Die onRemoved-Funktion . . . 87

5.1 Parameter definieren . . . 100
5.2 MQTT-Service . . . 108
5.3 Zeitreihendaten-Service . . . 109
5.4 Excel-Operation . . . 110
5.5 CSV-Operation . . . 111
5.6 PKL-Operation . . . 111
5.7 PDF-Operation . . . 112

A.1 Dockerfile zur Integration eines Python-Modells . . . 211
A.2 Die init-Funktion eines NiFi-Prozessors . . . 212
A.3 Die onScheduled-Funktion . . . 212
A.4 Die onTrigger-Funktion . . . 213
A.5 Die onRemoved-Funktion . . . 214
A.6 XML-Vorlage eines Workflows . . . 217
A.7 Parameter definieren . . . 219
A.8 MQTT-Service . . . 220
A.9 Excel-Operation . . . 221
A.10 CSV-Operation . . . 221
A.11 PKL-Operation . . . 222
A.12 PDF-Operation . . . 222
A.13 eASiMOV-Service . . . 223

Anhang A

A.1 Simulation einer Windkraftanlage

Um die Praxistauglichkeit der Co-Simulationsplattform zu bewerten, wird eine Co-Simulation einer Windkraftanlage als ein praktisches Modellierungsbeispiel für die Verwendung der Co-Simulationsplattform implementiert. Durch diesen konkreten Anwendungsfall wird die Nutzbarkeit und Erweiterbarkeit der Co-Simulationsplattform demonstriert. Die entsprechende Leistungsausgabe wird in einem Knoten simuliert und ihre Eingabedaten (z.B. Windgeschwindigkeit und Temperatur) werden durch einen zweiten Wetterdatenquellen-Knoten demonstriert. Abb. A.1 illustriert das Konzept der Co-Simulation, die aus zwei Simulatoren besteht:

- Ein Datenquellenknoten generiert die Eingabedaten der Windkraftanlage wie Druck, Windgeschwindigkeit, Temperatur und Rauhigkeitslänge. Dann stellt der Knoten dem Windkraftanlagenmodell alle Daten als ein anderer Simulationsknoten über eine Apache Kafka-Warteschlange mit einem Topic namens „WeatherInformation" bereit.
- Ein Python-Simulator, der die von der Python Software Foundation (US) bereitgestellte Bibliothek *windpowerlib* [Com22b] verwendet, wird zum Aufbau verschiedener praktischer Windkraftmodelle integriert. Anhand der Wettereingangsdaten wird die Berechnung von Zeitreihen der Stromerzeugungsleistung der Windkraftanlage ausgeführt.

Bevor eine Simulation gestartet wird, wird zunächst der Windkraftanlagensimulationsknoten parametrisiert, indem die Werte einer Liste von statischen Konfigurationsparametern eingestellt werden, z.B. die Höhe der Windkraftanlage und der Durchmesser ihres Rotors. In ähnlicher Weise ist der Wetterdatenquellenknoten so konfiguriert, dass er Wetterdaten gemäß einem bestimmten Zeitschema (z.B. alle 5 Sekunden) sendet. In diesem Beispiel liest der Wetterdatenknoten Wetterinformationen, die bereits den Druck, die Windgeschwindigkeit, die Temperatur und die Rauhigkeitslänge enthalten, aus einer CSV-Datei und leitet sie als ein JSON-Objekt an das Apache Kafka-Topic „WeatherInformation" des Co-Simulationsmodells weiter. Unter Nutzung der Daten aus dem Topic „WeatherInformation" berechnet das Windturbinenmodell die elektrische Leistung und sendet sie an eine andere Kafka-Warteschlange mit einem Topic namens „WindTurbinePowerOutput". Daraus werden die Leistungsdaten extrahiert und in einer Datenbank abgespeichert, z.B. zur Verwendung in einer späteren Analyse oder zur Visualisierung. Schließlich kann

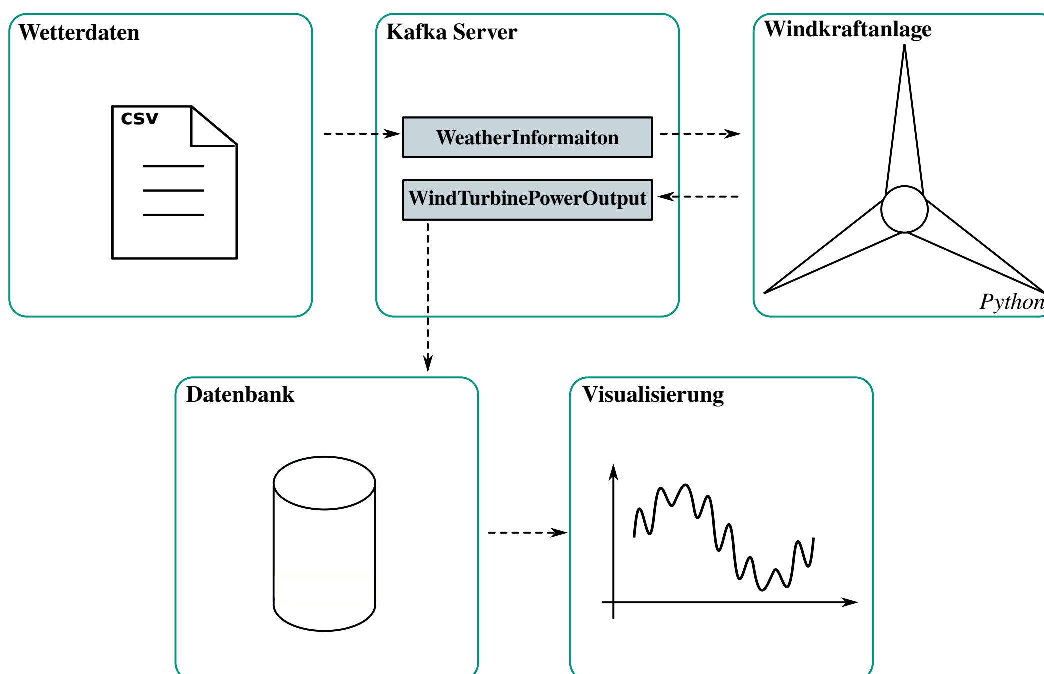

Abb. A.1.: Simulation einer Windkraftanlage als Beispiel einer Co-Simulation

der Co-Simulation eine Visualisierungsanwendung hinzugefügt werden, die die Leistungsausgabewerte auf der Benutzeroberfläche der Co-Simulationsoberfläche in Form einer geeigneten Visualisierung (z.B. eine Linienkurve) anzeigt.

A.2 Erstellung von ausführbaren Matlab-Applikationen

In diesem Abschnitt werden Schritt für Schritt beschrieben, wie ein Kompilationsprozess durchgeführt wird, um eine Matlab-Applikation als eine eigenständige ausführbare Datei zu erstellen und zu exportieren.

a. **Kompilieren**
Wie in A.2 markiert, stellt Matlab unter der Menüleiste *APPS* die Funktion ***Application Compiler*** zur Verfügung, um eine Matlab-Applikation als eine eigenständige Datei zu kompilieren. Nach dem Anklicken der Funktion beginnt ein Kompilations-Prozess.

b. **Laden**
Als zweiter Schritt wird die **Main**-Datei der Matlab-Applikation ausgewählt und geladen (siehe Abb. A.3).

c. **Verpacken**
Wie in Abb. A.4 unten zu sehen ist, werden dann die benötigten Dateien zur Ausführung der Matlab-Applikation automatisch eingefügt. Durch die

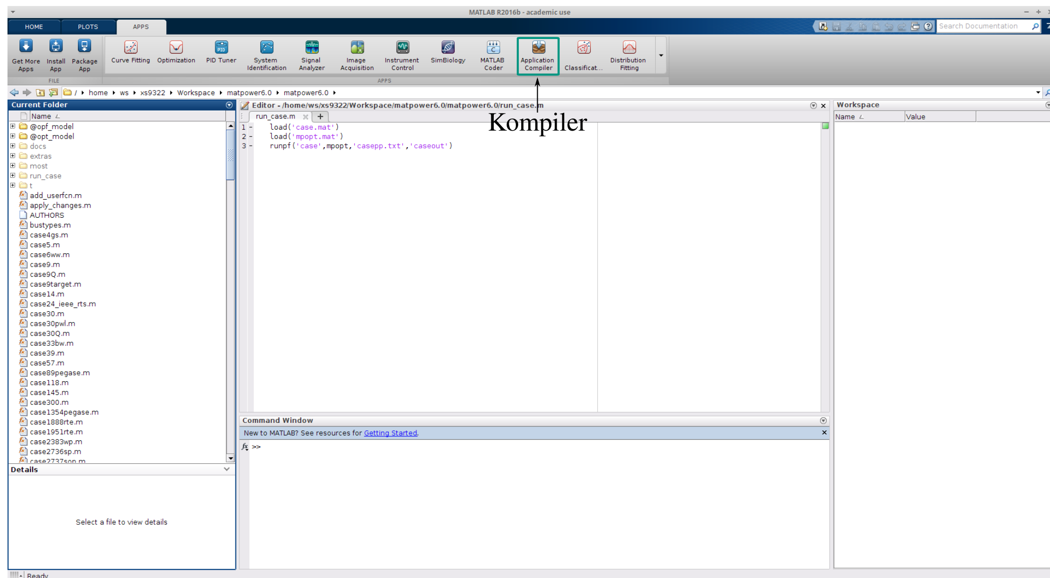

Abb. A.2.: Kompilieren der Matlab-Applikation

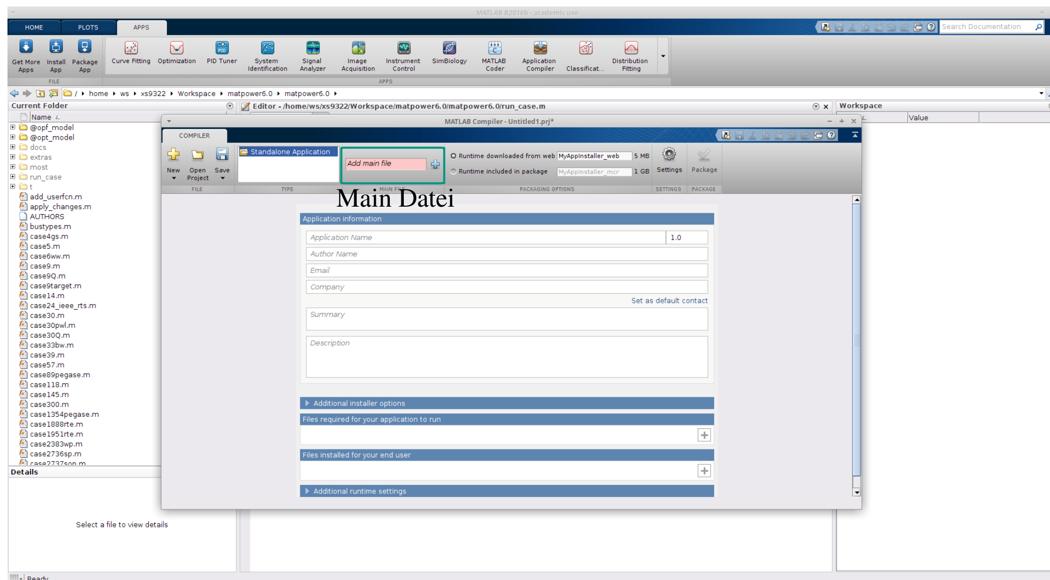

Abb. A.3.: Laden der Main-Datei der Matlab-Applikation

Funktion *Package* (siehe Abb. A.4 oben rechts) wird das Matlab-Projekt als eine ausführbare Datei verpackt.

d. **Integrieren**
Die ausgegebene Datei ***run_case*** (siehe in Abb. A.5) kann als eine eigenständige Applikation in einem Docker-Image integriert, dann in Docker-Containern ausgeführt werden. Die Details bezüglich der Integration werden im nächsten Abschnitt beschrieben.

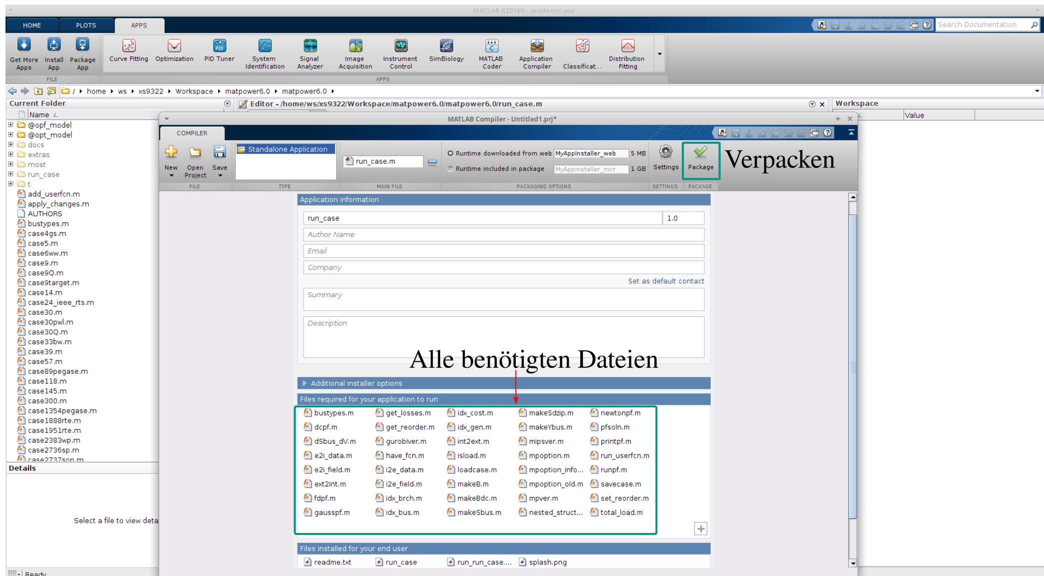

Abb. A.4.: Verpacken der Matlab-Applikation

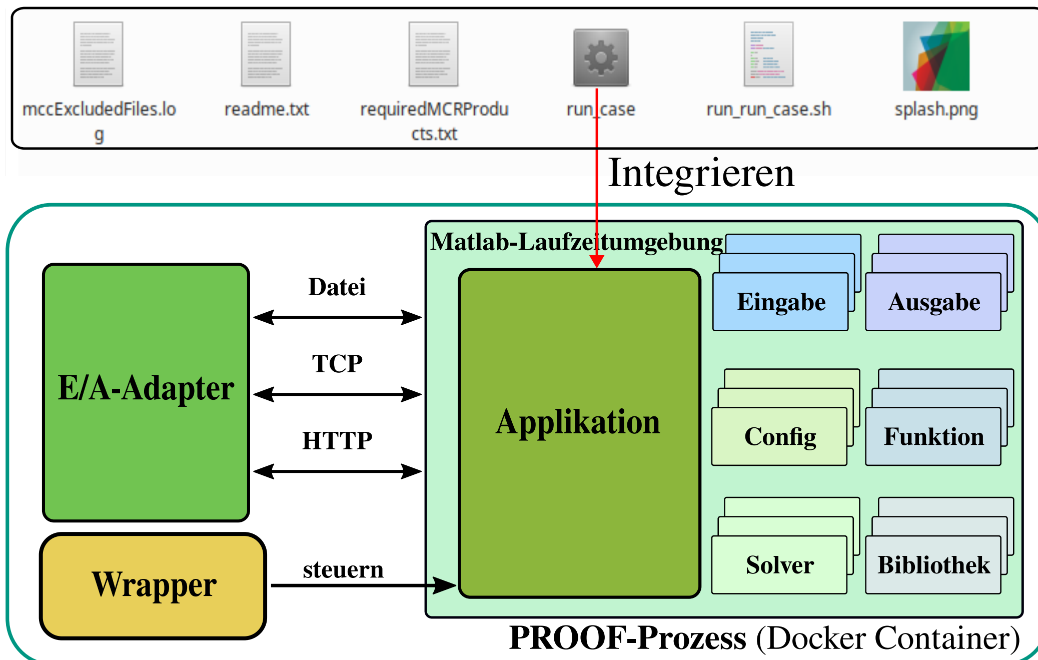

Abb. A.5.: Integration der Matlab-Applikation im Docker-Image

A.3 Dockerfile

In Auflistung A.1 wird ein beispielhaftes vollständiges Dockerfile dargestellt. Es umfasst dies im Detail die Anweisungen zur Installation aller Programmbibliotheken (Zeile 3-15) und anderer Softwarekomponenten (Zeile 19-24), von denen die ausführbare Applikation abhängt. Zudem werden einige benötigte Ordner angelegt

(Zeile 28-32), die für Eingabe, Zwischenergebnisse oder Ausgabe benutzt werden. Anschließend wird die Quellcodedatei der ausführbaren Applikation (Zeile 34) geladen. Dann werden die Adapter zur Interaktion mit der Applikation ausgewählt (Zeile 37-40), die selbst mit dem Framework angeboten werden. Zuletzt werden die Python-Skripte *RedisConsumer.py* (Zeile 42) und *RedisPublisher.py* (Zeile 44) zum Datenaustausch mit dem Message-Server *Redis* und *Wrapper.py* (Zeile 47) zur Steuerung der Applikation beigefügt. Das Kommunikations- und Steuerungssetup ist für alle ausführbaren Dateien identisch und muss nicht von Benutzern angepasst werden.

```
1  FROM ubuntu:20.04
2
3  RUN apt-get clean && apt-get update
4  RUN apt-get install -y software-properties-common
5  RUN add-apt-repository ppa:ubuntugis/ppa
6  RUN apt-get update
7  RUN apt-get install -y python3
8  RUN apt-get install -y python3-pip
9  RUN pip3 install --upgrade pip
10 RUN pip3 install numpy
11 RUN pip3 install pandas
12 RUN pip3 install matplotlib
13 RUN pip3 install scipy
14 RUN pip3 install descartes
15 RUN pip3 install redis
16
17 ADD source /source
18
19 WORKDIR /source/geokit
20 RUN pip3 install -e .
21 WORKDIR /source/glaes
22 RUN pip3 install -e .
23
24 RUN apt-get remove -y software-properties-common
25
26 WORKDIR /
27
28 RUN mkdir /outputs
29 RUN chmod -R 777 /outputs
30
31 RUN mkdir /datadir
32 RUN chmod -R 777 /datadir
33
34 ADD application.py /application.py
35 RUN ["chmod", "+x", "/application.py"]
36
37 ADD FileAdapterInput.py /FileAdapterInput.py
38 RUN ["chmod", "+x", "/FileAdapterInput.py"]
39 ADD FileAdapterOutput.py /FileAdapterOutput.py
40 RUN ["chmod", "+x", "/FileAdapterOutput.py"]
41
42 ADD RedisConsumer.py /RedisConsumer.py
43 RUN ["chmod", "+x", "/RedisConsumer.py"]
44 ADD RedisPublisher.py /RedisPublisher.py
45 RUN ["chmod", "+x", "/RedisPublisher.py"]
46
47 ADD Wrapper.py /Wrapper.py
48 RUN ["chmod", "+x", "/Wrapper.py"]
49
50 ENTRYPOINT ["python3", "/Wrapper.py"]
```

Auflistung A.1: Dockerfile zur Integration eines Python-Modells

A.4 Implementierung eines NiFi-Prozessors

Um einen neuen NiFi-Prozessor zu implementieren werden das Vorgehen (Initialisieren, Triggern und Löschen) und die entsprechenden Methoden definiert, um den Lebenszyklus eines NiFi-Prozessors zu beschreiben:

A.4.1 Initialisieren

```
1   public static final PropertyDescriptor INPUT = new PropertyDescriptor
2               .Builder().name("Input")
3               .description("The input name")
4               .required(true)
5               .addValidator(StandardValidators.NON_EMPTY_VALIDATOR)
6               .defaultValue("input.json")
7               .build();
8
9   public static final Relationship SUCCESS = new Relationship.Builder()
10              .name("success")
11              .description("This relationship is used when the process in the docker container was
                    successful")
12              .build();
13
14  @Override
15  public void init(final ProcessorInitializationContext context) {
16      final List<PropertyDescriptor> descriptors = new ArrayList<PropertyDescriptor>();
17      descriptors.add(INPUT);
18      this.descriptors = Collections.unmodifiableList(descriptors);
19      final Set<Relationship> relationships = new HashSet<Relationship>();
20      relationships.add(SUCCESS);
21      this.relationships = Collections.unmodifiableSet(relationships);
22  }
```

Auflistung A.2: Die init-Funktion eines NiFi-Prozessors

```
1   @OnScheduled
2   public void onScheduled(final ProcessContext context) {
3       Task task = new Task(redisHost, redisPort);
4       int replicas = context.getMaxConcurrentTasks();
5
6       JsonObject taskJsonObject = ProcessorTask.prepareTask(task.getId(),
            task.getInputContainerTopicName(), task.getOutputContainerTopicName(), get_IMAGE_NAME(),
            replicas, folder);
7       taskJsonObject.addProperty("operation", "create");
8
9       // send metadata to proof main topic
10      RedisPublisher redisPublisher = new RedisPublisher(PROOF_MAIN_TASK_TOPIC, redisHost);
11      redisPublisher.publish(taskJsonObject.toString());
12      logger.info(taskJsonObject.toString());
13  }
```

Auflistung A.3: Die onScheduled-Funktion

A.4.2 Triggern

Nach der Initialisierung wird ein Prozessor getriggert, wenn für den Prozessor Arbeit vorhanden ist. Dabei wird die Methode *onTrigger* aufgerufen, die die Logik zur Datenverarbeitung in Auflistung A.4 definiert:

- Zeile 4 bis 17
  Die Eingabedaten werden zuerst als ein *FlowFile* aus der *ProcessSession* aufgenommen und anschließend im Parameter *value* gespeichert. „final“ braucht man hier nur, damit der Compiler den Parameter „value“ in der anonymen Klasse InputStreamCallback verwenden kann.
- Zeile 19 bis 21
  Danach werden die Eingabedaten mittels eines *RedisPublisher* über Redis an den entsprechenden Docker-Container einschließlich des Modells übermittelt.
- Zeile 23 bis 36
  Anschließend wird ein *JedisSubscriber* in der *onTrigger*-Funktion erstellt, der das Output-Topic abonniert und abwartet, bis die Ausgabe vom Modell an das Topic zurückgesendet wird.
- Zeile 38 bis 47
  Die Ausgabedaten werden zuletzt als ein *FlowFile* an den nächsten verknüpften NiFi-Prozessor transferiert.

```
1  @Override
2  public void onTrigger(final ProcessContext context, final ProcessSession session) throws ProcessException
       {
3      // get input stream and set the value
4      FlowFile flowFile = session.get();
5      final AtomicReference<String> value = new AtomicReference<>();
6      session.read(flowFile, new InputStreamCallback() {
7          @Override
8          public void process(InputStream in) throws IOException {
9              try {
10                 String data = IOUtils.toString(in);
11                 value.set(data);
12             } catch (Exception ex) {
13                 ex.printStackTrace();
14                 getLogger().error("Failed to read string.");
15             }
16         }
17     });
18
19     RedisPublisher redisPublisher = new RedisPublisher(topicNameForContainer, redisHost);
20     String messageToModel = collectInput(context, value.get());
21     redisPublisher.publish(messageToModel);
22
23     Jedis jSubscriber = new Jedis(redisHost);
24         int finalNumberOfAvailableContainer = numberOfAvailableContainer;
25         jSubscriber.subscribe(new JedisPubSub() {
26             @Override
27             public void onMessage(String channel, String message) {
28                 // handle message
29                 super.onMessage(channel, message);
30                 returnValue = message;
31                 unsubscribe();
32             }
33         }, task.getOutputContainerTopicName() + "-" + numberOfAvailableContainer);
34         jSubscriber.set(task.getId() + "-" + finalNumberOfAvailableContainer + "-processor-ready", 0 +
               "");
35         jSubscriber.close();
36
37     flowFile = session.write(flowFile, new OutputStreamCallback() {
38         @Override
39         public void process(OutputStream out) throws IOException {
40             out.write(message.getBytes());
41         }
42     });
43
44     session.transfer(flowFile, SUCCESS);
45 }
```

Auflistung A.4: Die onTrigger-Funktion

A.4.3 Löschen

Falls der NiFi-Prozessor nicht mehr benötigt und seine Ausführung nicht mehr geplant ist, wird die *onRemoved*-Methode aufgerufen, um den NiFi-Prozessor und seinen Docker-Container bzw. den PROOF-Prozess komplett zu entfernen. Wie in Auflistung 4.7 zu sehen ist, wird ein neuer Task erstellt und dem Parameter *operation* der Wert *delete* übergeben (Zeilen 4 und 5). Der Task wird dann an das Redis-Topic *PROOF_MAIN_TASK_TOPIC* gesendet (Zeilen 6 und 7), um das Löschen des PROOF-Prozesses im Hintergrund durchzuführen.

```
1  @OnRemoved
2  public void onRemoved(final ProcessContext context) {
3      int replicas = context.getMaxConcurrentTasks();
4      JsonObject taskJsonObject = ProcessorTask.prepareTask(task.getId(),
           task.getInputContainerTopicName(), task.getOutputContainerTopicName(), get_IMAGE_NAME(),
           replicas);
5      taskJsonObject.addProperty("operation", "delete");
6      RedisPublisher redisPublisher = new RedisPublisher(PROOF_MAIN_TASK_TOPIC, redisHost);
7      redisPublisher.publish(taskJsonObject.toString());
8  }
```

Auflistung A.5: Die onRemoved-Funktion

A.5 Beispielhafter Workflow

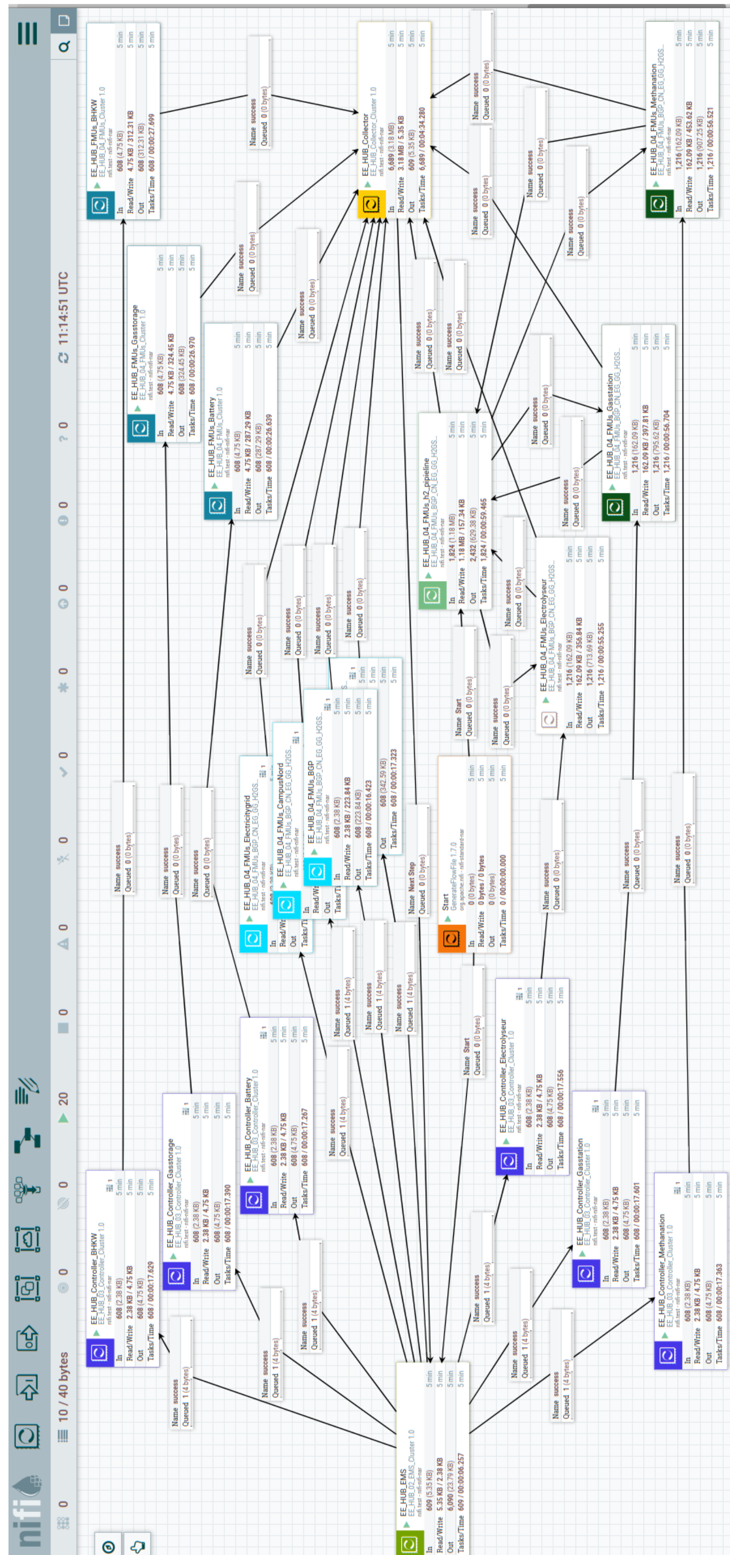

Abb. A.6.: Ein aufgebauter Workflow auf der Oberfläche

A.6 XML-Vorlage

Um alle Informationen bezüglich eines Workflows zu speichern, wird eine Vorlage so wie in Auflistung A.6 strukturiert:

- Zeilen 3 und 4
  Die Stammdaten des Workflows, z.B. die ID und der Name.
- Zeile 5 bis 164
  Die Beschreibung aller Prozessoren, z.B. Stammdaten (ID, Gruppe, Position, Name, Version, Typ), Konfigurationsdaten (Parameter, Attribute, Ablaufstrategie), Verbindungsdaten (*Relationships*) usw.

```
<template encoding-version="1.2">
  <description/>
  <groupId>4e04b3ef-0181-1000-0150-18c2ec31d934</groupId>
  <name>1</name>
  <snippet>
    <processors>
      <id>a805e098-50bc-3fe8-0000-000000000000</id>
      <parentGroupId>3c2d548a-1814-302e-0000-000000000000</parentGroupId>
      <position>
        <x>0.0</x>
        <y>0.0</y>
      </position>
      <bundle>
        <artifact>nifi-nifi-nar</artifact>
        <group>nifi.test</group>
        <version>1.0</version>
      </bundle>
      <config>
        <bulletinLevel>WARN</bulletinLevel>
        <comments/>
        <concurrentlySchedulableTaskCount>1</concurrentlySchedulableTaskCount>
        <descriptors>
          <entry>
            <key>Input Directory</key>
            <value>
              <name>Input Directory</name>
            </value>
          </entry>
          <entry>
            <key>Input File(s)</key>
            <value>
              <name>Input File(s)</name>
            </value>
          </entry>
          <entry>
            <key>Output Directory</key>
            <value>
              <name>Output Directory</name>
            </value>
          </entry>
          <entry>
            <key>Output File(s)</key>
            <value>
              <name>Output File(s)</name>
            </value>
          </entry>
        </descriptors>
        <executionNode>ALL</executionNode>
        <lossTolerant>false</lossTolerant>
        <penaltyDuration>30 sec</penaltyDuration>
        <properties>
          <entry>
            <key>Input Directory</key>
            <value>/</value>
          </entry>
          <entry>
            <key>Input File(s)</key>
            <value>input_struct.json</value>
          </entry>
```

```
60              <entry>
61                <key>Output Directory</key>
62                <value>/</value>
63              </entry>
64              <entry>
65                <key>Output File(s)</key>
66                <value>forecast.json</value>
67              </entry>
68            </properties>
69            <runDurationMillis>0</runDurationMillis>
70            <schedulingPeriod>0 sec</schedulingPeriod>
71            <schedulingStrategy>TIMER_DRIVEN</schedulingStrategy>
72            <yieldDuration>1 sec</yieldDuration>
73          </config>
74          <executionNodeRestricted>false</executionNodeRestricted>
75          <name>Energylab_01_Forecast_Cluster</name>
76          <relationships>
77            <autoTerminate>false</autoTerminate>
78            <name>success</name>
79          </relationships>
80          <state>STOPPED</state>
81          <style/>
82          <type>edu.kit.iai.webis.Processors.energylab.model_01_forecast.Energylab_01_Forecast_Cluster</type>
83        </processors>
84        <processors>
85          <id>ceed8406-d031-3c78-0000-000000000000</id>
86          <parentGroupId>3c2d548a-1814-302e-0000-000000000000</parentGroupId>
87          <position>
88            <x>661.5581619972645</x>
89            <y>0.09021874569373267</y>
90          </position>
91          <bundle>
92            <artifact>nifi-nifi-nar</artifact>
93            <group>nifi.test</group>
94            <version>1.0</version>
95          </bundle>
96          <config>
97            <bulletinLevel>WARN</bulletinLevel>
98            <comments/>
99            <concurrentlySchedulableTaskCount>1</concurrentlySchedulableTaskCount>
100           <descriptors>
101             <entry>
102               <key>Input Directory</key>
103               <value>
104                 <name>Input Directory</name>
105               </value>
106             </entry>
107             <entry>
108               <key>Input File(s)</key>
109               <value>
110                 <name>Input File(s)</name>
111               </value>
112             </entry>
113             <entry>
114               <key>Output Directory</key>
115               <value>
116                 <name>Output Directory</name>
117               </value>
118             </entry>
119             <entry>
120               <key>Output File(s)</key>
121               <value>
122                 <name>Output File(s)</name>
123               </value>
124             </entry>
125           </descriptors>
126           <executionNode>ALL</executionNode>
127           <lossTolerant>false</lossTolerant>
128           <penaltyDuration>30 sec</penaltyDuration>
129           <properties>
130             <entry>
131               <key>Input Directory</key>
132               <value>/</value>
133             </entry>
134             <entry>
135               <key>Input File(s)</key>
136               <value>input.json</value>
137             </entry>
138             <entry>
139               <key>Output Directory</key>
140               <value>/</value>
141             </entry>
```

```
            <entry>
              <key>Output File(s)</key>
              <value>output.json</value>
            </entry>
          </properties>
          <runDurationMillis>0</runDurationMillis>
          <schedulingPeriod>0 sec</schedulingPeriod>
          <schedulingStrategy>TIMER_DRIVEN</schedulingStrategy>
          <yieldDuration>1 sec</yieldDuration>
        </config>
        <executionNodeRestricted>false</executionNodeRestricted>
        <name>EnergyLab_02_Quantile2Distribution_Cluster</name>
        <relationships>
          <autoTerminate>false</autoTerminate>
          <name>success</name>
        </relationships>
        <state>STOPPED</state>
        <style/>
        <type>edu.kit.iai.webis.Processors.energylab.model_02_quatile2distribution.
              EnergyLab_02_Quantile2Distribution_Cluster
        </type>
      </processors>
    </snippet>
    <timestamp>07/05/2022 13:27:34 UTC</timestamp>
  </template>
```

Auflistung A.6: XML-Vorlage eines Workflows

A.7 Parameter definieren

```
public static final PropertyDescriptor INPUT_FILE =
    new PropertyDescriptor
    .Builder().name("Intput File(s)")
    .description("The intput file name")
    .required(true)
    .addValidator(StandardValidators.NON_EMPTY_VALIDATOR)
    .build()}

public static final PropertyDescriptor OUTPUT_FILE = ...

@Override
public void init(final ProcessorInitializationContext context) {
    final List<PropertyDescriptor> descriptors = new ArrayList<PropertyDescriptor>();
    descriptors.add(INPUT_FILE);
    descriptors.add(OUTPUT_FILE);

    this.descriptors = Collections.unmodifiableList(descriptors);
}
```

Auflistung A.7: Parameter definieren

A.8 Konfigurierbare Services

A.8.1 MQTT-Service

Auflistung 5.2 stellt die Implementierung des *MqttServices* von PROOF dar. Damit kann ein MQTT-Subscriber zum Empfangen der externen Nachrichten (siehe Zeile 31 bis 36) und ein MQTT-Publisher zum Senden der Nachrichten nach außen (siehe Zeile 41 bis 44) aufgebaut werden. Dabei sind fünf grundlegende Parameter

bei der Initialisierung zu konfigurieren, nämlich *broker*, *port*, *username*, *pw* und *topic* (siehe Zeile 5). Wie der im vorherigen Abschnitt 5.3.2 vorgestellte Parameter *Application Name* können die fünf Parameter auf der Registerkarte *PROPERTIES* im Konfigurationsdialog eingegeben werden, um eine Instanz vom MqttService zu erstellen.

```
import ssl
import paho.mqtt.client as mqtt

class MqttService:
    def __init__(self, broker, port, username, pw, topic = ""):
        self.broker = broker #iai-broker: 141.52.45.44
        self.port = port # iai-port: 8883
        self.topic = topic
        self.username = username # "IAI"
        self.pw = pw # "vcRu1lfp3MfPbvCPHBYfBV6b5"

    def configure_client(self):
        self.client.tls_set(tls_version=ssl.PROTOCOL_TLSv1_2)
        self.client.tls_insecure_set(True)
        self.client.username_pw_set(self.username, self.pw)

    def on_connect(self, userdata, flags, rc):
        print("Connected with result code " + str(rc))
        # Subscribing in on_connect() means that if we lose the connection and reconnect then
            subscriptions will be renewed.
        self.client.subscribe(self.topic)

    def on_message(self, userdata, msg):
        print(msg.topic + " " + str(msg.payload))

    def subscriber_connect(self):
        self.client.connect(self.broker, self.port, 60)
        # Blocking call that processes network traffic, dispatches callbacks and handles reconnecting.
        # Other loop*() functions are available that give a threaded interface and a manual interface.
        self.client.loop_forever()

    def create_subscriber(self):
        self.client = mqtt.Client("Subscriber")
        self.client.on_connect = on_connect
        self.client.on_message = on_message
        configure_client()
        subscriber_connect()

    def publisher_connect(self):
        self.client.connect(self.broker, self.port, 60)

    def create_publisher(self):
        self.client = mqtt.Client("Publisher")
        configure_client()
        publisher_connect()

    def publish(self, msg):
        self.client.publish(self.topic, msg)

    def disconnect(self):
        self.client.disconnect()

    def stop_loop(self):
        self.client.loop_stop()
```

Auflistung A.8: MQTT-Service

A.8.2 Dateioperationen

Excel

```
import os
import sys

import openpyxl

class ExcelOperation:
    def __init__(self, original_file):
        self.excel_file = original_file

    # write data
    def write_data(self, data):

        wb_obj = openpyxl.load_workbook(self.excel_file)
        sheet_obj = wb_obj.active

        for cell in data:
            value = data[cell]
            sheet_obj[cell] = value
            print(cell + ":" + str(value) + "is written into cell " + cell)

        wb_obj.save(self.excel_file)

    # collect data
    def collect_data(self, cells):
        wb_obj = openpyxl.load_workbook(self.excel_file, data_only=True)
        sheet_obj = wb_obj.active

        return_data = {}
        for cell in cells:
            value = sheet_obj[cell].value
            return_data[cell] = value

        return str(return_data)
```

Auflistung A.9: Excel-Operation

CSV

```
import csv

class CSVOperation:
    def __init__(self, file):
        self.file = file
        if ".csv" not in file:
            self.file = file + ".csv"

    # Create a reader but maps the information in each row to a dict
    def read(self):
        with open(self.file, mode='r', newline='') as csv_file:
            data = csv.DictReader(csv_file)
        return data

    # Create a writer but maps dictionaries onto output rows.
    def write(self, dict_data):
        with open(self.file, mode='w', newline='') as csvfile:
            writer = csv.DictWriter(csvfile, fieldnames=list(dict_data.keys()))

            writer.writeheader()
            writer.writerow(dict_data)
```

Auflistung A.10: CSV-Operation

PKL

```python
import pickle

class PKLReader:
    def __init__(self, file):
        self.pkl_file = file

    def load_obj(self):
        file_name = self.pkl_file
        if ".pkl" not in self.file:
            file_name = file_name + ".pkl"
        with open(file_name, 'rb') as f:
            return pickle.load(f)

    def write_file(self, data):
        with open(self.file, 'wb') as f:
            pickle.dump(data, f)
```

Auflistung A.11: PKL-Operation

PDF

```python
import tabula
import PyPDF2

class PdfOperation:
    def __init__(self, original_file):
        self.pdf_file = original_file

    # read text of a page
    def read_text(self, page=0):
        file = open(self.pdf_file, "rb")
        pdfReader = PyPDF2.PdfFileReader(file)
        text = pdfReader.getPage(page).extractText()
        file.close()
        return text

    # read data from a table
    def read_data_from_table(self, row, column, page, factor, table_number=1):
        df = tabula.read_pdf(self.pdf_file, pages=page)
        data = df[int(table_number)]
        value = data.values.tolist()
        return_data = value[row][column]
        if not str(factor) == '':
            return_data = return_data.replace(',', '.')
            return_data = float(return_data) * factor

        return str(return_data)
```

Auflistung A.12: PDF-Operation

A.8.3 Weitere Services

eASiMOV-Service

Das Software-Framework **E**lectrical Grid **A**nalysis **Si**mulation **M**odeling **O**ptimization and **V**isualization (eASiMOV) [Çak+18] ist eine erweiterbare Sammlung von Softwaretools zur interaktiven Stromnetzmodellierung und -analyse, die eine verteilte Systemarchitektur einführt. Daten können in Form von Stromverbrauchs-/Erzeugungszeitreihen über das epowweb-Modul des Frameworks extrahiert werden. Dieses

Modul bereitet die Daten zur Qualitätssteigerung vor und stellt sie über einen Service mit REST-Schnittstelle in verschiedenen Aggregationsstufen und Zeiträumen zur Verfügung.

Um die historischen Netzdaten vom eASiMOV abzuholen, stellt PROOF einen *eASiMOVService* mit HTTP-Request bereit (siehe Auflistung A.13). Drei Parameter müssen auf der Weboberfläche konfiguriert werden, nämlich *url* (die Adresse des HTTP-Servers vom eASiMOV), *start_time* und *end_time* (siehe Zeile 5). Durch Aufruf der Methode *collect_data ()* werden die erforderten Netzdaten automatisch gesammelt und zurückgegeben (siehe Zeile 12 bis 18). Dieser Service wird für den Anwendungsfall 7.2.2 [Gon+21] in Kapitel 7 verwendet, um die Netzdaten zur Prognose zu sammeln.

```
1  import json
2  import requests
3
4  class eASiMOVService:
5      def __init__(self, url, start_time, end_time):
6          self.start_time = start_time
7          self.end_time = end_time
8          self.url = url #http://141.52.44.103:8090/eASiMOV/service/S-
9          self.strands = {"131": 9, "211": 9, "244": 9, "273": 9, "308": 5, "311": 9, "323": 7, "348N": 4,
               "351": 7, "353": 3, "414": 5, "429": 7, "500": 2, "523": 8, "546": 8, "611": 5, "631": 3,
               "660": 10, "675": 9, "713": 2, "717": 2}
10         self.output = {}
11
12     def collect_data(self):
13         for strand in self.strands:
14             url = self.url + strand + "/start/" + self.start_time + "/end/" + end_time + "/vis"
15             r1 = requests.get(url)
16             data = json.loads(r1.content)
17             self.output[stand] = data
18         return self.output
```

Auflistung A.13: eASiMOV-Service

A.9 Anwendungsfälle

A.9.1 Welder et al.: Spatio-temporal optimization of a future energy system for power-to-hydrogen applications in Germany

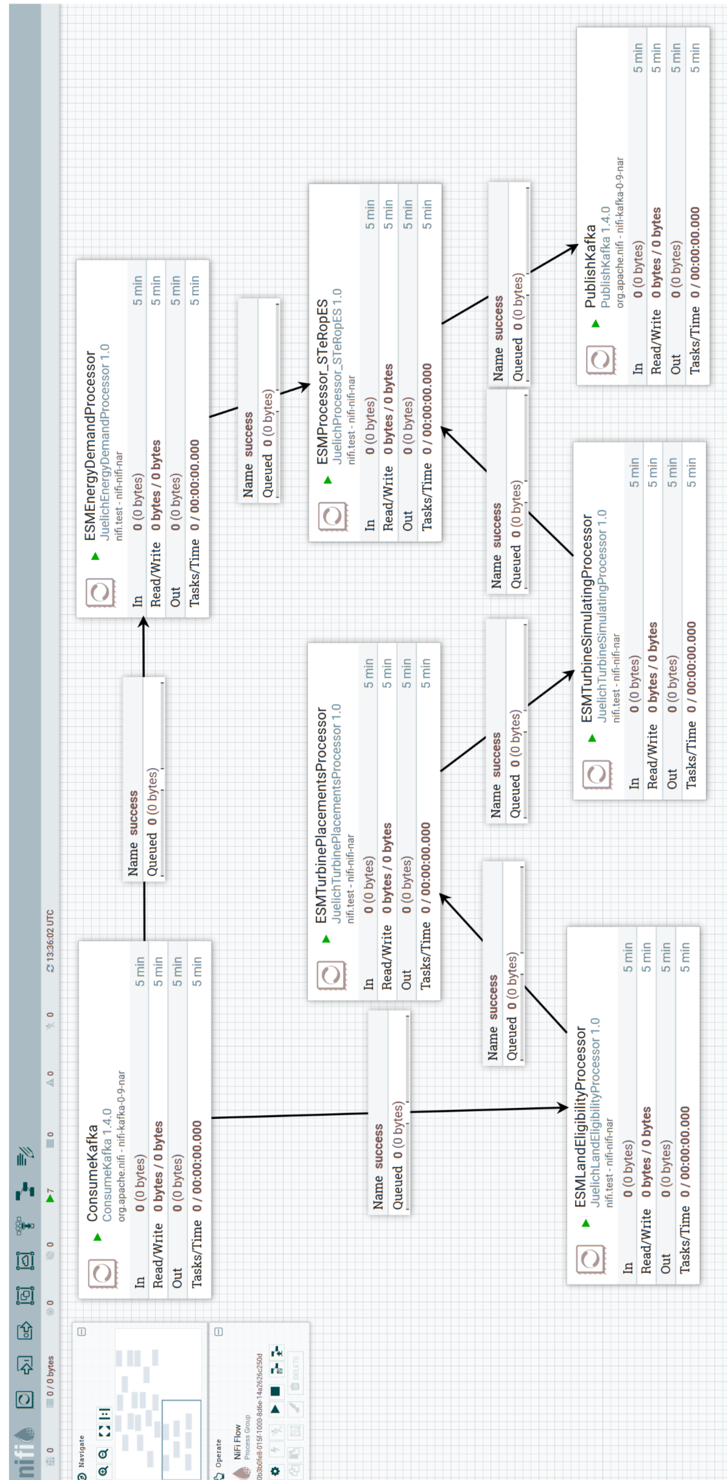

Abb. A.7.: Integration des ESM in PROOF [Liu+18]

A.9.2 González-Ordiano et al.: Probabilistic forecasts of the distribution grid state using data-driven forecasts and probabilistic power flow

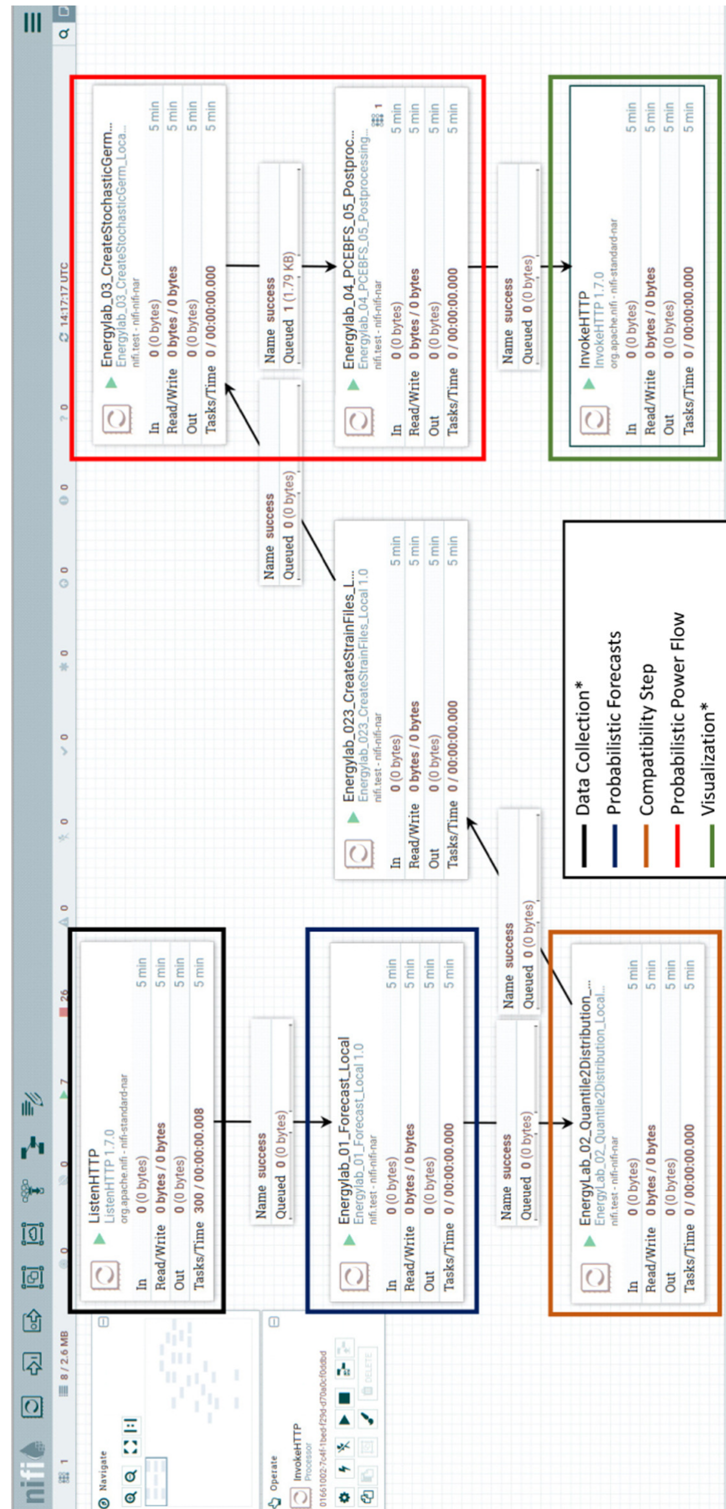

Abb. A.8.: Implementierung des Frameworks in PROOF: Sowohl die Datenerfassungs- als auch die Visualisierungsschritte werden durch die Blöcke dargestellt, die eine Verbindung zu easimov herstellen, um Daten zu extrahieren oder Daten für die Visualisierung zu speichern. [Gon+21]

A.9.3 Poppenborg et al.: Energy Hub Gas: A Multi-Domain System Modelling and Co-Simulation Approach

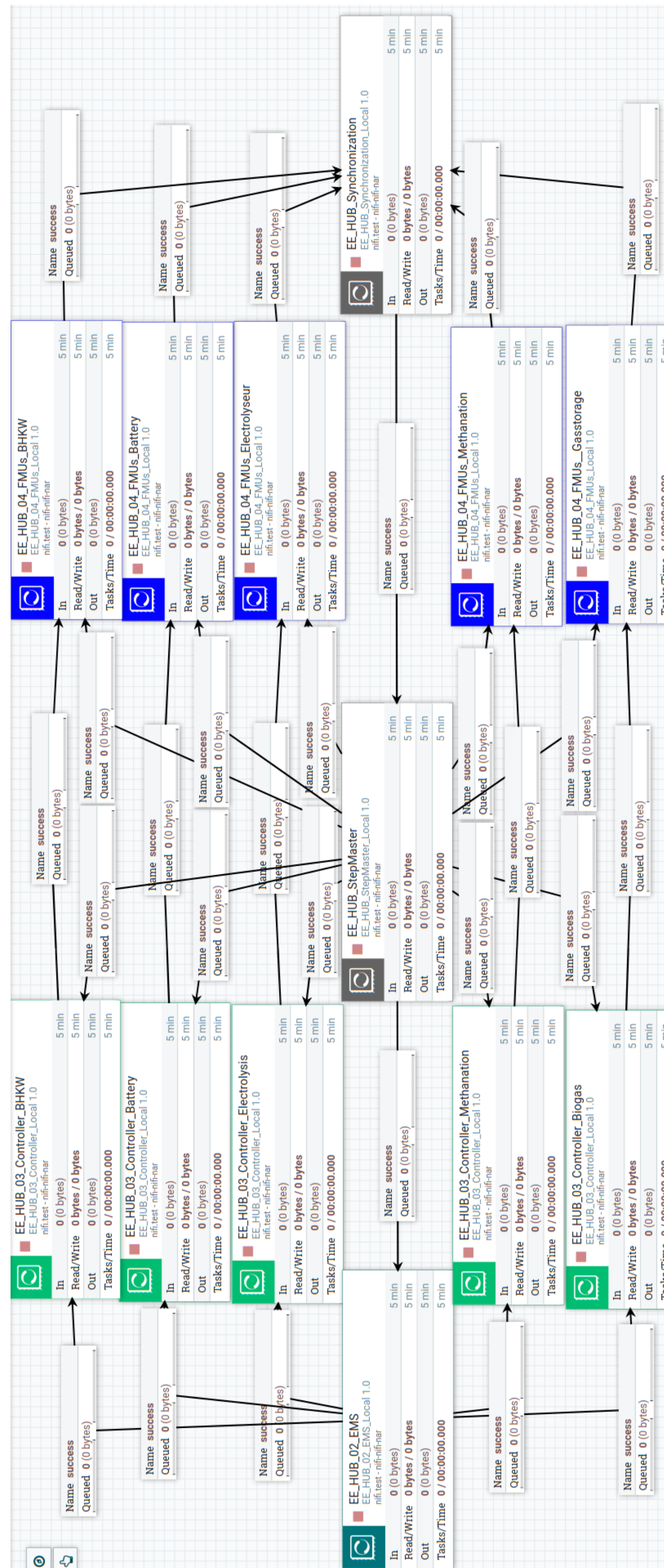

Abb. A.9.: Implementierung des EH in PROOF [Pop+21]

Publikationen

[Chl+22] Malte Chlosta, Jianlei Liu, Rafael Poppenborg et al. "An adapter-based architecture for evaluating candidate solutions in energy system scheduling". Englisch. In: *Energy Informatics* 5.S4 (2022). 37.12.02; LK 01, S. 56. DOI: 10.1186/s42162-022-00246-z (zitiert auf Seite 77).

[Gon+21] Jorge Ángel González-Ordiano, Tillmann Mühlpfordt, Eric Braun et al. "Probabilistic forecasts of the distribution grid state using data-driven forecasts and probabilistic power flow". In: *Applied Energy* 302 (2021), S. 117498. DOI: https://doi.org/10.1016/j.apenergy.2021.117498 (zitiert auf den Seiten 109, 145, 146, 148, 153, 154, 223, 227).

[Kha+18] Hatem Khalloof, Wilfried Jakob, Jianlei Liu et al. "A generic distributed microservices and container based framework for metaheuristic optimization". Englisch. In: *Proceedings of the Genetic and Evolutionary Conference Companion, Kyoto, J, July 15-19, 2018*. 2018 Genetic and Evolutionary Computation Conference. GECCO 2018 (Kyōto, Japan, 15.–19. Juli 2018). 37.98.11; LK 01. Association for Computing Machinery (ACM), 2018, S. 1363–1370. DOI: 10.1145/3205651.3208253 (zitiert auf den Seiten 152, 153, 155).

[Liu+18] Jianlei Liu, Eric Braun, Clemens Düpmeier et al. "A Generic and Highly Scalable Framework for the Automation and Execution of Scientific Data Processing and Simulation Workflows". In: *2018 IEEE International Conference on Software Architecture (ICSA)*. 2018, S. 145–14510. DOI: 10.1109/ICSA.2018.00024 (zitiert auf den Seiten 27, 28, 32, 111, 142, 145, 153, 154, 225).

[Liu+19] Jianlei Liu, Eric Braun, Clemens Düpmeier et al. "Architectural Concept and Evaluation of a Framework for the Efficient Automation of Computational Scientific Workflows: An Energy Systems Analysis Example". In: *Applied Sciences* 9 (Feb. 2019). DOI: 10.3390/app9040728 (zitiert auf den Seiten 3, 111, 145, 153, 154).

[Liu+23] Jianlei Liu, Malte Chlosta, Nicolas Schaber et al. "Introducing PROOF - A PROcess Orchestration Framework for the Automation of Computational Scientific Workflows and Co-Simulations". In: *2023 Open Source Modelling and Simulation of Energy Systems (OSMSES)*. 2023, S. 1–6. DOI: 10.1109/OSMSES58477.2023.10089680 (zitiert auf Seite 180).

[LDH17] Jianlei Liu, Clemens Düpmeier und V. Hagenmeyer. "A New Concept of a Generic Co-Simulation Platform for Energy Systems Modeling". In: *FTC 2017 - Future Technologies Conference 2017*. Nov. 2017 (zitiert auf den Seiten 30, 153, 154).

[Pop+21] Rafael Poppenborg, Johannes Ruf, Malte Chlosta et al. “Energy Hub Gas : A Multi-Domain System Modelling and Co-Simulation Approach”. Englisch. In: *Proceedings of the 9th Workshop on Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES ’21)*. MSCPES 2021 (Online). 37.12.03; LK 01. Association for Computing Machinery (ACM), 2021, Art.Nr. 12. DOI: 10.1145/3470481.3472712 (zitiert auf den Seiten 109, 111, 130, 134, 149, 152–155, 229).