Tampere University

Panu Aho

AN OPEN SOURCE DIGITAL TWIN FRAMEWORK

Faculty of Engineering and Natural Sciences
Master's thesis
November 2019

ABSTRACT

Panu Aho: An Open Source Digital Twin Framework
Master’s thesis
Tampere University
Degree Programme in Management and Information Technology
November 2019

In this thesis, the utility and ideal composition of high-level programming frameworks to facilitate digital twin experiments were studied. Digital twins are a specific class of simulation artefacts that exist in the cyber domain parallel to their physical counterparts, reflecting their lives in a particularly detailed manner. As such, digital twins are conceived as one of the key enabling technologies in the context of intelligent life cycle management of industrial equipment. Hence, open source solutions with which digital twins can be built, executed and evaluated will likely see an increase in demand in the coming years.

A theoretical framework for the digital twin is first established by reviewing the concepts of simulation, co-simulation and tool integration. Based on the findings, the digital twin is formulated as a specific co-simulation class consisting of software agents that interact with one of two possible types of external actors, i.e., sensory measurement streams originating from physical assets or simulation models that make use of the mentioned streams as inputs.

The empirical part of the thesis consists of describing ModelConductor, an original Python library that supports the development of digital twin co-simulation experiments in presence of online input data. Along with describing the main features, a selection of illustrative use cases are presented. From a software engineering point of view, a high-level programmatic syntax is demonstrated through the examples that facilitates rapid prototyping and experimentation with various types of digital twin setups.

As a major contribution of the thesis, object-oriented software engineering approach has been demonstrated to be a plausible means to construct and execute digital twins. Such an approach could potentially have consequences on digital twin related tasks being increasingly performed by software engineers in addition to domain experts in various engineering disciplines. In particular, the development of intelligent life cycle services such as predictive maintenance, for example, could benefit from workflow harmonization between the communities of digital twins and artificial intelligence, wherein high-level open source solutions are today used almost exclusively.

Keywords: Digital Twin, Co-Simulation, Tool Integration, Intelligent Life Cycle Management, Open Source

TIIVISTELMÄ

Panu Aho: Avoimen lähdekoodin ohjelmistokehys digitaalisille kaksosille
Diplomityö
Tampereen yliopisto
Johtaminen ja tietotekniikka
Marraskuu 2019

Diplomityössä tutkittiin korkean tason ohjelmointikehyksiä, jotka mahdollistaisivat digitaalisten kaksosten suunnittelun sekä niihin liittyvien simulointitehtävien ajamisen. Digitaaliset kaksoset ovat erityinen luokka simulaatioita, jotka peilaavat jonkin fyysisen maailman laitteen tai prosessin toimintaa erityisen tarkasti. Digitaalisia kaksosia pidetään eräänä älykkäiden elinkaarenhallintapalveluiden tärkeimmistä mahdollistavista teknologioista. Näin ollen on luultavaa, että kysyntä avoimen lähdekoodin ratkaisuille, jotka mahdollistavat digitaalisten kaksosten rakentamisen, suorittamisen ja arvioinnin tulee lähivuosina lisääntymään.

Työssä rakennetaan ensin teoreettinen viitekehys digitaalisille kaksosille tutkimalla simuloinnista, co-simuloinnista ja työkaluintegraatioista julkaistua kirjallisuutta. Tuloksiin pohjautuen muotoillaan digitaaliselle kaksoselle malli. Mallissa digitaalinen kaksonen käsitetään co-simuloinnin erikoistapaukseksi, jossa erikoistuneet ohjelmistoagentit kommunikoivat joko fyysisten laitteiden tai simulaatiomallien kanssa, välittäen fyysisiltä laitteilta tulevaa mittausdataa simuloinnin syötteiksi.

Työn empiirinen osuus koostuu ModelConductor -kirjaston kuvauksesta. Kyseessä on uusi Python-kirjasto joka tukee reaaliaikaisella syötedatalla varustettujen digitaalisten kaksosten kehittämistä. Pääominaisuuksien kuvaamisen lisäksi työssä esitellään joukko kuvaavia esimerkkikäyttötapauksia. Esimerkkien kautta demonstroidaan myös kirjaston korkean tason ohjelmointisyntaksia, joka mahdollistaa nopeat kokeilut useilla erityyppisillä digitaalisilla kaksosilla.

Työn päätuloksena esitetään, että digitaalisten kaksosten rakentamiseen soveltuvien työkalujen luonti on mahdollista toteuttaa olioparadigmaan perustuen korkean tason ohjelmointikielillä. Kyseisellä lähestymistavalla voi tulevaisuudessa olla vaikutuksia siihen, minkälaiset organisaatiot ja henkilöt digitaalisiin kaksosiin liittyviä tehtäviä suorittavat. Tärkeimpänä mainittakoon havainto, että ohjelmistokehykseen pohjautuva ratkaisu voisi tuoda digitaalisten kaksosten kehittämisen lähemmäs ohjelmistoalan ammattilaisia, sen lisäksi että sitä tekevät useiden perinteisempien insinöörialojen asiantuntijat. Tämä pitää erityisesti paikkansa älykkäiden elinkaarenhallintapalveluiden, kuten prediktiivisen kunnossapidon, konteksissa, missä jo nyt pitkälti käytetään samantyyppisiä avoimen lähdekoodin ratkaisuja koneoppimis- ja tekoälytehtävissä.

Avainsanat: Digitaalinen kaksonen, co-simulointi, työkaluintegrointi, älykäs elinkaaren hallinta, avoin lähdekoodi

PREFACE

As it is the case with all major undertakings in life, the realization of this thesis would have been considerably harder without the valuable support from numerous advice givers, colleagues, friends and family members. I wish to extend my gratitude to prof. Tarmo Lipping and prof. Sami Hyrynsalmi at TUNI Pori for meticulously scanning through various revisions of the text and giving insightful improvement suggestions, both of the scientific and technical variety. I also want to thank Dr. Eero Immonen and Mr. Mika Lauren at Turku University of Applied Sciences (TUAS) for providing the initial idea, helping shape the research question to fit academic standards and always being available for discussion when work seemed to be hitting the brick wall.

Kudos also go out to Mr. Krister Ekman, Mr. Toomas Karhu and Mr. Miika Laivola at TUAS Internal Combustion Engine Laboratory for helping with extracting the raw data required for the thesis' empirical part and for being such great co-workers for all these years. Lassi Roininen (LUT University), Tommi Karhela (Semantum Oy), Juha Kortelainen (VTT), Karri Penttilä (VTT) and Petteri Kangas (VTT) are to be thanked for pointing me to new, interesting directions in various discussions throughout the process. Sadly, Petteri passed away during the finalization of the thesis; my thoughts are with his friends and family.

Finally, I want to express my deepest love and gratitude for my wife, Susanna, and my son, Niilo. As You know, three years ago I chose the extremely difficult path of pursuing a master's degree while also working full-time for the most part. While I do feel relieved and proud that this endeavor is now coming to an end with the finishing of this thesis, I am also painstakingly aware that during this process I have not been able to be the husband and father You deserve. Words do not exist to thank you enough for your support, patience and (still) being there for me. Now it's my turn to return all the support and encouragement, for which ever paths You might choose to follow in your lives.

Turku, 25th November 2019

Panu Aho

CONTENTS

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

AI	Artificial Intelligence
API	Application Programming Interface
BTDC	Before Top Dead Center
CAN	Controller Area Network
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
CSV	Comma Separated Values
DL	Deep Learning
DSML	Domain-Specific Modeling Language
DSR	Design Science Research
ECU	Engine Control Unit
FEDS	Framework for Evaluation in Design Science Research
FIFO	First In, First Out
FMI	Functional Mock-up Interface
FMU	Functional Mock-up Unit
FOM	Federation Object Model
FTG	Formalism Transformation Graph
GUI	Graphical User Interface
HILS	Hardware-In-Loop Simulation
HLA	High-Level Architecture
I/O	Input/Output
ICEL	Internal Combustion Engine Laboratory
IDM	Integrated Data Model
IEEE	Institute of Electrical and Electronics Engineers
IMS	Integrated Model Server
IoT	Internet of Things
JSON	JavaScript Object Notation
MAE	Mean Absolute Error
ML	Machine Learning
MQTT	Message Query Telemetry Transport
MVP	Minimum Viable Product
NASA	National Aerospace and Space Administration
NO_x	Nitrogen Oxides
NRTC	Non-Road Transient Cycle
ODE	Ordinary Differential Equation
OEM	Original Equipment Manufacturer
OMT	Object Model Template
OOP	Object Oriented Programming
OTI	Open Tool Integration
PM	Particulate Matter
REST	Representational State Transfer
RTI	Runtime Infrastructure
SOM	Simulation Object Model
SQL	Structured Query Language
TA	Tool Adaptor
TCP	Transmission Control Protocol
TDC	Top Dead Center
TUAS	Turku University of Applied Sciences
XMI	XML Metadata Interchange
XML	Extensible Markup Language

T_i	i-th tool in a set of tools
A_n	Injector nozzle total area
C_d	Discharge coefficient
P_{out}	Break output power of an internal combustion engine
R_{DT}	A digital twin relation
m_f	Injected fuel mass
p_{exh}	Exhaust back pressure (before turbine)
p_{int}	Boost pressure
t_x, t_i, t_f	Individual time instants in $\mathcal{T}$
$\theta_{0,E}$	Injection timing flag
θ_0	Start of injection timing, crank degrees BTDC
ρ_f	Fuel density
$\omega_{[t_i,t_f]}$	An input function evaluated over a closed interval, $\omega([t_i, t_f])$
$\Delta\tilde{x}_N$	Left Riemann Sum approximate for displacement
Δx_T	The one-dimensional displacement of a body during elapsed time T
Δp	Pressure drop across nozzle
Ω	Set of input functions
N	Target engine speed
P	A set of process agents
Q	Set of system states
S	A set of simulation agents
T	Torque
X	Set of input states
Y	Set of outputs
n	Revolution rate
p	A process agent
q, q_i	A system's internal state (at time instant i)
s	A simulation agent
t	Simulation time
$v(t)$	The velocity function
x	An input state
y	An output state
$\mathcal{T}$	Set of time instants
δ	Transition function
λ	Output function
τ	Wall-clock time
ω	An input function (Chapter 3), Angular velocity (Chapter 5)

1. INTRODUCTION

The industrial paradigm is gradually shifting from product-centric to service-centric. Instead of simply selling products to customers, an increasing number of businesses are now putting the emphasis on development of long-term partnerships with their customers, wherein the physical product plays a necessary but insufficient role. From the perspective of the manufacturer, this implies that revenue streams can be extended to span the whole life chain of a product, including design, manufacturing, delivery, operational use and retirement. Simultaneously, this allows for the customer to keep focus on their core competences, when many of the utilities and assets once perceived as physical hardware are now available for procurement in “as a service” manner.

With the advent of Internet of Things (IoT), the very concept of what might constitute an industrial service is also transforming [1]. Of particular interest are the services that a manufacturing company might provide during the operational lifetime of a physical asset. Depending on the particular field of industry, the basic idea of providing maintenance services for installed production equipment is, of course, nothing new. However, technological advancements in areas such as sensor and actuator technology, low-latency network connectivity and Artificial Intelligence (AI) are opening up unforeseen avenues for applications regarding predictive maintenance, fault detection, implementation of dynamic lifetime optimization strategies, and more [2]. Intelligent life cycle services, along with the adoption of other smart manufacturing strategies, show considerable potential to foster businesses’ productivity and competitiveness [3].

As applications of the discussed variety become more mainstream, the requirement for high quality *process models* and numerical *simulation* capabilities becomes evident. For illustration, let us consider a complex engineering system with dynamic multi-input, multi-output characteristics, e.g., an internal combustion engine or a nuclear power plant. If an agent (be that human or a machine) is to make inferences about such a system’s behavior in the future, scarcely good results can be expected if no other information is available than the system’s behavior in the past. Instead, state-of-art process models are commonly used, for which future trajectories of crucial quantities of interest can be approximated by numerical methods. Simulation is the process of solving a dynamic model’s state trajectory with respect to time, given a certain set of preconditions, e.g., the system’s state history and other model-specific attributes. Whereas a model can be considered an approximation and an abstraction of a well-defined portion of reality, simulation is the act of observing that model’s behavior in time.

When conducting dynamic system simulation, it is common to divide a complex engineering system to multiple submodels. In industrial practice, the submodels are often developed in isolation from one another; by different people, using different modeling tools with various internal formalisms. While this approach might result in a set of submodels that can be considered accurate within their respective domains, integration complexities are likely to arise when these submodels are coupled to perform system-level simulations. With the recent emergence of IoT based life cycle services, an increasing demand for such complex model interoperability is identified, stemming from the need for extremely accurate system models. This has motivated substantial scholarly activity in the field of *co-simulation*, which refers to solving state trajectories for complex coupled models regardless of the individual submodels' formalisms [4]. High-Level Architecture (HLA) [5] and, more recently, Functional Mock-up Interface (FMI) [6] are some concrete attempts aiming to promote this kind of modeling tool interoperability.

An extreme use case for (co-)simulation emerges with the introduction of *digital twins*, software representations of physical hardware that coexist with and mirror the evolution of their physical counterparts in a particularly detailed manner [7]. If the target system is of any considerable complexity, a multidisciplinary approach is herein mandatory, enforcing the requirement for interoperability between various simulation tools and modeling paradigms. Often digital twin simulations need also to be augmented with online real-world sensor data to reflect the changes occurring in the target system over time. Hence digital twin technology can, arguably, provide fertile ground for the development of IoT-based life cycle applications. A prerequisite for this, however, is that a specific class of middleware software exists; one that would take care of low-level tasks such as facilitating communication between individual sub-simulations and provide means for the physical and digital product to exchange data.

Traditionally, tool integration capabilities are often built-in to proprietary "tool suites", such as the MATLAB Simulink package. Open source solutions, while certainly not without their flaws, are preferred in many cases for better extensibility, workflow integration, transparency and lower costs. This premise holds true in some of the most interesting domains that intersect the present discussion about IoT life cycle applications. Most prominently, the deep learning community is herein referred to, where open source codebase is used almost exclusively as opposed to proprietary solutions [8], [9]. Modern open source deep learning frameworks have gained substantial traction recently as they generally provide an intuitive high-level interface for fast experimentation and prototyping of a variety of deep learning scenarios. Meanwhile the frameworks of this category (e.g., Tensorflow or PyTorch) have evolved to be powerful enough to support production-grade applications, with strong user communities able to provide the required level of development and support.

In the empirical part of the present thesis, a software engineering approach to the digital twin application development process is proposed. It is a valid argument that many contemporary simulation tool suites already offer the functionality to develop tool integration solutions suitable for digital twins [10]. However, the workflows incorporated into these pieces of software are generally aimed at the domain experts of the particular engineering discipline, rather than at the software engineer. The interrelation between the discussed developments in the AI community and intelligent industrial life cycle services suggests that future digital twin implementations might require more and more coordinated efforts between the two expert domains. This sparks the driving motivation of the present work: a suggestion that a demand exists for high-level open source programming interfaces in the domain of simulation tool and sensor data integration. Specifically, the present work aims to address the following questions:

- What kind of standards, theoretical constructs and/or “best practices” are reported to facilitate the development of digital twin middleware solutions?
- What features should such a solution entail and how should they be implemented in order to facilitate rapid prototyping and experimentation?

The remainder of this thesis is organized as follows. Chapter 2 discusses the background of the research objectives in a finer detail and, correspondingly, describes and gives rationale for the chosen research methodology. In Chapter 3, literature in the closely related fields of co-simulation and tool integration is investigated, in an attempt to establish a concise conceptual space for the empirical part of the work. Based on the findings, a general digital twin framework meta-model is proposed that is utilized throughout the work’s empirical part. Subsequently in Chapter 4, the main outcome of the constructive part, *ModelConductor*, is described: an original Python library to support rapid development of digital twin experiments integrated with online measurement data. A handful of illustrative use cases are presented in Chapter 5. Finally, in Chapter 6 some concluding remarks and ideas for further study are presented.

2. PROBLEM SPACE AND METHODOLOGY

In this chapter, the justification for an open source digital twin framework is established by synthetizing, based on literature review, a set of features that characterize digital twins. The mapping of these features to the functional requirements of an envisioned software environment that would facilitate the construction of digital twin applications is discussed. Moreover, it is described how the present work is motivated by recent developments in closely related fields in the intelligent industrial service space, namely artificial intelligence and deep learning. Finally, the chosen constructive research methodology, including the artefact evaluation strategies utilized, is elaborated.

2.1 Problem background

Agile practices in software engineering have been actively developed since the 1990's and today they have practically replaced the traditional waterfall model [11]. In general, they are valued over traditional methods for their potential to create customer value incrementally in rapid development cycles. While not implying a causal relationship between the two, it is an interesting observation that the earliest adoptions of agile methods loosely coincide with the introduction of the first wide-spread object-oriented programming languages such as C++ (first appeared 1983, ISO-standardized in 1998) and Java (1996). This supports the idea of an interconnection between the two; that the availability of high-productivity tools might, at least indirectly, promote the uptake of high-productivity industrial practices and vice versa.

The motivation of the present work is to study the existence, utility and ideal composition of high-level, high-productivity software frameworks in the context of the emerging technology of digital twins. Although similar ideas have been proposed earlier[1], the term "digital twin" is predominantly cited to appear first in Glaessgen's and Stargel's article commissioned by NASA and the U.S. Air Force [7]. While the formal definitions are deferred to Chapter 3.4, the main findings from recent literature (e.g. [7], [10], [12]–[17]) can be summarized as digital twins being a specific class of simulation artefacts that exhibit traits of the following guidelines:

- **Representativity:** Each digital twin is a *digital* artefact that has a corresponding paired *physical* artefact. The digital counterpart is expected to be representative,

[1] See, e.g., [72], or [55] for more of a metaphysical approach.

in the context of an *experimental frame* (Chapter 3.1), of the physical one. Generally, this means that reliable conclusions can be made of the physical system's behavior by observing the digital twin's response to artificial excitation signals.

- **Inspectability:** At any time, any attribute entailed in the digital twin should be easily accessible by an appropriately authorized agent. This can be viewed as a principle of transparency, where digital twins typically expose their input and output signals, but also the internal state of the simulation unit that could be hard to measure by conventional means from the corresponding physical equipment.
- **Contemporality:** The existence of a digital twin should, in the most general case, extend to span the whole life cycle of its physical counterpart. Active efforts should be taken by the twin itself and/or an encapsulating middleware layer to reflect changes occurring during the lifetime.

The key observation underlying the rationale of the present work is an idea that the realization of a digital twin requires bridging the gap between two distinct domains, i.e., sensory measurement data derived from real-world objects (IoT) and, on the other hand, the complex compositions of high-fidelity simulation models (co-simulation). Conceptually, this can be viewed as being related to Hardware-In-Loop Simulation (HILS), a generic set of techniques that is commonplace in numerous disciplines such as power electronics [18], civil engineering [19] and biomedicine [20] only to name a few. It is observed, however, that the general HILS paradigm, as viewed in a large portion of published work, is rooted in the strict segregation between the digital and physical realms. In a coupled HILS system, some parts may exist in a digital and some in a physical form, without significant overlap. Moreover, the connections between the subsystems are usually sequential in nature, with clearly defined input-output relations between the subsystems. In this regard, a digital twin is very different. In order to support the design goal of representativity, it is actually desirable to make the digital-physical overlap as large as possible. Similarly, in a digital twin, it is clear that signals are exchanged between the physical and digital counterparts (contemporality), but without any clear-cut static definitions regarding what are the data sources and data consumers.

Clearly, the implications of above discussion are multitudinous regarding the extended pool of use cases that could be envisioned for a digital twin versus a HILS setup of more conventional nature. Perhaps even more importantly, a myriad of questions arises regarding the imposed functional requirements for the middleware between the digital and physical realms. This is emphasized by the fact that in the most general envisioned use case the data sources (i.e., the physical objects producing the sensory data) and data consumers (the executable simulation units) can be distributed in terms of their physical locations and/or computational environments. One could argue that in very simple cases digital twin setups could be successfully constructed, utilized and evaluated using an approach that is more or less ad hoc in nature. However, more complex setups, wherein qualities such as input-output validity, stability, real-time performance and scalability are of interest, seem to create a demand for more rigorous digital twin software frameworks that are developed and tested against the best available software engineering practices.

On a high abstraction, a complex digital twin could be realized as an ensemble of software agents communicating with artefacts either in the physical or digital domain. Correspondingly, any framework that is conceived to support the building of such configurations should be able to appropriately curate these intricate many-to-many connections. For general usability, it would need to be considered that the involved physical/digital artefacts might be accessible through a variety of technologies, e.g., REST APIs, SQL/NoSQL database queries, or a more sophisticated IoT communication protocol such as the Functional Mock-up Interface (FMI) [21] or Message Queuing Telemetry Transport (MQTT) [22]. The appropriate resolving of the varying formalisms to provide a coherent environment for the envisioned network of digital and physical systems would be one of the core functions of a digital twin framework.

At this point, a legitimate question is: What exactly do statements like "*appropriately* curate" or "the *appropriate* resolving of the varying formalisms" mean? The answer is that these refer to fulfilling the non-functional requirements of representativity, inspectability and contemporality. As it is proposed, representativity translates to the functional requirement of co-simulation: the act of coordinating the parallel execution of multiple state-of-art simulation models that might require distinct tools and computational environments to run. Inspectability, on the other hand, can be interpreted so that the system should entail a sufficient level of transparency. In practice, this could mean that up-to-date information is, to the extent possible, made available to the user near real-time about the state of physical and digital assets.

The third design criterion, contemporality, is perhaps the hardest one to grasp in terms of easily explainable functional requirements. One possible standpoint is that one could think of it referring to ensuring the concurrent existence of a physical object and its digital counterpart throughout the whole product life cycle including planning, manufacturing, delivery, operational use and retirement. Obviously, if the digital twin is to be live throughout each and every phase alongside the physical product, this places considerable implications on making the phase-specific computational infrastructure available at each point in time. Even if one only considers the more realistic use case of using the digital twin only during the operational life of a product, issues regarding persistence and fault tolerance would still need to be resolved.

In the design of digital twin applications, the notion of expansibility needs also to be considered. When the operational lifetime of a product might span tens of years, it is conceivable that add-ons will be augmented to the physical product that cannot be adequately replicated within the original experimental frame of the twin. Hence, the digital twin framework should, ideally, provide convenient means to make adjustments to the experimental frame and add new model components as the physical counterpart evolves. An additional design point, as will be illustrated in Chapter 4.4, is that a user-centric design point is essential for such a software to gain significant popularity. Hence, the software should ideally be intuitive enough so that it can accommodate fast prototyping, as well as flexible enough to accommodate extensibility and deployability for real-world applications.

2.2 Research methodology

The methodology of choice in seeking answers to questions outlined in Chapter 1 predominantly falls to the category of action research. More accurately, to position the present study in the space of research approaches and to give rationale for the selected methods, the works of Myers & Avison [23, pp. 7–8] and, further, Rapoport [24] are referred to. Therein, action research is postulated as a qualitative instrument well suited to situations where the researcher wishes to accomplish two distinct goals, i.e., i) solve a contemporary problem encountered in an organization and ii) contribute to the knowledge pool that can be later utilized by others in order to solve instances of the same problem class.

The initial inspiration for this thesis resulted from the author’s employment as a member of the Smart Machines research group in Turku University of Applied Sciences (TUAS), thus fulfilling the first criterion. At TUAS, the idea of using state-of-art process models to create digital twins of internal combustion engines and various types of vehicles and heavy-duty machines was recently contemplated[2]. Initially, this motivated the search for the middleware components of the discussed variety in the context of an isolated use case. The main motivation for this thesis, however, stems from the apparent extensibility of potential results, hence fulfilling the second criterion.

[2] At the time this thesis is published, the employment of digital twins in this context remains an active investigation within the e3Power project funded by Business Finland and various industrial partners [73].

Following the taxonomy proposed by Järvinen [25], action research can also be viewed as a subclass of artefact building / artefact evaluating research approaches, wherein the building and evaluation phases are executed in a cyclical manner. In the field of Design Science Research (DSR), it is an established idea that the simple[3] act of building things and observing their performance, sometimes even in a post-hoc manner, has great potential to contribute both to direct technological advancements as well as the scientific knowledge base [26], [27]. Accordingly, in the present work, the constructive approach is selected with the intent of gaining a deeper understanding on the process of designing and executing digital twin experiments in the context of various continuous-time processes. The main part of present work's contribution resides in the Minimum Viable Product (MVP) prototype of a software environment that supports this goal. More specifically, the proposed solution will be an object-oriented library, written in Python, that seeks to create abstractions for various low-level digital twin functionalities. It is further observed that the cyclical artefact building / artefact evaluating research approach is obviously akin to the various modes of agile software development practices (e.g., Scrum and the likes). As such, it seems like an appropriate choice for the present work, the empirical part of which essentially boils down to rather standard software engineering project.

The chosen approach is further reasoned by observing the interconnection between the digital twins and some of the most interesting contemporary research topics in the space of data science. Today Machine Learning (ML), Deep Learning (DL) and Artificial Intelligence (AI) are practiced in increasing numbers by software engineers. This is contrary to what used to be the case until early 2010's, when these duties were predominantly handled by personnel with rigorous training in applied mathematics and statistics. An important contributing factor behind this transformation is the availability of modern open source deep learning libraries (e.g., Tensorflow, PyTorch, Keras) that are highly accessible to the general developer population in terms of their abstraction level. With ML/DL/AI technologies playing a crucial role in some of the future's most interesting digital twin applications (e.g., predictive maintenance), it is plausible that a subset of future developers will be involved in projects of both fields. Hence arises the notion of tool harmonization. This is an idea that improved productivity could be achieved in such endeavors if the programming interfaces in both domains would, at least approximately, support similar workflows. To the author's knowledge, no attempts to implement such tools in the digital twin domain have previously been reported.

[3] By "simple", only the general methodological concept is referred to, and not the process of actually constructing artefacts that could be of arbitrary complexity.

In a routine design operation, it is usually acceptable to simply conclude that the developed artefact “works” (or, that it “does not work”, for that matter) with respect to the immediate circumstances in which the development took place. On the contrary, design can obviously only become research when the evaluation part is applied with an adequate level of rigor — in such a manner that the research outcomes are somehow made generalizable to a wider class of problems. In formulating the evaluation strategy for the artefact(s) produced within the scope of the present work, the work reported in [28]–[30] is referred to. Therein, the authors describe a generic framework that can be utilized to examine the (software) artefacts that emerge from conducting a DSR project. The FEDS (Framework for Evaluation in Design Science Research) discusses an *evaluation episode* as the basic unit of evaluation, several of which might take place during a given DSR project. The evaluation episodes are mapped onto a 2-dimensional plane with one dimension representing the functional purpose of the evaluation (formative/summative), and one dimension representing the paradigm of the evaluation study (artificial/naturalistic). While the discussion about the particularities in the original work is rather involved, a rough interpretation for the two dimensions can be summarized as follows:

- Whereas artificial evaluation emphasizes controlled experiments in isolated environments, naturalistic evaluation tends to involve real users, real organizations and real problems.
- Whereas formative evaluation is concerned with producing suggestions how the evaluand can be improved, summative evaluation tends to produce information regarding the utility of the evaluand in the context of the evaluand’s envisioned applications.

An *evaluation strategy* is formulated by plotting consecutive evaluation episodes and development iterations in the $(\text{artificial–naturalistic}, \text{formative–summative})$ plane (Figure 1). The selection of an evaluation strategy for a given DSR project is, among other things, related to the analysis of risks involved in the development of the artefact. If there is little to no ambiguity involved in the choice of technologies used to produce the artefact, and there is strong a priori evidence that those technologies will function well in the scope of the project, the project can be considered having a low technological risk. If, at the same time, there is considerable uncertainty regarding how will the targeted users actually use the artefact and whether the design provides an appropriate social fit to the target organization, the project can be considered having high human risk. The same applies conversely, and it is the balance between the assessed risks on the human-technology continuum that can act as the guiding principle of selection of an appropriate evaluation strategy for each DSR undertaking. Obviously, this is merely a guideline and in practice, additional constraints apply: for instance, it could be very costly, or for practical reasons impossible to evaluate a given artefact with real users or real problems. [28]–[30]

Some examples of plausible evaluation strategies based on [28] are depicted in Figure 1, also introducing the applied strategy for ModelConductor, the main outcome of the empirical part of the thesis. In the present work, the evaluation continuum is built on top of three use case examples, as they are documented in Chapter 5, specifically:

- Combining a synthetic data stream to a Simulink-based simulation model. The purpose of this is to provide a proof of concept on the basic input-output functionalities of ModelConductor as well as auxiliary functions such as logging. This experiment tends towards the artificial-formative corner of the FEDS spectrum, since no real users or organizations are involved and the data is generated artificially.
- Combining real-world measurement data with a machine learning model. In this experiment, the data is real, but it is relayed through a dedicated intermediary database specifically engineered to accommodate the experiment. Hence, the experiment can be viewed as being somewhat closer to the naturalistic end.
- Combining real-world measurement data to a GT-SUITE based internal combustion engine simulation model. Herein the data originates directly from the target organization’s productional database, onto which ModelConductor interfaces. Hence, the experiment is very much involved with real problems occurring in a real organization. Furthermore, this evaluation episode tends to the summative end of the FEDS spectrum since the goal here will be to gather all the accumulated knowledge, based on which suggestions and limitations regarding future ModelConductor applications can be discussed.

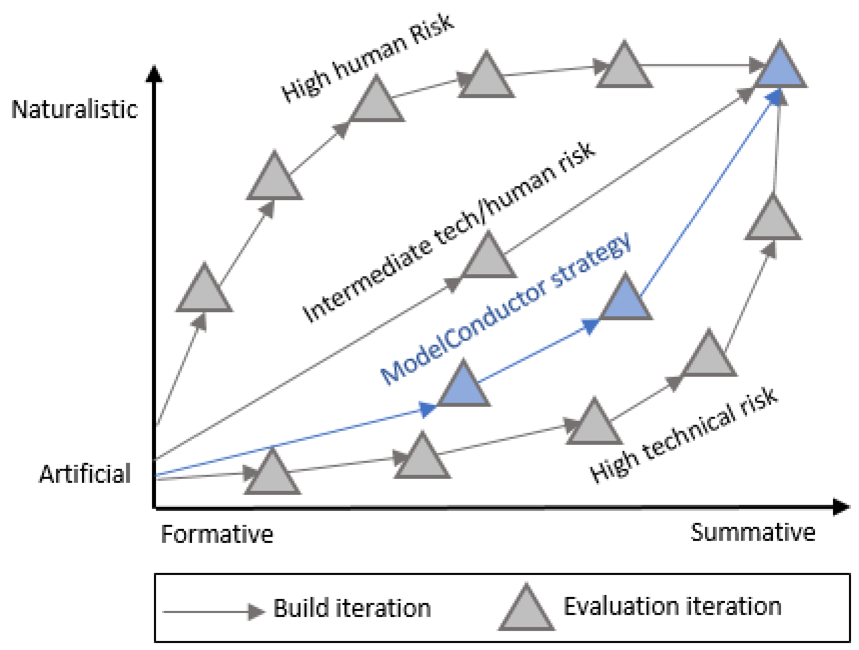


Figure 1. *Framework for Evaluation in Design Science Research (FEDS), adapted from* [28]

The presentation of the thesis will advance in a sequential manner from description of the research problem, establishing the underlying theoretical framework, describing the main features of the developed software and finally evaluating its performance and discussing the results. This approach is chosen merely for convenience, with simultaneously acknowledging that the empirical part of the work has in fact been carried out by following the iterative practices widely adapted in modern software engineering. Following the agile philosophy [31], a conscious effort is made throughout this text to keep the focus on the results and their applicability, rather than describing in detail what was done in each and every iteration. Fortunately, however, modern software engineering practices, if applied correctly, produce code that is self-documentative in nature. The interested reader will, consequently, find the version history of the proposed MVP's public GitHub repository a valuable resource [32].

3. CO-SIMULATION BACKGROUND AND RELATED WORK

The objective of this chapter is to establish a conceptual space within which various aspects of a digital twin can be examined in an appropriate context. Since there are surprisingly few published works that aim to establish a concise theoretical framework for digital twins, the subject is instead approached indirectly by reviewing literature in the closely related fields of simulation, co-simulation and tool integration. Specifically, the discussion aims to build up an argument that a digital twin can be viewed as a special occurrence of co-simulation, wherein agents representing physical measurement data from real-world objects and those representing various simulation models interact in a coordinated arrangement.

3.1 Simulation

It is a long-lived engineering tradition that the input data for a design process can generally be obtained by two orthogonal approaches, i.e., i) conducting experiments in the physical world and quantifying the made observations as measurement data and ii) by means of numerical simulation. As discussed in Chapter 1, simulation is defined as the practice of observing a dynamic computational model's behavior in time. Typically, simulation is conducted with the objective of achieving some design goal [33]. Also briefly discussed in Chapter 1, the model is, in this context, understood to be an abstraction of reality with some degree of representativeness of the target system being modeled.

As it is scarcely possible (or reasonable, for that matter) for a model to capture the full spectrum of features and interrelations present in complex engineering systems, the normal approach is that a model is constructed in accordance to some experimental frame [33]–[35]. An experimental frame is the collection of descriptive attributes regarding the model's components (i.e., inputs, internal states, outputs) that is necessary and sufficient to describe the evolution of a system with respect to some specific design or research goal [33]. As such, there might be multiple plausible experimental frames for a given real-world system, the selection of which is ultimately limited only by the hardware the simulation is targeted to run on, e.g., the finite memory space of a computer [33]. Aside from these technical considerations concerning computational resources, it is usually the case that other factors, such as model interpretability and workload of implementation become predominant in model selection. Following the Occam's Razor principle, one could argue that it is the modeler's task to find a suitable experimental frame that "just barely" gets the job done.

From an ontological point of view, a model can be considered an abstract idea that only becomes a concrete artefact once it is expressed with respect to some established set of grammar rules — a *formalism*. To discuss this intricate subject, a systems theory based approach can serve as a useful starting point. Consider, for instance, the discussion in [34] and [36, pp. 30–31] wherein a generic system meta-model is defined as:

$$SYS \equiv \langle \mathcal{T}, X, \Omega, Q, \delta, Y, \lambda \rangle \tag{1}$$

wherein

$\mathcal{T} \subset \mathbb{R}_{\geq 0}$	Set of possible time instants
$X \ni x$	Set of possible input states
$\Omega = \{\omega \mid \omega : \mathcal{T} \to X\}$	The set of possible input functions
$Q \ni q$	Set of possible system states
$\delta : \Omega \times Q \to Q$	Transition function
$Y \ni y$	Set of possible outputs
$\lambda : Q \to Y$	Output function

Notice that Equation 1 itself can be viewed as a formalism since it exhaustively defines the attributes an artefact must entail for it to be considered a valid SYS. The various components described in Equation 1 can, subsequently, be expressed in their own internal formalisms. Most importantly, consider the transition function, which takes in a current state, an input segment and produces a new state. According to the specific problem domain, the transition function might be expressed in a variety of formalisms such as differential equations, difference equations, bond graphs or finite state automata, for example [34], [37]. The selection of formalism used to describe a system should, in principle, be guided by such questions as the desired abstraction level or the availability of data that can be used to calibrate the model [34].

To conduct numerical simulation, in addition to a model, a *solver* is required [38]. In the context of continuous-time system models, a solver can be defined as an algorithm that sequentially advances the model time $\mathcal{T}$ and, at each time step, evaluates the model's internal state q and, possibly, the output y, with respect to an input x. The fact of y and q being fundamentally separate concepts allows the efficient separation of concerns in a solver's implementation [34]. So does also the observation that, in a valid SYS, transition function δ evaluated over the closed interval $[t_i, t_f] \subset \mathcal{T}$, with q_i being the internal state at t_i, can be recursively decomposed to a set of arbitrary sub-intervals (Equation 2) [34].

$$\forall t_x \in [t_i, t_f], \delta\left(\omega_{[t_i,t_f]}, q_i\right) = \delta\left(\omega_{[t_x,t_f]}, \delta(\omega_{[t_i,t_x]}, q_i)\right) \tag{2}$$

A distinction is made in solver algorithms between the time-stepped execution approach described above and event driven execution. In event driven execution, the simulation only proceeds when something interesting, in the experimental frame's context, happens and triggers the solver to compute the model's updated state [39, Ch. 2.3]. Each trigger event that occurs in a simulation must be explicitly tagged with a timestamp defining when in model time are the associated computations supposed to be executed [39, Ch. 2.3.2]. The realization of such a system typically involves considerations about how to assert the correct execution order maintaining the incoming events in some kind of a queue structure [33]. In addition, devising a strategy on how to resolve possible conflicts arising from simultaneous events might be required [33].

Regardless of whether the time-stepped or event-stepped approach is chosen, a careful semantic distinction must be made between model time (simulation time) t and the physical time τ (wall-clock time) that passes in the real world while the simulation is being executed [4]. For illustration, consider a dynamic model being simulated at the interval $[0, t]$ with the program taking $[0, \tau]$ in wall-clock time to execute. Obviously, the ratio t/τ can be considered a performance metric for the simulation algorithm if and only if as-fast-as-possible execution is desired [40, Ch. 2.1]. In contrast, in some other cases it will be desirable for the simulation to proceed near real-time, i.e., so that $t/\tau \approx 1$ [40, Ch. 2.1]. Most notably, these use cases include those where the simulation algorithm must interact with external actors with real-time constraints, e.g., human operators or sensors feeding in measurement data to the simulation [39, Ch. 2.2].

As a result of the simulation, a *behavior trace* $\mathcal{T} \to Q \times Y$ of the relevant model variables is produced, describing the model's evolution through time [40, Ch. 2.1]. The behavior trace can then be examined by the simulationist in order to gain insights about the properties of the simulation's target system [38]. Furthermore, it is often the case that the resulting dataset ends up being exported to another simulation tool for the purpose of conducting additional research on systems that are adjunct to that originally studied. This practice gives motivation to the discussion about co-simulation and tool integration, topics which will be separately addressed in Chapters 3.2 and 3.3.

The act of simulating many interesting systems is inherently a sequential process [39, Ch. 2.3.1]. For a simple example, consider a body moving along a straight line in 1-

dimensional space. The displacement of the body Δx_T, with respect to the initial position at time instant T is given as:

$$\Delta x_T = \int_0^T v(t)\, dt \tag{3}$$

If a closed-form expression for the function $v(t)$ exists and its integral function can be found, one is guaranteed to find an exact analytical solution for Δx_T. Unfortunately, in real-world settings such an event would be rare. Instead $v(t)$ is usually represented by discretely spaced samples. This becomes eminently true at the very moment one wishes to represent the velocity function in a computer's memory. For simplicity, let us assume the velocity is sampled at uniform intervals. Then, an approximate numerical solution for the problem could be obtained by partitioning the interval $[0, T]$ into N consecutive steps of duration n:

$$[0, T] = [0, n] \cup [n, 2n] \cup \ldots \cup [(N-1)n, Nn] \tag{4}$$

and taking the left Riemann sum:

$$\Delta x_T \approx \Delta \tilde{x}_N = \sum\nolimits_{i=0}^{N-1} v(t_i) \cdot n \tag{5}$$

wherein $t_i = in$. As Equation 5 illustrates, there clearly is no way to compute $\Delta \tilde{x}_k$ before $\Delta \tilde{x}_j$ has been computed, given that $k > j$. Hence, the only way to proceed in the simulation is the sequential process described earlier, which could in this example be formalized as:

$$\Delta \tilde{x}_i = \Delta \tilde{x}_{i-1} + v(t_i) \cdot n\,, \qquad i = 1,2, \ldots, N \tag{6}$$

The requirement for sequential solving imposes certain design constraints on the solver algorithm. This is particularly true in the context of co-simulation, wherein an ensemble of individual solver-model configurations must be orchestrated with the goal of simulating a complex system constituting of various submodels [4], [38], [40].

3.2 Co-simulation concepts and challenges

As has been established, problems usually emerge when one wishes to formulate coupled models out of the individual submodels developed under different formalisms. The driving motivation of co-simulation is that an integrated set of tools working together should streamline the design process compared to that carried out by applying the separate tools for different parts of the process [41]. On a more subtle level, the integrative approach can also be viewed as enforcing a design team to take into account the interdependences between various modeling domains [42], [43].

As proposed in [34], an ensemble of integrated tools could be achieved by means of i) introducing a superformalism (metaformalism, see [37]) that subsumes all the individual formalisms, ii) transforming the different submodels to one common formalism or iii) co-simulating the individual models within specialized, formalism-specific solvers and making the integrations only at the trajectory level (i.e., the inputs and outputs of the constituent submodels). In the first approach, challenges might arise with extensive model complexity, accompanied by relatively little gain in expressiveness in terms of the coupled model. The transforming approach is similar, with the distinction that in the general case it does not guarantee to keep the perceived expressiveness of the individual models. [34]

Generally, model integration is involved with finding an appropriate intermediary formalism from a set of possibilities, with the obvious tradeoff between the formalism's appropriateness for the individual models [34]. Herein, the "set of possibilities" can be described, as was proposed originally in [37], in terms of a Formalism Transformation Graph (FTG), an illustrative excerpt of which is depicted in Figure 2. In the FTG, vertices correspond to formalisms, blue edges correspond to possible mappings from one formalism to another, and green edges correspond to possible mappings from formalisms to possible system behavior traces (state trajectories). The formalism-trajectory transformations [40, Ch. 2.2] can be thought as simulation realizations, given a model expressed in some formalism, along with the relevant context, e.g., the input function ω (Equation 1). Hence, given a set of models and their formalisms, one can traverse the graph, resulting in a subgraph of the FTG. The subgraph will be a tree with the original formalisms as the leaves and with the common formalism as the root. The FTG's mappings are constructed so that the semantic integrity of the models is preserved. Specifically, this means that the model's behavior should not change as a result of applying a transformation. [34], [37]

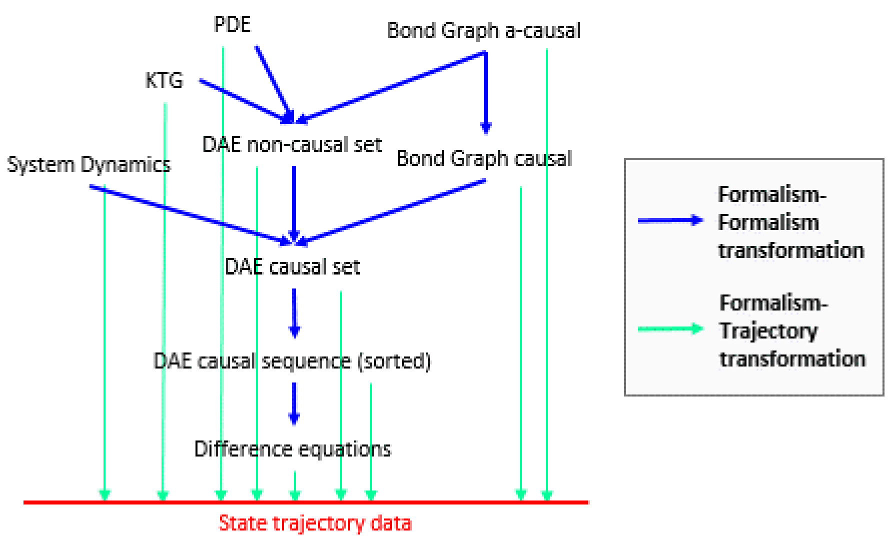


***Figure 2.** An illustrative excerpt of Formalism Transformation Graph, adapted from* [34], [37]

The third mode of model integration, co-simulation, as proposed in [34] is the one to which the bulk of this work is devoted to. The obvious challenge, in addition to establishing appropriate data formalism translations, has to do with how should one deal with synchronizing the time axes of the various models. The states of the different submodels, simulation units [44], are solved in their respective software environments and possibly also distributed to multiple workstations. A generic approach that might be utilized in a simple co-simulation scenario featuring two simulation units (SU1, SU2) is depicted in Figure 3.

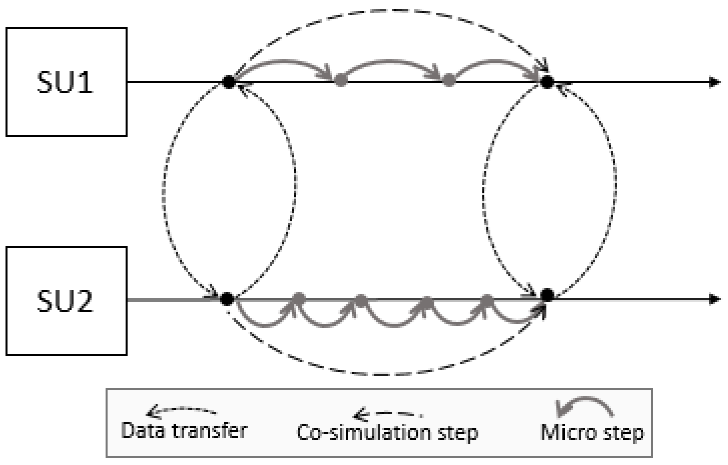


***Figure 3.** An example co-simulation timeline depicting the communication points between the simulation units, adapted from* [38]

Figure 3 essentially describes one possible mode of conservative time synchronization, as opposed to optimistic synchronization (see [45] for detailed discussion about the distinction between the two). In this approach, communication between the simulation units only takes place at distinct communication points defined by a master algorithm. At each communication point, the master should read the current outputs from the models and set the new input variables accordingly. Subsequently, the master should ask the simulators to run their respective solvers up until the next communication point. In doing this, the various models might utilize their own internal time steps (micro steps) [38], [44]. To add even further complexity, the individual solvers might use iterative approaches in order to try to ensure a convergent solution for each micro step [44]. Since the internal time steps might be considerably shorter than that of the master, it is not uncommon for co-simulation models to utilize some form of interpolation of the inputs between communication steps.

Moving on to more practical considerations regarding co-simulation of software units developed in varying formalisms, various authors have proposed different variations of *Open Tool Integration* (OTI[4]) architectures [46]–[49]. The target of these, in general, is to provide a conceptual framework within which co-simulation scenarios of the discussed variety can be planned, built, executed and evaluated. An OTI architecture that is powerful enough to represent any meaningful real-world integration scenario should be agnostic of the method by which a tool's data is delivered to the framework. The framework should only provide the engineer with a rough roadmap in order to design suitable software wrappers for tools that might expose their data in various formats, e.g., by means of file export/import, COM-based API or a GUI command interception mechanism [41]. The data source agnostic principle does not, however, imply that the tools necessarily are pure black boxes from the framework's perspective. Instead, in the most general case the data transmitted from one tool to another could also contain active objects such as pointers to subroutines which the simulation tool might be willing to expose for remote invocation [46].

Furthermore, the transformations occurring within the framework should not be limited to the simple one-to-one translation operation between two syntactic domains. For instance, it is conceivable to have a configuration where the outputs from two different tools are merged together and then translated to match the expected input format of a third one. The framework should generally allow arbitrary many-to-many workflows and

[4] The naming convention chosen here stems from the work of Karsai et al. [41], [46] but in following discussion the term "Open Tool Integration (OTI)" is used quite liberally in referral to similar architectures proposed by others as well.

provide the necessary control services for the translation, transferal and synchronization between the different operations [41]. As the number of tools increases, it is not hard to see that the interrelations amongst them can quickly become extremely complex. Some of the applicable strategies in order to facilitate the orchestrations of these intricate ensembles include [41]:

- Batch-based integration. In this approach, the experiment must be arranged in a topological workflow manner. When a “producer” tool produces a dataset, it is straightforwardly transferred to the direct “consumer” tool(s) in the chain with the appropriate semantic/syntactic translations taking place along the way.
- Transaction-based integration. Here a producer tool executes a write operation to an intermediate database to which all the tools have I/O access. A consumer tool should execute a read operation to retrieve that data.
- Notification-based integration. In this approach the subscribed consumers are explicitly notified of changes occurring in producer tools and hence they are able to update their own internal status correspondingly.

Finally, another key feature of an OTI architecture is that the logic and semantics of integrating the various tools should be, as far as possible, kept separate from the tools themselves. This approach is in contrast to various so called tool suites supplied by many commercial vendors [46]. They usually function well in sharing engineering artefacts between individual elements contained in the suite, but lack the ability to efficiently build tool chains with software external to the suite [41], [46].

3.3 Tool integration methodology

For the purposes of the present work, OTI is interpreted as a plausible approach to realize a sufficient level of modeling tool integration to serve engineering design processes in a co-simulation context. In following sections, two general approaches underlying many contemporary tool integration architectures are discussed, namely the approach based on an Integrated Data Model (IDM) and the point-to-point integration approach. The discussion then proceeds to describe Functional Mock-up Interface (FMI), a contemporary co-simulation framework supported by many commercial and open source tool vendors, which is utilized extensively also in the present work’s empirical part.

3.3.1 Generic modes of integration

A useful visualization of a generic approach to tool integration is provided in [46] which captures the main aspects of a generic OTI architecture (Figure 4). The system consists of two main components, namely an Integrated Model Server (IMS) and the Tool Adaptor (TA), sharing a common middleware communication backplate called the Common Model Interface (CMI). When a tool wants to publish data to be utilized by other tools, it is the corresponding TA's responsibility to parse that data from the tool's native format in to an intermediate network format; a low-level data structure able to express arbitrary objects and their relationships, for instance XMI or CORBA IDL. Subsequently, the data is shipped to the IMS, where it is processed by a semantic translator. In this process, the data is transformed from the intermediate network format into another form called the IDM. The IMS serves as a run-time short-term persistence repository for IDM artefacts produced by the various models. When another tool wants to access the data, the tool's TA issues a fetch command to the IMS. This triggers the relevant reverse semantic translation process, which now transforms the IDM-formatted data back to the network layer. Finally, the TA itself converts the data back to the tool's native format. [46]

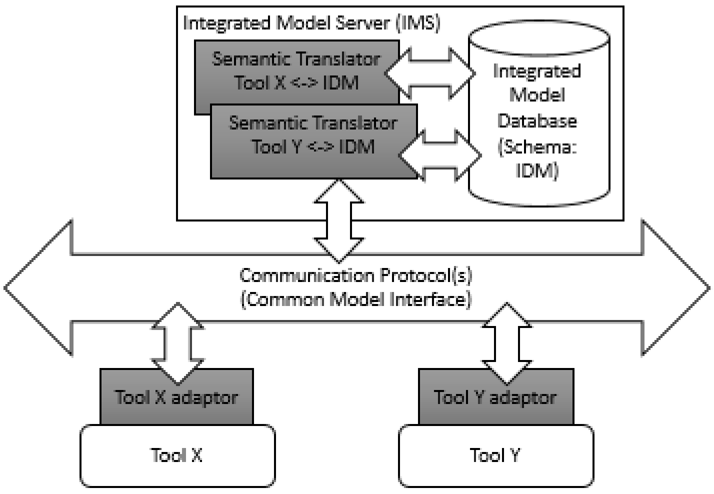

Integrated Model Server (IMS)
Semantic Translator Tool X <-> IDM
Semantic Translator Tool Y <-> IDM
Integrated Model Database (Schema: IDM)
Communication Protocol(s) (Common Model Interface)
Tool X adaptor
Tool Y adaptor
Tool X
Tool Y

Figure 4. *A generic Tool Integration Architecture based on an integrated data model, adapted from* [46]

An earlier iteration of the backplate-based OTI architecture is suggested in [47]. The ToolBus system, as the authors have dubbed their implementation, is based on a collection of processes run in parallel as well as collection of external tools. The ToolBus system exchanges both data and control signals with connected tools as well as between the individual processes. Direct communication between the tools is by definition not possible. The processes $P_1, \ldots, P_n$ themselves should not transform the data in any manner but simply provide the necessary communication routes and synchronization between the external actors. Internally, data must strictly be represented in a predetermined format called the *term* and the connected tools are expected to consume and/or produce data in this particular format. Hence, existing third party tools that do not natively have support for the data types (and control signals) used internally by ToolBus must be encapsulated in a custom software adapter in order to permit their use as a part of the integrated tool solution being developed. [47]

In [45] a multi-agent approach for co-simulation is proposed that conceptually is very similar to the OTI variants already discussed. During a co-simulation, each simulation tool is connected to a common message bus via an agent component. During the execution of a time step, each simulation tool reports the completion of the time step to the corresponding agent and subsequently pauses its operation. Each agent, in turn, further sends a message to a clock agent, that is responsible for orchestrating the total simulation. When the clock agent has received reports from all the participating agents, it can then issue a request to each representing agent to start the simulation of the next time step. [45]

In a co-simulation session, the specifics of the artefacts produced by a given tool are explicitly defined in the tool's *metamodel* [34]. The purpose of a metamodel is to define a Domain-Specific Modeling Language (DSML) for the particular tool [46], [50]. The rationale behind constructing a DSML arises from the practice of Model Driven Engineering. The DSML can be viewed as a formalism that is tailor-made to support the expression of concepts in a limited domain as efficiently as possible, as opposed to general-purpose programming languages. Hence, with the use of a DSML designers should be able to formalize descriptions of artefacts that are closer to problem domain than the implementation domain, with the additional benefits of abstracting away the often burdensome implementational syntax [51].

In the context of an OTI solution, each tool typically represents a distinct domain whose DSML must be formulated accordingly. The lingual constructs of a DSML can be utilized to precisely and exhaustively define the outputs that are anticipated to be produced by a given tool during the life cycle of an experiment [46]. Furthermore, within the tool-specific DSML other grammar of that language is included as well, i.e., the description of legal combinations of data that the tool is expected to consume [46].

Conceptually, the whole process of converting back and forth between the individual tools' native data types and the IDM type can be formulated as the relation R between the union of the attributes of individual tools' metamodels and a composite structure called the IDM. Within a system consisting of n tools, denoting the i-th tool by T_i, this yields:

$$R = \{(a,b) \in \bigcup_{i=1}^{n} \mathrm{T}_i \times IDM \mid \text{tool attribute a is represented by IDM attribute b}\} \qquad (7)$$

By definition, all the elements within the IDM set should have a corresponding element in at least one of the tools and, conversely, all the elements in a given tool's metamodel should be represented in the IDM [46], as is illustrated in Figure 5. It therefore becomes possible to formulate tool integration schemes that facilitate the sharing of different subsets of the data within different subsets of tools. One must, however, at all times bear in mind that the data being transmitted to a given tool must be natively supported by that particular tool. Equally well, on the other extreme, the framework allows one to conceive scenarios where the tools work in total isolation from one another. While this might have limited applications, a specific use case might be one where the goal of the integration is purely the synchronized execution of different tools, and not sharing of data per se.

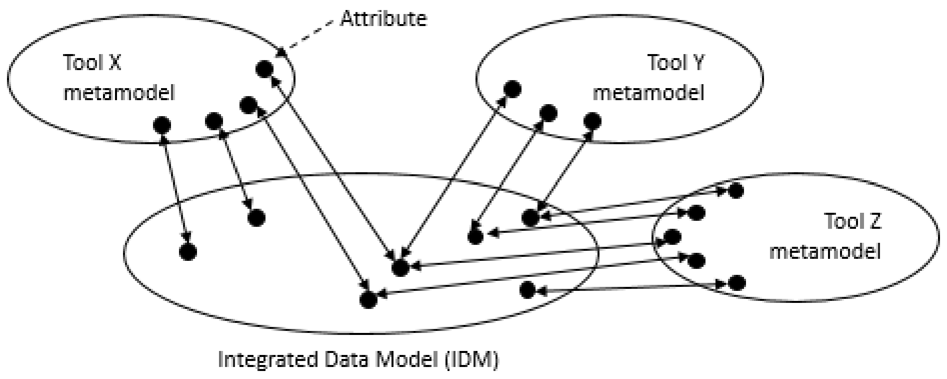


***Figure 5.** Concept of the Integrated Data Model, adapted from* [46]

Within the IDM, the attributes are uniquely tagged to facilitate the semantic conversion to and from the different tools' native data types. The main implication of this is that given a particular set of tools, the data inside the IMS is always persisted in a unified manner, agnostic of the actual tool from where it originated. At runtime, the semantic translators are able to figure out the correctness of a given dataset using the unique tags that were created by the TA at the time the data was extracted from the tool. [46]

Karsai et al. [46] proceed to present, among other discussions, an interesting alternative to OTI development with emphasis on the application-specific workflow from tool to another (Figure 6). Instead of enforcing a predefined internal data model, adapter components known as semantic translators are implemented at the points where the data and/or control flow is transitioned from one tool to another. This point-to-point approach can be useful in cases where experiment workflows are well defined and data is meant to be transferred mainly in one-to-one fashion from tool to another. [46]

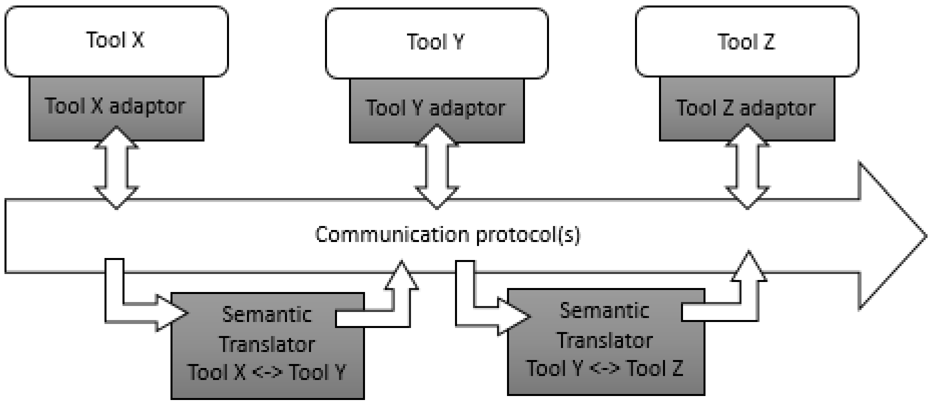

Tool X
Tool Y
Tool Z
Tool X adaptor
Tool Y adaptor
Tool Z adaptor
Communication protocol(s)
Semantic Translator Tool X <-> Tool Y
Semantic Translator Tool Y <-> Tool Z

Figure 6. *Workflow-based tool integration architecture, adapted from* [46]

So far, two main categories of OTI architectures have been identified, namely the approach based on an Integrated Data Model (Figure 4, Figure 5) and the one based on point-to-point tool integration (Figure 6). Given that a certain set of prerequisites are met, the IDM approach has the ability to achieve full integration amongst a set of tools, but on the downside, the workload can become prohibitive for large integration schemes. The IDM approach requires one to implement N bidirectional translators for N bidirectional tools. Consequently, for efficient implementation the tool number should be kept relatively small. The most beneficial use cases for IDM are where a high level of cohesion exists between the data models of the respective tools. This allows the maximally streamlined design of the individual tool adapters and translators. [52]

The point-to-point approach does not suffer from the discussed drawbacks. When the workflow is well-defined, one only generally needs to implement unidirectional translation between the producer-consumer pairs, dramatically cutting the workload. While this allows for a more manageable approach to handle longer tool chains, a tool integration solution developed in this manner obviously provides a solution to a more narrow class of problems. This is because the workflow from tool to another is fixed a priori, unlike in the IDM approach. [52]

Yet another undertaking towards advancing the purpose of simulation tool interoperability is provided by the High Level Architecture (HLA), which originally began as a development project of U.S. Department Of Defense [49], and later was adopted as an official standard of the Institute of Electrical and Electronics Engineers (IEEE) [5]. The main entities described in the standard are called *federates*, which represent individual actors taking part in the simulation task. Subsequently, a collection of federates is denoted as a *federation.* The communication between the various participants is realized via a Runtime Infrastructure (RTI). The purpose of the RTI is to offer generic services to accommodate the synchronized interoperation of the individual participants in accordance to the HLA runtime interface specification. In the HLA specification, the set of artefacts that are commonly accessible to all participants in a federation are defined in a Federation Object Model (FOM). Similarly, the set of artefacts exposed to the outside world by a single federate — which might become a member of multiple federations during its lifetime — is represented in a structure called the Simulation Object Model (SOM). While the HLA does not assume any specific semantics within the FOM or SOM, it is however enforced in that they are documented in a standard format called the Object Model Template (OMT). [33], [49], [53]

Similarly to other OTI approaches, the HLA is designed to be agnostic of internal realization of the individual federates. In fact, in HLA this idea is taken one step further, since one of the HLA's most distinctive features arises from the very definition of the possible participants in an experiment. Specifically, members of a federation are allowed to represent not only computational simulation tools, but also incoming data from real-world physical objects, and even passive listener objects that do not publish any data of their own [49]. This idea would allow the realization of various Hardware-In-Loop and even Human-In-Loop simulation scenarios.

It should be stressed that while OTI-based solutions generally aim for a high degree of reusability of generic software components, each realization of architectures similar to the examples depicted in Figure 4 or Figure 6 should still be treated as a fully separate software project. The aim is to be able to tailor the general components to construct collections of tools and processes — tool integration solutions — for a particular engineering task by carefully considering the system's requirements, and the environment in which the system will operate. In this process, an OTI framework can assist by constructing useful high-level abstractions out of some of the low-level functionalities involved in building such a system.

3.3.2 Functional Mock-up Interface

Recently, even commercial vendors have made attempts to ease the development of custom OTI schemes between various design tools. Case in point is the *Functional Mock-up Interface* (FMI), a standard tool coupling methodology whose development was initially started by Daimler A.G. FMI is a tool independent standard to support model exchange and co-simulation of dynamic models, using a combination of xml files and binary libraries exported from various tools implementing a standard set of functions. The development began in the automotive industry, where it is typical for an OEM (Original Equipment Manufacturer) to have tens or even hundreds of suppliers, all of which use their own set of design tools. Without any standardization, it is very difficult for the OEM to perform system-level design operations. [21]

The main idea is very similar to that presented in previous work done in the field of tool integration (e.g., [46]–[49]) in the sense that FMI co-simulation architecture is envisioned so that a common backplane is constructed, via standard tool adapters implementing the FMI standard, into which the individual tools can register. The FMI supports two main modes of operation, namely, Model Exchange and Co-Simulation [21]. The main use cases of Model Exchange are the export and import operations of dynamical models across tools that implement the interface. For instance, let us assume that an engineer working at an engine manufacturer company designs a dynamical model of a car engine in tool A. Being satisfied with the results, she then compiles the model (using tool A's built-in operations that support the FMI standard) in to a standard format file called the *Functional Mock-up Unit* (FMU). Then, a car manufacturer might receive that FMU and import it to tool B which supports the FMI Model Exchange standard. Now it becomes possible to perform system-level simulations and get an insight how the engine model behaves in conjunction with the other components such as the gearbox, clutch, driveline and the behavior of the (simulated) human driver.

The most important remark concerning the Model Exchange approach described above is that none of the actual computing takes place *in* the FMU that is imported to another tool. FMU units implemented with the Model Exchange variation of the standard cannot be run in stand-alone fashion. Instead, a target environment (e.g., tool B in the case of the small example above) is expected to handle, at each time step of the simulation, the actual solving (see Figure 7a) of the subsystem encapsulated in the imported FMU. The FMU merely exposes the expected input variables and the equations to be computed, which might be of the algebraic or differential variety. [21]

The FMI Model Exchange can be characterized as being a useful method for more of an ad hoc approach to tool integration between a limited set of tools. At the same time, it lacks many of the qualities of a true OTI architecture. Most importantly, it breaks the tool-backplane independence rule which states that it should be possible to integrate tools without interacting with the tools themselves. This clearly is not the case with the FMI Model Exchange approach, since for every destination tool the engineer must physically use the tool's user interface in order to perform the FMU import and the necessary configuration. As the details of this process might significantly vary between individual destination tools, it quickly becomes burdensome when the number of tools is increased.

More closely related to an actual OTI architecture is the FMI Co-Simulation standard (Figure 7b). Herein the subsystems are solved independently from one another by application logic that is encapsulated into the FMU package itself as a collection of shared executable libraries. The control of FMU ensembles takes place by means of a backplate-like master algorithm, whose responsibility is to control the data exchange and the convergence of the total simulation result. In general, what makes this task difficult is the fact that the individual subsystems might exhibit very different characteristics in terms of real-world execution times, and the master algorithm must be able to cope with this asynchronous behavior. Furthermore, FMI Co-Simulation only offers a partial solution to the general OTI problem, since the actual implementation of master algorithm is not explicitly defined within the FMI specification but instead it is on the large left to the operator's hands. [21]

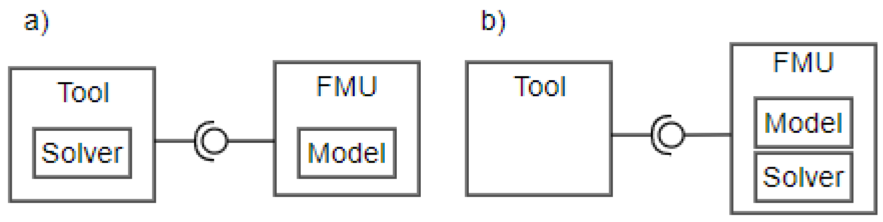


Figure 7. *Functional Mock-up Interface for a) Model Exchange b) Co-Simulation, adapted from* [54]

FMI Co-Simulation implicitly supports a third mode of operation, co-simulation with tool coupling (Figure 8). Herein from the integrator's perspective, the FMU can still be used in a very similar manner to the standard co-simulation scenario. The difference is that the internal implementation of the FMU does not directly contain any means of computing the model's state. Instead, it merely acts as a wrapper and provides a custom communication protocol that is not a part of the FMI specification and must be implemented in a tool-by-tool fashion. The intent is that as the subroutines defined in FMI specification are requested from the wrapper component by the master algorithm, the necessary commands are parsed and further relayed to an external tool. As such, the FMU itself actually has the capability to act as a translator between different semantic domains in the co-simulation tool chain. The prerequisite for using this approach is, of course, that the external tool that the wrapper targets must be available at runtime. In most cases, this means that the tool must be installed on the local system or it must be accessible via a network socket, and the appropriate licenses must be checked out. [21]

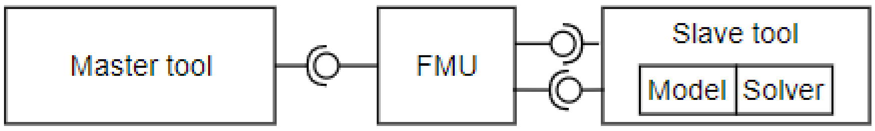


Figure 8. *FMU Co-Simulation with tool coupling, adapted from* [21]

The FMU, being the concrete object that implements the FMI standard, encapsulates both the compiled binaries that can be natively run on the target machine and an XML description of the input/output arguments of the standard set of FMI subroutines embedded in the binaries. In the file system, the compiled FMU which a simulation tool outputs is represented as a single file with the .fmu suffix, but the file actually is a zip archive that can be extracted by standard archiving tools. For FMU's normal operation, manually extracting the contents is not usually required. However, as it will turn out in the empirical part of the present work, this feature becomes very useful for debugging and/or exploratory purposes.

An FMU exposes its internal artefacts (variables and operations) by means of a standardized set of C-functions, which are packaged into a shared library (e.g., a .dll file in a Windows environment or .so in Linux) nested in the FMU. Alongside with the binaries, a "modelDescription.xml" file is distributed. The purpose of the modelDescription.xml file is to provide the target system with a precise description on how the C-functions can be invoked, which parameter configurations and data types are legal, what are the available attributes that can be read or written to, and what is the expected output of each and every function. In addition to model inputs and outputs, the FMI standard supports the exposition of model's tuning parameters and variables' partial derivatives as well, if any are present, making it possible to build optimization tasks around FMU models. [21]

During an FMU Co-Simulation session, communication between the various FMUs (or "tools", in a more generic OTI context) takes place at discrete communication points, similarly to what was depicted in Figure 3. It is the job of the master algorithm's (the "backplane's") designer to implement suitable operations for facilitating the data exchange between the individual FMUs and, additionally, design the functionality that properly synchronizes the simulation results of all the subsystems involved, so that the overall simulation task can proceed from start to finish in a stable manner [21]. A naive, yet in simple cases effective approach to constructing a master algorithm could be something like the following. Let $i \in \mathbb{N}$ be an integer designating the index of the current time step, $i_{max} \in \mathbb{N}$ the index of the time step when the simulation is halted and $\Delta_t \in \mathbb{R}^+$ be the real-world time in seconds that passes between communication points:

```
while $i \leq i_{max}$ do:
   pause the execution of all the connected slave FMUs
   collect the output $y_t$ from each of the slaves

   based on $y_t$ figure out what the inputs $u_{t+1}$ should be for
   each submodel

   invoke the respective C-functions to set the input
   variables to $u_{t+1}$ for each model

   unpause the slaves
   set $i := i + 1$
   pause the master algorithm for $\Delta_t$ seconds
```

Listing 1. *Pseudocode for a potential master algorithm of a FMU Co-Simulation scenario, adapted from* [21]

Following this discussion, a nontrivial implication for designing FMI Co-Simulation compatible tools is that the tool must be able to interrupt and subsequently resume its operation with updated input data. Moreover, care must be exercised when deciding what should the global time step size be in the master algorithm. In the general case, it is by no means guaranteed that computation taking place in the individual FMUs can reach convergence in a timeframe that is even remotely similar. Hence, the master algorithm should explicitly define the procedure taken in the case of varying computational complexities and wall-clock running time of the individual FMUs. A more sophisticated master algorithm could probably include features requiring the individual models output some kind of convergence criteria, and if need be, repeating the (partially) failed time step with an increased time window in order to reach simulation stability.

Regarding these implementational details, the designer of the master algorithm has relatively free hands so long as the general allowed call sequences are followed. In general, the joint entity consisting of a master algorithm and a single FMU can be interpreted as a state machine in which state transitions are only possible via a predetermined set of functions (Figure 9). The main simulation loop takes place within the `slaveInitialized` block which can only be entered and exited through a prescribed set of compounding stages, triggered by a prescribed set of events. It should be remarked that the state machine behavior is a required, but not sufficient feature of the master algorithm since it does not tell anything about coordinating the execution of a set of multiple FMUs in relation to one another.

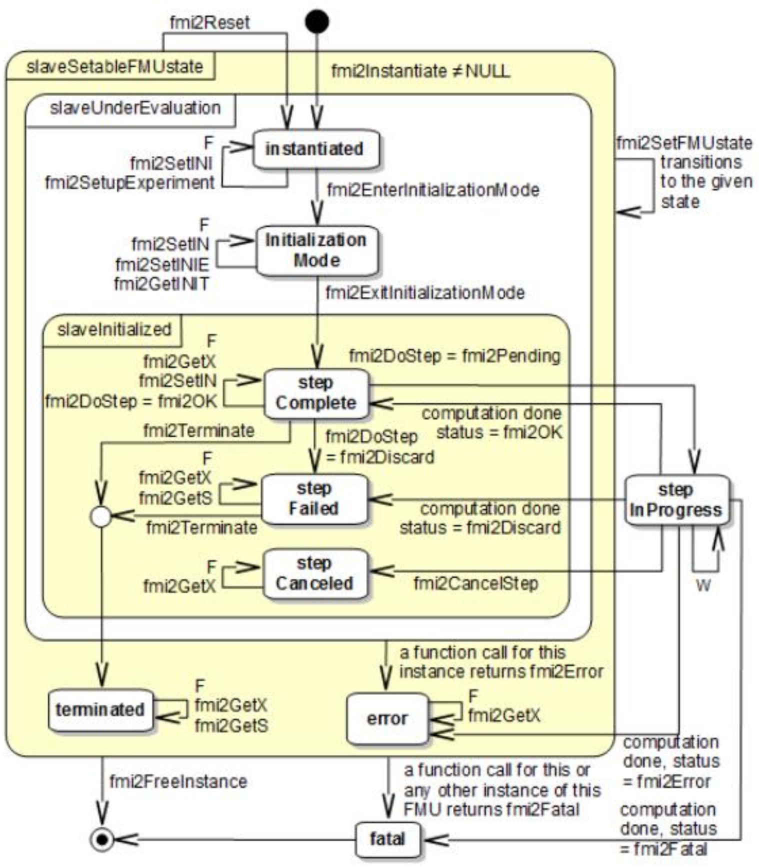


Figure 9. *State machine representation of co-simulating a single FMU, reproduced under CC BY-SA 4.0 from [21]*

3.4 The Digital Twin

The act of simulation has been established as observing, in the context of a given experimental frame, the behavior of a dynamic model in time. By this justification one could, at least informally, envision the experimental frame having some kind of mapping to a segment of the timeline in the life cycle of the physical system being simulated. By extension, simulation scenarios where the entire life cycle of a physical system *is* the experimental frame are also conceivable. Simulation setups of such nature are, in the context of the present work, referred to as digital twins.

To illustrate the concept, consider the process of designing and manufacturing a physical product, for instance, a car or a spacecraft. Depending on the exact industry, the properties of the individual end products coming out of the factory pipeline will have some degree of variance involved. This could be either due to purposeful customizations imposed by the end customer, the intended mission and operating conditions of the product and/or small, uncontrollable fluctuations occurring in the manufacturing process. Alongside with delivering the tangible product into existence, a software counterpart, a digital twin, of that unique product could be instantiated in a computational environment. The semantic appearance of this digital twin should then, to the extent possible, reflect the variance involved in the design and manufacturing process. [7]

The discussed examples can be considered realizations of *mass customization*, the notion of establishing manufacturing concepts that are able to achieve low unit costs for customized products [17]. As today's design and manufacturing systems already are highly digitized, each unique instantiation of a product in the physical world leaves behind a trace in the cyber realm; a *digital shadow* that entails all the operation, condition, process etc. data produced during that instantiation [12]. The digital shadow of an individual product instance can, then, be linked with a *digital master model* — another digital artefact that comprises of data and functionality that is common to all instances of a given product class [12]. In this manner, a new digital twin is born.

Equipped with state-of-art phenomenological models that are able to simulate the various subsystems present, the instantiated digital twin should now have the ability to "experience" every possible event that its physical counterpart might confront during its lifetime. This could include operating conditions that were not originally considered at the time the initial designs of the product were made. In doing this, the ensemble of simulations which makes up the computational part of the digital twin is continuously fed with sensor data received from the physical device. In the context of industrial systems that are prone to component failures, this could allow the development of unforeseen applications of predictive maintenance, component lifetime forecasting and even the dynamic activation of various self-healing mechanisms (provided that they exist). [7]

As discussed briefly in Chapter 2, a digital twin has the definitive properties of being extremely accurate and always up-to-date. This goes to the extent that a digital twin can be used as a direct surrogate for reasoning about events occurring in the physical world in real time. As such, the concept has historically inspired texts with a rather generic (and, arguably, metaphysical) approach (see, e.g., [55]). Be that as it may, one of the recent attempts to formulate a more technical definition for a digital twin is that of Glaessgen and Stargel:

"A Digital Twin is an integrated multiphysics, multiscale, probabilistic simulation of an as-built vehicle or system that uses the best available physical models, sensor updates, fleet history, etc., to mirror the life of its corresponding flying twin." [7]

Reportedly, the National Aerospace and Space Administration (NASA) has a long history of utilizing the twin approach that dates back to the first Apollo missions. Obviously, back then the computing power was not available to support very accurate digital representations of space systems, but instead physical replicas were used. This would allow the flight control personnel back on earth to try out various solution scenarios for any issues that might have occurred in-flight on the actual mission, before relaying the instructions to the astronauts. Moreover, the twins served an important function in pre-flight training. With the obvious monetary implications of constructing accurate physical twins of extremely complex engineering artefacts such as spacecraft, the concept of digital twin emerged in NASA's own studies as the availability of computational capabilities to conduct numerical simulation improved. [14], [15]

Another definition of a digital twin is given by Stark et al. as:

"A Digital Twin is the digital representation of a unique asset (product, machine, service, product service system or other intangible asset), that compromises [sic] its properties, condition and behavior by means of models, information and data" [12]

Despite the somewhat orthogonal nature of the two definitions, they both have an obvious overlap between the concepts of representativity, inspectablity and contemporality discussed in Chapter 2. Most importantly, the notion of a digital twin being a counterpart of some physical object ("as-built vehicle", "unique asset") is highlighted in both definitions. The definition of Stark et al. expands the concept of object somewhat, arguing that also "intangible assets" such as services could be represented by digital twins. Both definitions reflect also the concept of contemporality ("mirror the life…", "compromises its … behavior…"). The inspectability property is perhaps harder to interpret directly from these definitions, but can instead be thought of as an underlying assumption — an intrinsic property of *any* digital artefact versus a physical one. Software can, as opposed to physical machines, be designed in a manner that allows users to freely observe the internal state of an execution at any point in time without affecting the process itself. Clearly, the same cannot generally be said for physical machinery, hence providing another argument for the use of digital twins.

The digital twin concept can be viewed as extending the temporal window of simulation's utilization to the whole life cycle of the product. Whereas traditionally simulation was viewed as a tool for mainly model-based design operations in R&D, the digital twin paradigm allows the simulation capabilities to be utilized during the delivery, operational life and even decommissioning phases as well. This can be viewed as a natural continuum for how the role, and also, the general user population, of simulation in organizations has evolved over the recent decades (Figure 10). Specifically, beginning in 1960's as a highly specialized set of techniques, computer simulations were mainly conducted by experts in computer science and mathematics. Today, computer simulations are conducted extensively by engineers in various disciplines, thanks to highly specialized tools that support model-based design activities. With the introduction of digital twins, instead of solving individual engineering problems, simulation is envisioned to become a core functionality of the organization. This is realized through constructing a ubiquitously connected network of digital and physical assets that support operation and service with direct linkage to operation data. [15]

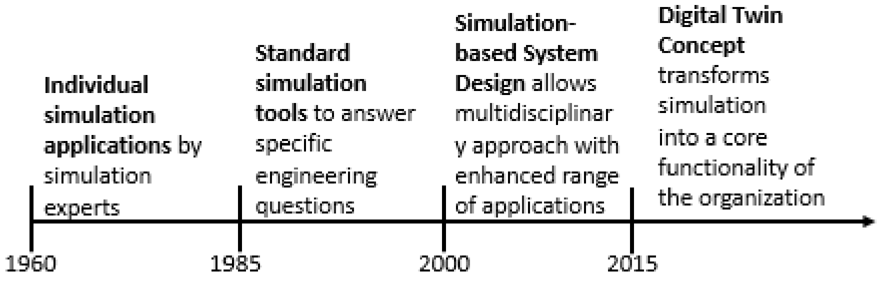

Individual simulation applications by simulation experts
Standard simulation tools to answer specific engineering questions
Simulation-based System Design allows multidisciplinar y approach with enhanced range of applications
Digital Twin Concept transforms simulation into a core functionality of the organization
1960
1985
2000
2015

Figure 10. *The timeline of simulation paradigm's evolution in engineering organizations, adapted from* [15]

In context of co-simulation, a digital twin can be characterized as a specific kind of co-simulation arrangement. Specifically, recall the discussion in Chapter 3.3.1 wherein co-simulation was described as an arrangement of simulation tools communicating through a common messaging bus, utilizing tool-specific translator and/or agent components. As discussed briefly in, e.g., the HLA specification, the participating components of a co-simulation session could be pure software, but equally well sensory measurement streams originating from real-world objects. Correspondingly, a description of a software framework capable of hosting digital twin applications could be realized through defining two distinct classes of agents, i.e.:

1. **Process agents**: entities that interact with process data originating from physical objects or processes and translate that data to and from an intermediate formalism interpretable by other agents.
2. **Simulation agents**: entities that interact with simulation units and translate their data to and from an intermediate formalism.

It must be noticed that the representativity criterion does not directly translate into the functional requirement of a digital twin framework that the connection between process agents and simulation agents could be always constructed as a simple one-to-one mapping [16], [17]. One reason for this emerges from the discussed multidisciplinary nature of modeling an accurate representation of a complex system. The modeling of an engineering system of any considerable complexity typically involves an array of specialized simulation tools, resulting in a set of co-simulation units rather than a large monolithic executable. Similarly, it would be an oversimplification to suggest that all data coming in from a physical entity could be appropriately handled by a single process agent. The interaction and translation capabilities of process agents obviously would need to be implemented case-by-case towards the varying interfaces that different portions of that entity could expose to outside world.

Following this discussion from a software engineering point of view, the generic outline of a producer-consumer arrangement emerges. Herein a digital twin could be interpreted as the relation R_{DT} between a set of process agents P and set of simulation agents S:

$$R_{DT} = \{(p, s) | (p, s) \in P \times S\ ,\ s \text{ consumes data produced by } p\} \tag{8}$$

Clearly, the simplified definition given in Equation 8 does not capture the full space of conceivable interrelations between the co-simulation units in S. In the general co-simulation scenario, it is obvious that simulation agents may not only receive data from the process agents in P, but from other simulation agents equally well. Similarly, in this discussion control-oriented approaches are omitted, i.e., those where nodes in P may receive data from some subset of nodes in S in addition to transmitting it. Nevertheless, with these restrictions in mind, Equation 8 serves as a mental backplane for the software architecture discussed in the empirical part of the work, wherein an object-oriented approach is proposed to represent and implement the generic behavior patterns of process and simulation agents, respectively.

4. MODELCONDUCTOR ONLINE CO-SIMULATION LIBRARY

This chapter describes the main outcomes of the empirical part of the work, wherein a constructive approach was utilized in order to gain insights to the research objectives. Following the conclusions established in Chapter 3.4 regarding the general ontology of a digital twin, a prototype software solution is described which is aimed at allowing the orchestrating of arbitrary many-to-many relations between two distinct types of software agents, i.e., those interacting with physical processes and those interacting with simulation models.

4.1 Overview

A prototype of a Python digital twin library called *ModelConductor* [32] was developed during the work. The library is a Minimum Viable Product (MVP) in the sense that it captures the main functionalities along with the relevant unit tests, but has not yet been rigorously integration tested against all possible use scenarios. The software architecture is based fully on an Object Oriented Programming (OOP) approach, so that the high-level classes can easily be extended to support even more different kinds of input streams and models.

An early draft of a use case diagram for the developed software is presented in Figure 11, which describes the system and its actors. Not all the features were implemented during the scope of the thesis, but the figure illustrates the software architecture's main paradigm at high level, based on the co-operation of three distinct classes of actors:

- Data sources (the `MeasurementStreamHandler` class) representing **physical assets** and process data streams originating therein.
- Data consumers (The `ModelHandler` class) representing **digital asset models** that simulate or make inferences of the behavior of their corresponding data sources.
- The human operator.

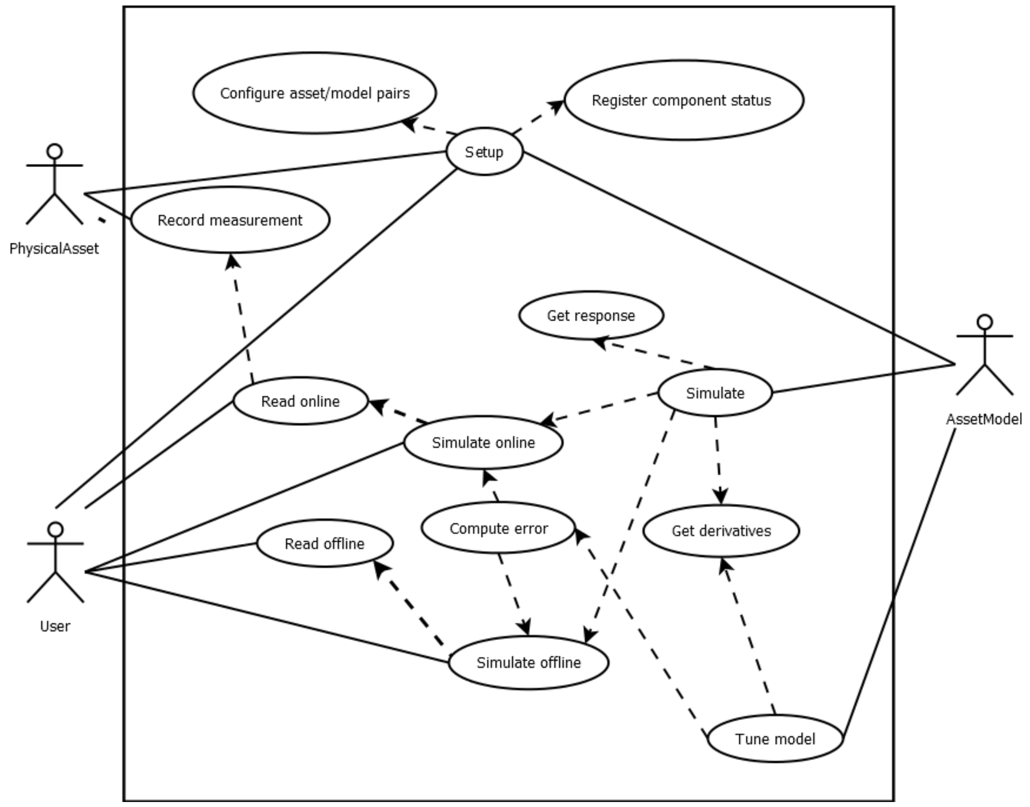


Figure 11. *An early design phase use case diagram of ModelConductor*

In particular, the proposed MVP provides a solution to the problem of concisely building online data pipelines between the data sources and data consumers, providing the necessary formalism translations along the way (Figure 12). Specifically, this is realized via a variable length queue object containing measurement objects originating from a data source, waiting to be processed. The solution allows a degree of temporal independence between the data sources and data consumers, which can be operated in an asynchronous manner to accommodate the co-operation of components that might have very different data throughput rates.

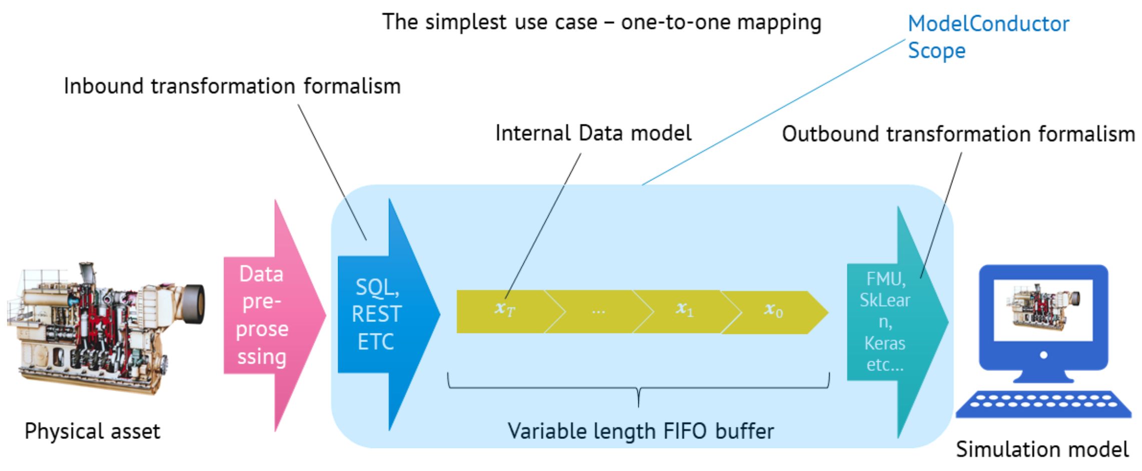

The simplest use case – one-to-one mapping
ModelConductor Scope
Inbound transformation formalism
Internal Data model
Outbound transformation formalism
Data pre-prosessing
SQL, REST ETC
x_T
...
x_1
x_0
FMU, SkLearn, Keras etc...
Physical asset
Variable length FIFO buffer
Simulation model

Figure 12. *The general ModelConductor scope*

4.2 Structure and description of internal data types

As discussed, in general, between the data sources (physical assets from which measurements are derived) and data consumer asset models a many-to-many relationship exists. In OOP terms, this relationship is constructed via a separate `Experiment` entity which is responsible for, among other things, storing the data routes from measurements to models for a given invocation of the software (Figure 13).

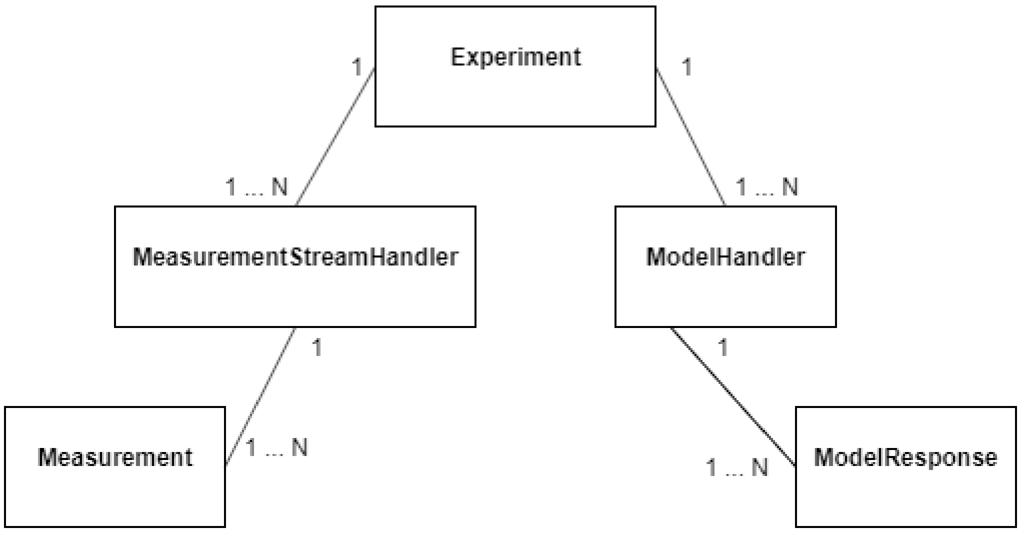


Figure 13. *Specification-level class diagram of the ModelConductor library.*

Within ModelConductor, all data that is to be exchanged from one data source / data consumer to another, is asserted to exists in one of two possible formats: i) a `Measurement` object that denotes a timestamped data structure that is received from a *physical* asset, best understood as a snapshot of the asset's state at a given time or ii) a `ModelResponse` object that similarly denotes a timestamped data structure that is received from a *digital* asset, a snapshot of the model's virtual state at a given time.

Generally, for each `Measurement` object that is generated from data received from a physical object, exactly one `ModelResponse` object is generated in the paired digital object(s). Concretely, both `Measurement` and `ModelResponse` are key-value collections. A variable (a key) may (but is not required to) belong to one or more variable categories based on the variable's purported function inside the ModelConductor ecosystem. In the current version, the following categories are implemented:

- Input keys: The subset of keys in `Measurement` object corresponding to the keys that are expected as inputs by the relevant `ModelHandler`.
- Target keys: The keys that are expected to be output from a `ModelHandler` instance onto a `ModelResponse` object.

- Control keys: A subset of keys in `Measurement` that is not intended to be used as an input to a `ModelHandler`, but rather as a validation variable against the `ModelHandler` output.
- Timestamp key: A key denoting the instant when the relevant `Measurement` or `ModelResponse` instance was created.

In practical applications (most prominently when working with FMU models), simulation models might counterintuitively choose to *output their inputs*, as well as the actual model responses to the input signals. In these instances, the `ModelResponse` object may actually contain the model's internal representation of some subset of the relevant `Measurement` object that was given to that model. This is a useful feature since it allows one to validate that the input variables are perceived in the model's world in a coherent manner. However, this can also create confusion. Situations can occur where a subset of keys intersects `Measurement` and `ModelResponse`, but they represent two fundamentally different things, or more accurately, two separate representations of the same thing. Obviously, if the model works correctly, the difference should not be very large.

In an attempt to keep behavior consistent between different subclasses of `ModelHandler`, the code implementation asserts in a helper method that each `ModelResponse` contains as a subset either the original `Measurement` data or, alternatively, a representation thereof obtained directly from the simulation model. While this approach definitely has some pitfalls involved, it turns out that advantages can be gained in, e.g., the simplified implementation of logging functionalities, as will shortly be illustrated.

At an implementational level, the `MeasurementStreamHandler` and `ModelHandler` are parent classes that only provide low-level services for input-output functionality. In practice, this means that they must be extended separately for each technology for which interfacing is desired, similarly to the semantic translator concept discussed in Chapter 3.2. An example is provided in Figure 14, depicting the inheritance hierarchy of `MeasurementStreamHandler`. Currently, the two main modes of getting the measurement data into ModelConductor are periodically polling a database and instantiating a TCP socket connection between a remote client and ModelConductor. As it is observed, both of these modes have their separate implementations, i.e., `MeasurementStreamPoller` and `IncomingMeasurementListenerClasses`.

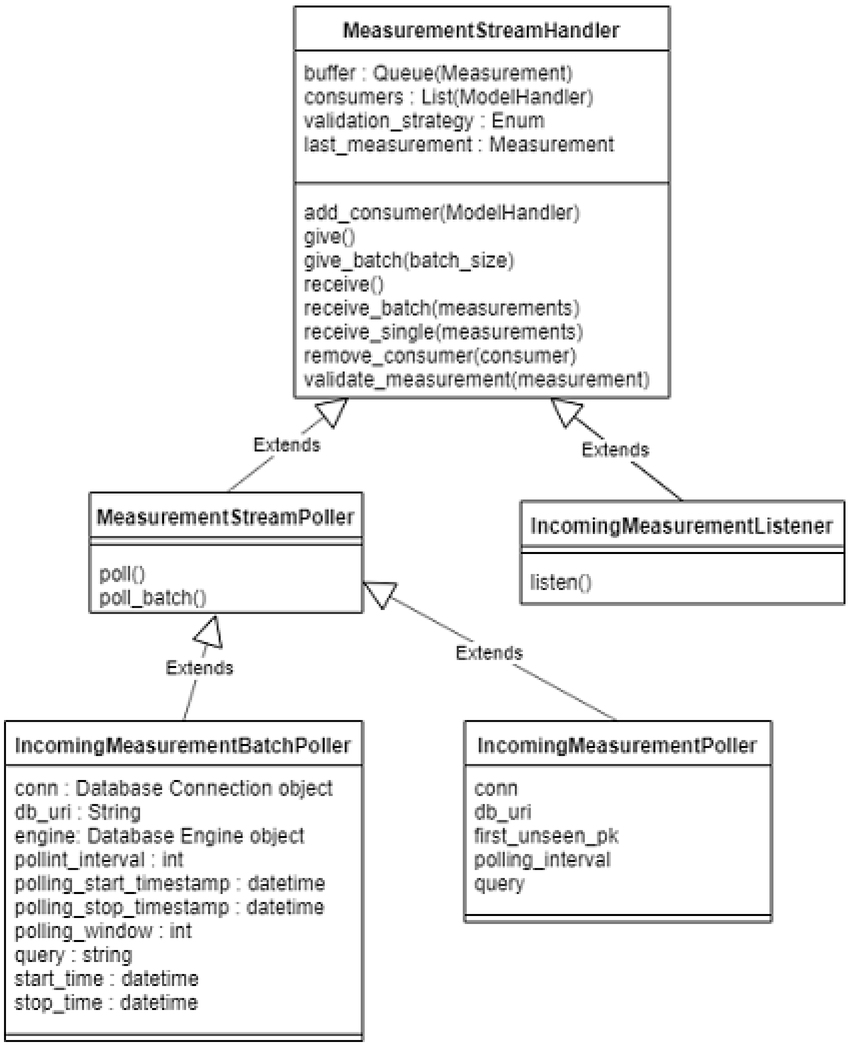

MeasurementStreamHandler
buffer : Queue(Measurement)
consumers : List(ModelHandler)
validation_strategy : Enum
last_measurement : Measurement
add_consumer(ModelHandler)
give()
give_batch(batch_size)
receive()
receive_batch(measurements)
receive_single(measurements)
remove_consumer(consumer)
validate_measurement(measurement)
Extends
Extends
MeasurementStreamPoller
poll()
poll_batch()
IncomingMeasurementListener
listen()
Extends
Extends
IncomingMeasurementBatchPoller
conn : Database Connection object
db_uri : String
engine: Database Engine object
pollint_interval : int
polling_start_timestamp : datetime
polling_stop_timestamp : datetime
polling_window : int
query : string
start_time : datetime
stop_time : datetime
IncomingMeasurementPoller
conn
db_uri
first_unseen_pk
polling_interval
query

Figure 14. *Class diagram of MeasurementStreamHandler and descendant classes*

A similar hierarchical structure to that depicted in Figure 14 is present in the `Model-Handler` class. Currently, the simulation of models exported as Functional Mock-up Units (Chapter 3.3.2) from a compliant modeling tool [6], as well as machine learning models implemented with the scikit-learn library are supported. Figure 15 illustrates the `ModelHandler` structure.

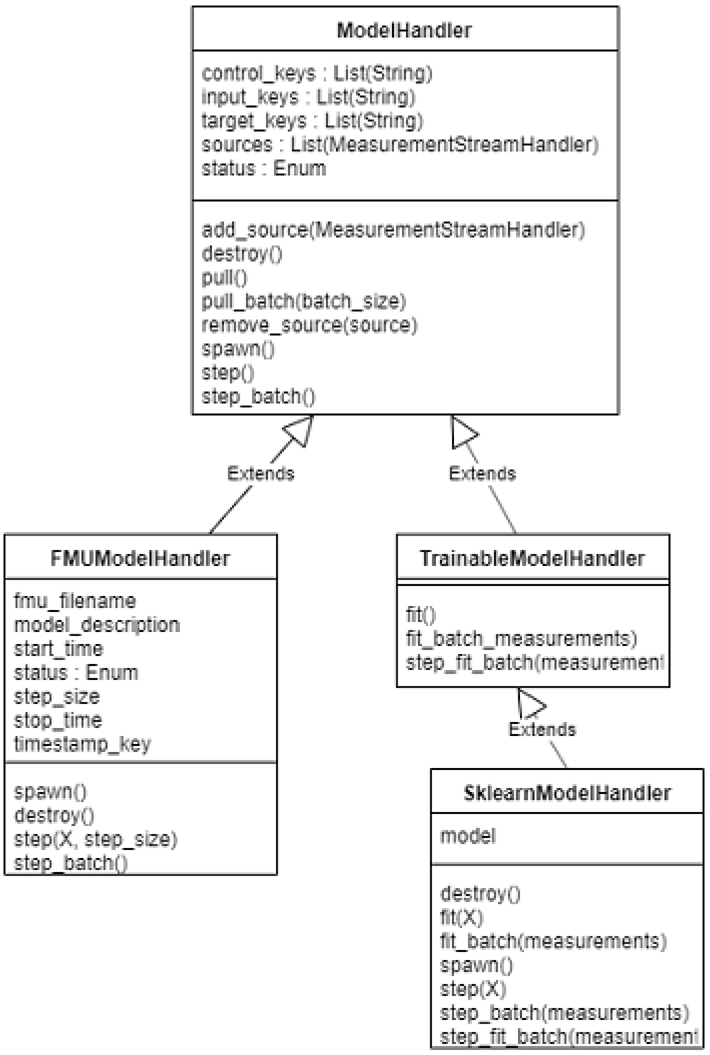


Figure 15. *Class diagram of ModelHandler and descendant classes*

4.3 Behavior and interactions

From an early stage in development, it was clear that the two main portions (i.e., `MeasurementStreamHandler` and `ModelHandler` classes) needed to be designed to accommodate asynchronous and parallel operation between each other. The reasons for this become evident when one considers how real-world data might actually be collected from a physical process and, on the other hand, how the computational complexity and, hence, execution time might vary between various types of process models. Basically, this means that in order for the software to have any potential for general applicability, it should be flexible enough to accommodate for situations where the arrival rate of data is very different from the rate with which that the data can be used.

From the discussion above, a multitude of design constraints arise. Most importantly, if a model is meant to replicate a real-world process as closely as possible using sensor data as input, a simulation step can only be performed when the necessary data is available in its entirety. For a given model, the format of this data must be explicitly defined in order for the software to understand whether this criterion is met at a given point in time or not. Conversely, even if input data is available, it cannot be fed forward to the model while the model is occupied computing a simulation step for a previous input. Consequently, the solution was constructed so that model inference operates on a “pull” manner at the command of the associated model, additionally constrained on the fact that new input data is available at the time issuing the command.

The most simplistic use case occurs when exactly one measurement source is paired with exactly one model. Essentially, this boils down to two loops that are executed in parallel threads. In the first loop depicted in the sequence diagram of Figure 16, the receive method of the `MeasurementStreamHandler` object is called starting the execution of the main loop of the data receiving thread.

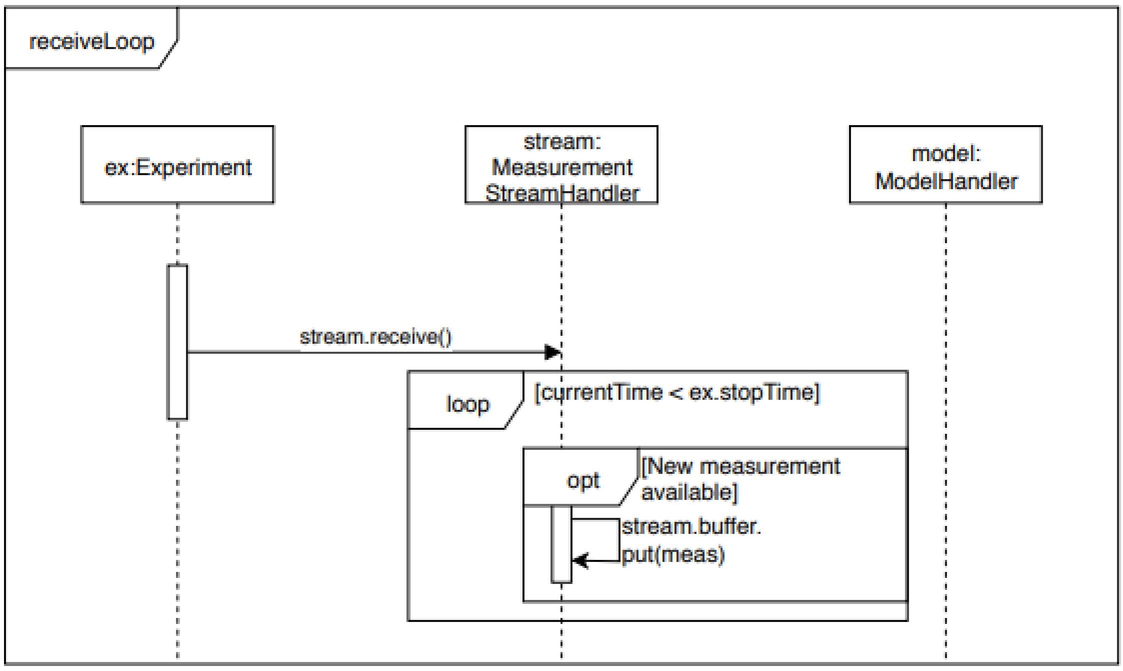


Figure 16. *The main data receiving loop of Experiment*

On the abstract level, at each iteration the loop now checks whether there is new data available on the associated input data stream, and, if yes, puts the data into a buffer, which is concretely a FIFO (First In, First Out) queue object. The implementation of checking the “new measurement available” rule will vary according to the concrete data source, which could be, e.g., a database or a stream of incoming data via a REST API. For instance, if the method of getting the data is to periodically poll a database, such an implementation might choose to suspend the thread for a predefined time at each iteration of the polling loop in order to limit the number of queries executed. Finally, just before putting the data in to the queue, the implementing classes might choose to do some additional pre-processing, e.g., remove parts of the data that are unnecessary for the experiment at hand. The data itself is treated as a simple key–value collection called `Measurement` by extending it from Python's native `dict` class and adding some additional application-specific methods, such as converting the key–value pairs to plain numerical arrays, which can be utilized in some models, most importantly of the machine learning variety.

Simultaneously, another thread depicted in Figure 17 is executed which is set up to listen to incoming data in the buffer.

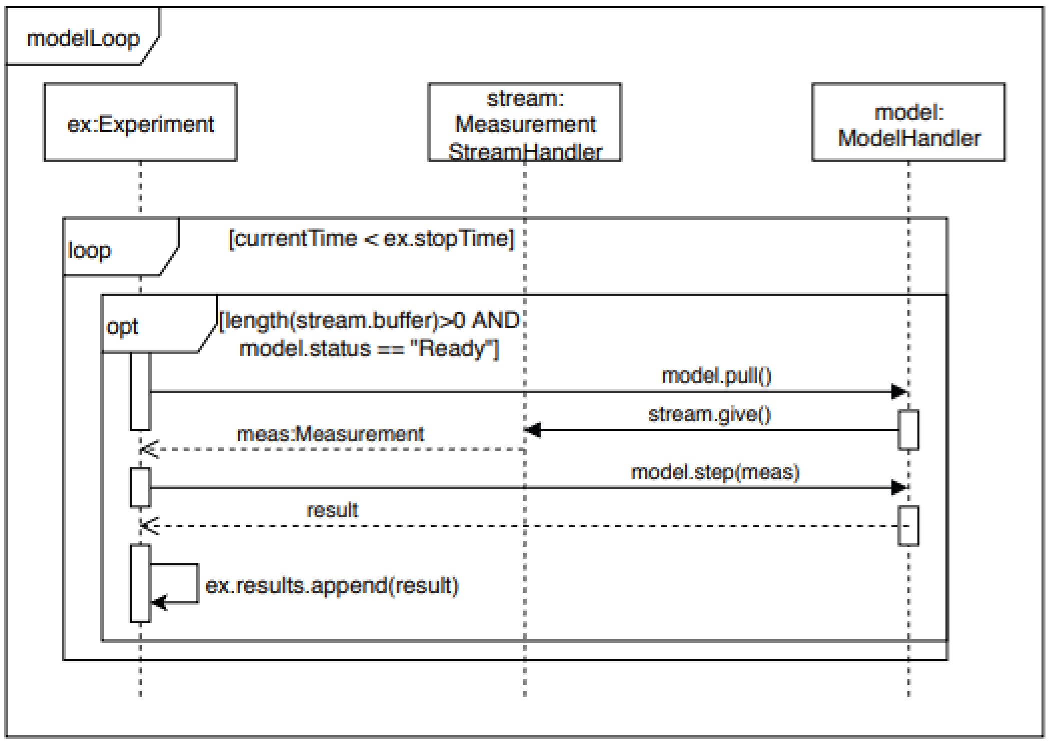


Figure 17. *The main model loop of Experiment*

At each iteration the loop checks whether there is at least one element in the buffer. If that is true and also the model is in "ready" state so that new data can be fed in to the model, a `Measurement` element is removed from the buffer in FIFO manner and used to make an inference from the associated model. The result — a `ModelResponse` object — is then appended to another list, an attribute of the `Experiment` object.

Putting it all together, Figure 18 illustrates what a single invocation of a one-source-to-one-model online experiment might look like. Most importantly, before the main loops can be executed it is the responsibility of the operator to make sure that the appropriate stream and model objects exist, that they are in a "ready" state, and to set up various experiment parameters such as the running time. Subsequently, at the simple call of the `run` method, the experiment will run for a predetermined time, after which both of the threads are terminated and the final results list is returned.

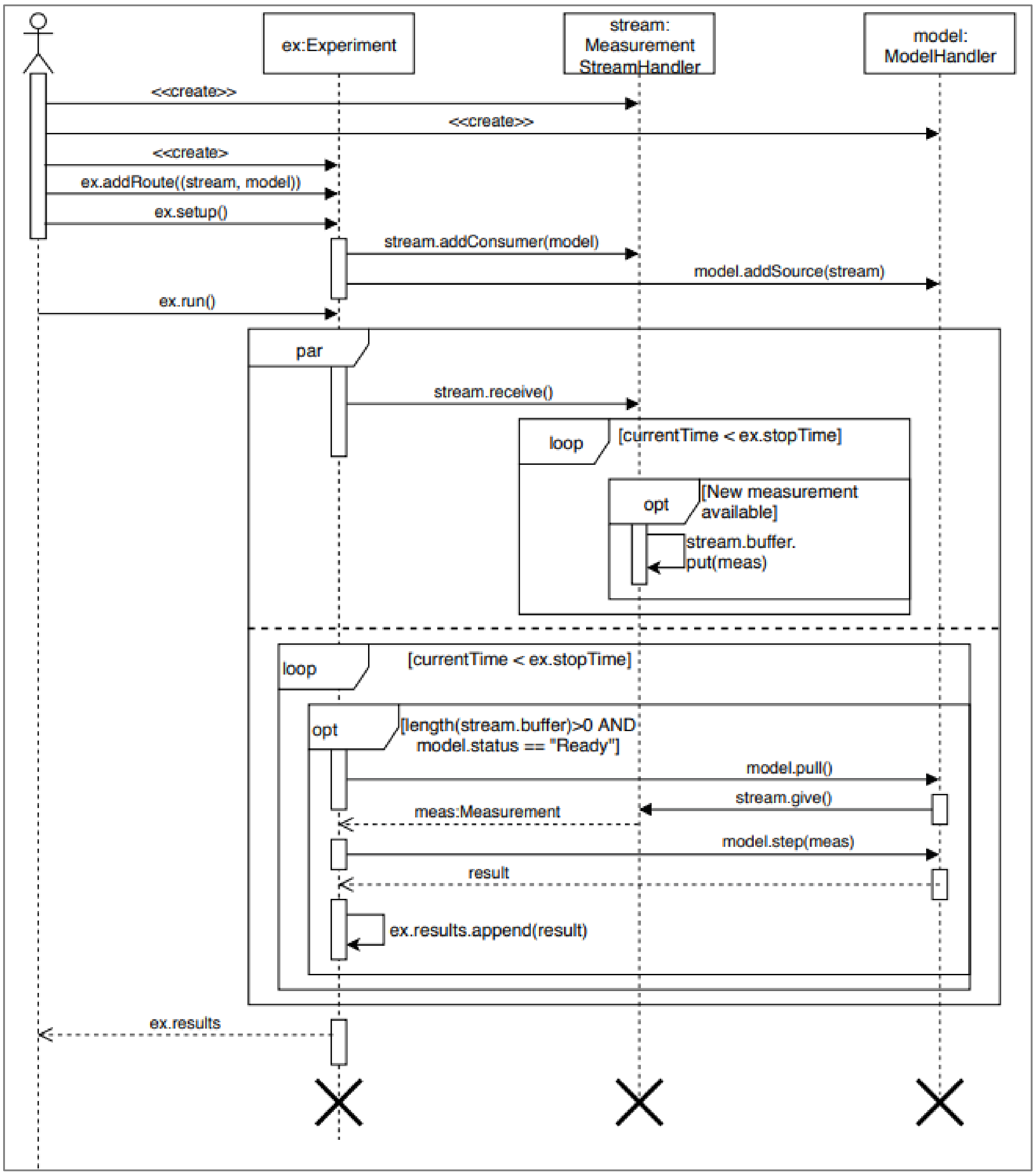

ex:Experiment
stream: Measurement StreamHandler
model: ModelHandler
<<create>>
<<create>>
<<create>
ex.addRoute((stream, model))
ex.setup()
stream.addConsumer(model)
model.addSource(stream)
ex.run()
par
stream.receive()
loop
[currentTime < ex.stopTime]
opt
[New measurement available]
stream.buffer.put(meas)
loop
[currentTime < ex.stopTime]
opt
[length(stream.buffer)>0 AND model.status == "Ready"]
model.pull()
stream.give()
meas:Measurement
model.step(meas)
result
ex.results.append(result)
ex.results

Figure 18. *An illustrative sequence diagram of a one-to-one experiment use case in ModelConductor*

The ModelConductor framework is designed so that it can be extended to support arbitrarily complex multi-input, multi-output scenarios by means of nested threaded executions between the source–model tuples, as Figure 19 will illustrate. Concretely, the concurrence implied by the *par* frame in the program is achieved by utilizing Python's native Thread module [56], [57].

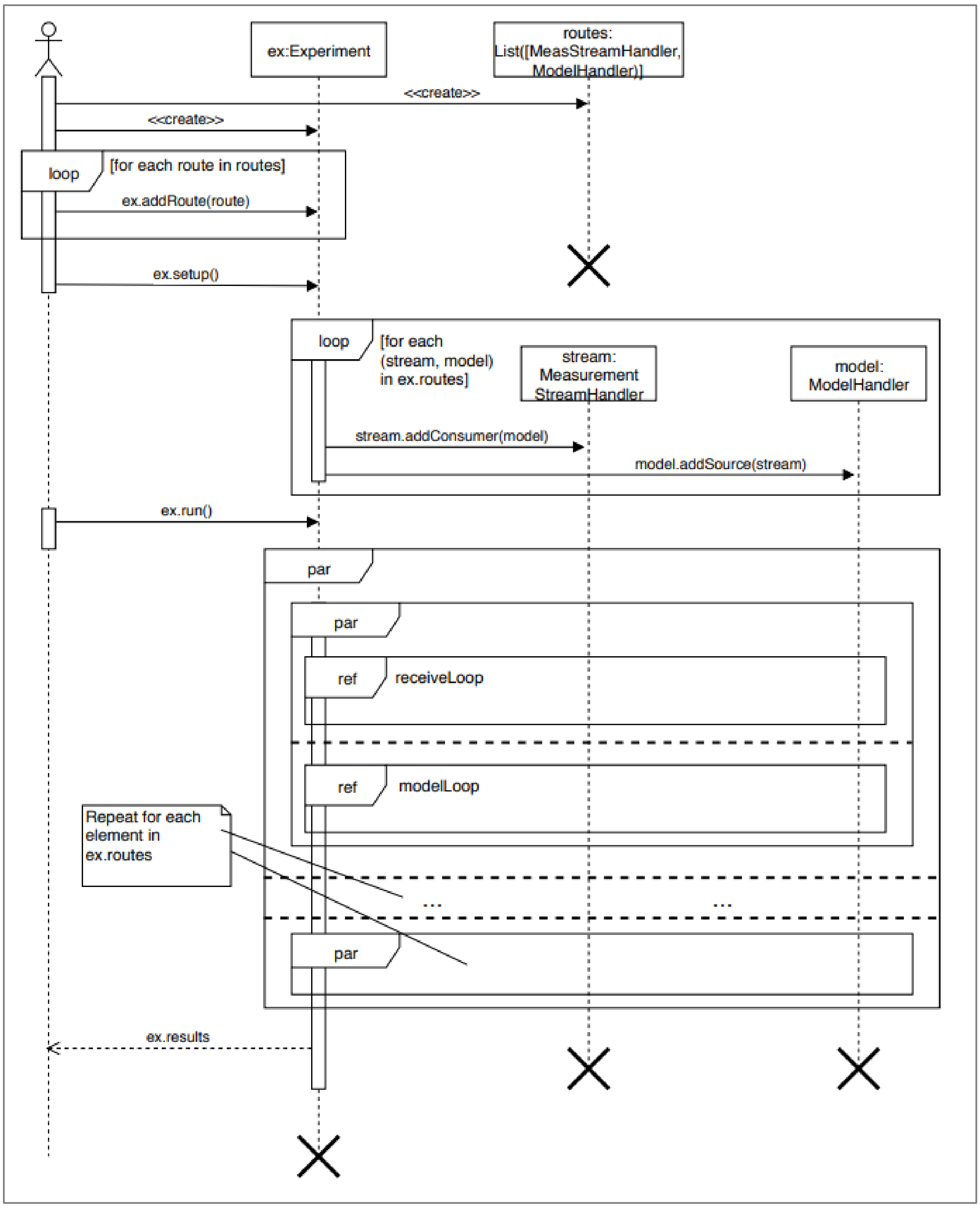

ex:Experiment
routes: List([MeasStreamHandler, ModelHandler)]
<<create>>
<<create>>
loop
[for each route in routes]
ex.addRoute(route)
ex.setup()
loop
[for each (stream, model) in ex.routes]
stream: Measurement StreamHandler
model: ModelHandler
stream.addConsumer(model)
model.addSource(stream)
ex.run()
par
par
ref
receiveLoop
ref
modelLoop
Repeat for each element in ex.routes
...
...
par
ex.results

Figure 19. *A generic many-to-many use case in ModelConductor*

4.4 Auxiliary components

ModelConductor includes components that are not viewed as being part of the core functionality of setting up input-output mappings between physical assets and simulation units, but are (at least) equally important from a user experience point of view. Specifically, herein result logging and input data validation strategies are briefly discussed, as they are implemented in the current MVP.

4.4.1 Result logging

Crucial points of interest in the timeline of a ModelConductor `Experiment` are the instants when the `ModelResponse` object is returned, concluding a single iteration of the model loop depicted in Figure 17. At this point, the contents of `ModelResponse` of the time step are written to an external file as comma separated values, ordered in the following manner:

$$timestamp\,, input\_keys\,, target\_keys\,, control\_keys$$

Consider the following pseudocode example, which governs a single `modelLoop` cycle from the moment the Measurement object becomes available to the moment when result logging takes place:

```
input_keys = ['a', 'b']
target_keys = ['c']
control_keys = ['sum']
initialize_log_file(input_keys, target_keys, control_keys)
measurement = {'a': 3, 'b': 5, 'irrelevant_key': 18, 'sum': 8}
model_input = []
for key in input_keys:
    model_input.append(measurement[key])
model_response = model_function(model_input)
assert(model_response == {'a': 3, 'b': 5, 'c': 8, 'sum': 8})
write_to_log(model_response, log_file)
```

As a result of calling `initialize_log_file` and `write_to_log` the following lines will appear in `log_file`:

```
timestamp, a, b, c, sum
dd.mm.YYYY HH:mm:ss, 3, 5, 8, 8
```

While the actual ModelConductor logging implementation is somewhat more involved, this simplified example serves the purpose of understanding the outline of main functionalities. From this example, it is also noticed that it is allowed for a measurement object to contain redundant information that is not used in any way by the experiment.

4.4.2 Measurement validation

Real-world data is noisy. In general, simulation models that could be used in conjunction with ModelConductor are not guaranteed to have tolerance against faulty inputs, but instead each `Measurement` should, in principle, be validated against a set of application specific rules before it is passed on to the `ModelHandler` instance. Currently, this functionality in ModelConductor is very limited in the sense that it can only check against missing values represented as `None` objects in the Python syntax. Two alternative strategies are currently implemented to deal with the `None` values found, i.e., i) revert to the latest value that is historically known to be valid in the experiment's context ii) do nothing and hope for the best — most likely ending up in the simulation crashing. The following pseudocode provides an illustration of the first approach:

```
validation_strategy = ‘latest_datapoint’
model_input1 = {‘a’: 3, ‘b’: 5}
# Selection of initial values is application-specific
a = 0
b = 0
def model_function(model_input):
   # input validations
   a = previous_a if model_input[‘a’] is None else \
        model_input[‘a’]
   b = previous_b if model_input[‘b’] is None else \
        model_input[‘b’]
   return {‘sum’ : a + b}
model_input1 = validate(model_input1, validation_strategy)
model_response1 = model_function(model_input1)
assert(model_response1 == {‘sum’: 8})
model_input2 = [‘a’ : 3, ‘b’: None}
model_response2 = model_function(model_input2)
assert(model_response2 == {‘sum’: 8})
```

4.5 Distribution, installation & licensing

The current development version of ModelConductor prototype is distributed as a MIT-licensed Python package repository, allowing comparatively liberal use, modification or distribution of the software for any purpose, including commercial purposes. The current version at the time of writing the thesis has been tested with Python version 3.7.4. The core structure of the project is spread across six Python modules (nine including those used for unit tests) and is outlined as follows:

```
|   LICENSE
|   README.md
|   setup.py
|   requirements.txt
|
\---modelconductor
        \---testresources
                train.py
                __init__.py
        exceptions.py
        experiment.py
        measurementhandler.py
        modelhandler.py
        server.py
        tests.py
        test_utils.py
```

```
                utils.py
                __init__.py
```

Listing 2. *The ModelConductor Project structure*

In the scope of the present work, no efforts have been made to make the library officially available from Python’s package index [58], but instead the source repository can be cloned directly from [32], wherein the latest stable version is maintained in the repository’s master branch. Subsequently, the library can be installed locally from sources using pip [56] by navigating to the project’s root directory (i.e., the one where `setup.py`) is located and running:

```
pip install .
```

Alternatively, to install ModelConductor and all the dependencies on one go, the following approach can be used:

```
pip install -r requirements.txt
```

5. EXAMPLE USE CASES

In the following subchapters, the organizational context in which the experimental part of the work was carried out is briefly discussed, i.e., the TUAS Internal Combustion Engine Laboratory (ICEL), by whom the thesis was commissioned. Subsequently, the developed Python library's utility is demonstrated by implementing a set of different measurement–model configurations, running the associated experiments and briefly examining the results.

5.1 Internal Combustion Engine Laboratory

At TUAS ICEL, experimental internal combustion engine studies are routinely carried out. The laboratory provides engine testing and measurement services mainly for industrial customers in the field of non-road mobile machinery such as in agriculture, forestry, construction or other similar areas. In a typical project, a heavy-duty diesel engine is fitted to an engine test stand by the laboratory's research personnel. The laboratory's four engine test stands are equipped with eddy current engine dynamometers that are designed to accommodate engines of up to roughly 250 kW peak output power. Concretely, the engine's flywheel is connected to the dynamometer via a specifically designed rigid clutch mechanism (Figure 20). The dynamometer then essentially acts as a brake, allowing the emulation of various static and dynamical loading profiles that would be imposed on the engine while the adjunct mobile machinery is being operated[5]. The dynamometer's loading throughout time can be controlled by a pre-set profile (such as the NRTC, see Chapter 5.2) or manually by the research personnel (a common approach when steady-state testing is desired). Specifically, the controlling is done utilizing LabView based software running on a remote workstation in a nearby control room.

[5] For in-depth discussion about various engine dynamometer types' working principles, as well as internal combustion engine testing practices in general, the interested reader is referred to the comprehensive text by Martyr & Plint [74].

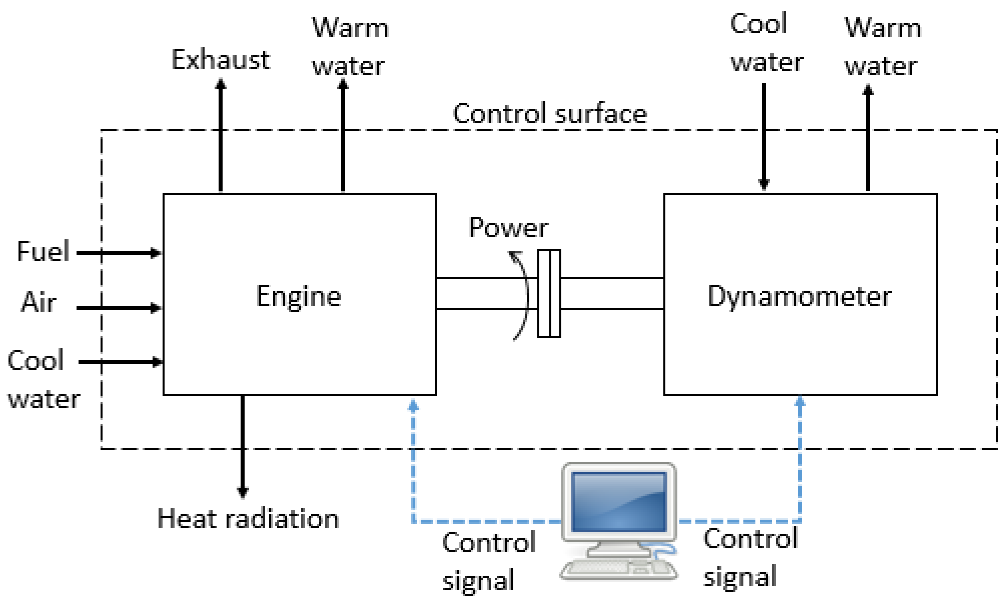


Figure 20. *A simplified schematic of an engine test stand similar to that used at TUAS ICEL*

The dynamometer provides means to measure engine speed (Revolutions Per Minute, RPM) and torque, which are registered to a MySQL database throughout the test runs. Simultaneously, various other measurements are registered as well. The engine test stands are equipped with various sensors and other devices for the measurement of, e.g.:

- Air intake and fuel consumption rates.
- Temperatures and pressures at various locations of the intake and exhaust manifolds and the engine body.
- Atmospheric conditions such as temperature, pressure and humidity.
- Environmental pollutant's concentrations in exhaust gases such as carbon dioxide, carbon monoxide, hydrocarbons, nitrogen oxides and particulate matter.
- Various attributes regarding engine auxiliaries such as the turbocharger or exhaust gas after-treatment devices.
- Attributes read directly from the Engine Control Unit (ECU) via CAN bus, such as various parameters regarding fuel injection strategy dependent on engine operation conditions.

In addition to these primary measurements, various derivative values are also recorded at experiment time. For instance, as will be subsequently illustrated in Chapter 5.2, the engine's output power is not measured directly, but instead computed as the product of torque and angular velocity. Altogether, some 150–200 separate measurement channels are registered to the database depending on the particular project configuration.

5.2 Prediction of engine power using FMU Simulink model

For this demonstration, the data used was the Non Road Transient Cycle (NRTC) used for emission testing of heavy duty off-road diesel engines [59]. The raw data is represented as a 1238 x 3 matrix with the first column corresponding to elapsed time in seconds, second column corresponding to normalized engine speed (percentage of maximum), and the third column corresponding to normalized engine torque (percentage of maximum). For the purposes of this experiment, the relative values were converted to absolute values by using 2600 rpm and 540 Nm for the maximum values of speed and torque, respectively. Subsequently, the data was augmented with a simulated noisy power measurement. The output power from a reciprocating engine is given as the product of the angular velocity of the crankshaft and the torque:

$$P_{out}\ [\mathrm{W}] = \omega \left[\frac{\mathrm{rad}}{\mathrm{s}}\right] \cdot T\ [\mathrm{Nm}] \tag{9}$$

where

$$\omega \left[\frac{\mathrm{rad}}{\mathrm{s}}\right] = \frac{n}{60}\ [\mathrm{s}^{-1}] \cdot 2\pi \tag{10}$$

wherein n is the revolution rate expressed in Revolutions Per Minute (RPM) [60]. Because the objective is to demonstrate a configuration which generates online evaluations of a system and is able to compare the results against a stream of incoming measurement data, some gaussian noise was added on top of the computed power. The final input data obtained in this manner is depicted in Figure 21. Here, the “power” time series is computed utilizing Equations 9 and 10 and adding random gaussian noise with 0 mean and a standard deviation of 5 kW, hence simulating a noisy measurement.

In the model side, the computation of power was implemented as a simple Simulink model (Figure 22), having an input vector consisting of speed and torque values and, correspondingly, a single scalar output P_{out} based on Equations 9 and 10.

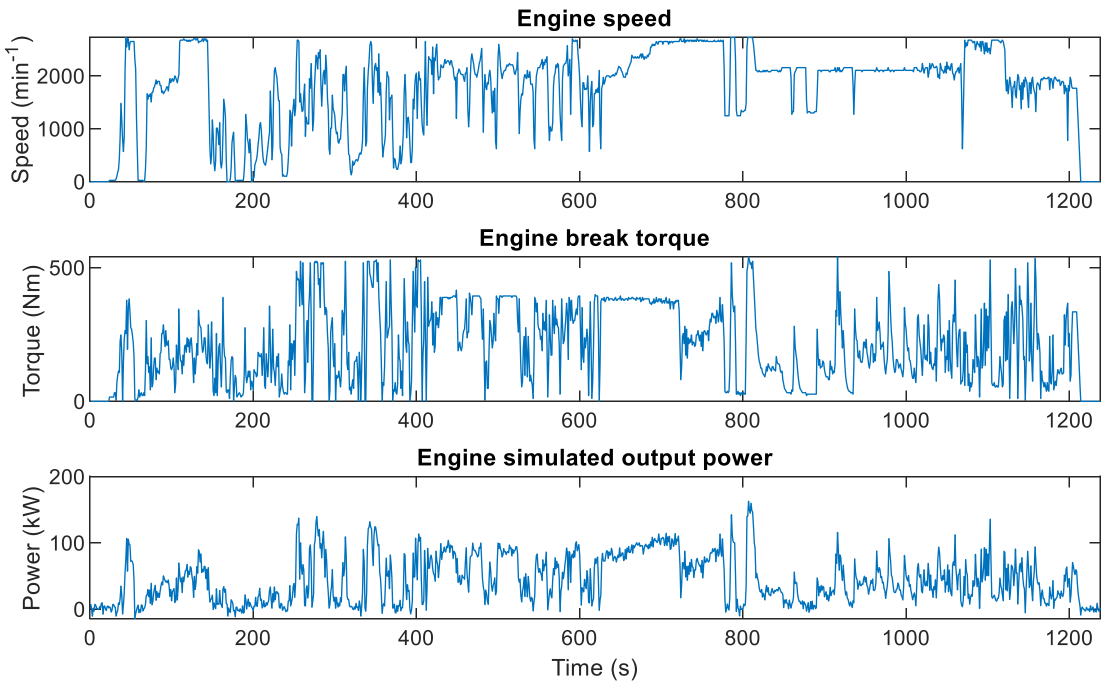


Figure 21. *The input signals for FMU ModelConductor experiment*

To allow the running of the model developed in Simulink (Figure 22) within a co-simulation session instantiated with ModelConductor, the model was exported using MATLAB's project functionality [61] as an FMU.

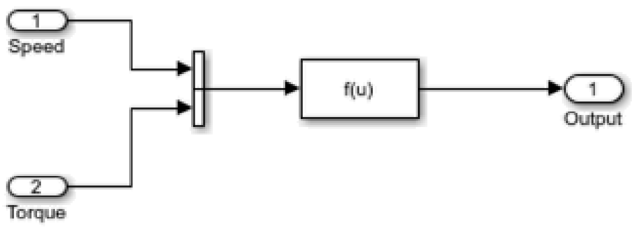


Figure 22. *The Simulink model for testing ModelConductor*

As discussed in Chapter 3.3.2, an FMU is an archive file which implements the Functional Mock-up Interface [54]. At the very core of the produced FMU is the `modelDescription.xml` file, an excerpt of which is outlined in Listing 3. The utility of `modelDescription.xml` is to act as a resource from which an external tool supporting the FMI (in our case, the ModelConductor library) will get all the necessary information about how to invoke the binaries. In this instance, it can be verified by examining the file that the variables exposed for FMI operations by the packaged Simulink model are clearly described (highlighted in bold text), along with some additional metadata (Listing 3).

```
...
<ModelVariables>
   <ScalarVariable causality="input"
                   description="Input Port: Speed"
                   name="Speed"
                   valueReference="0"
                   variability="discrete">
        <Real start="0"/>
        <Annotations> ... </Annotations>
   </ScalarVariable>
   <!--Index = 1-->
   <ScalarVariable causality="input"
                   description="Input Port: Torque"
                   name="Torque"
                   valueReference="1"
                   variability="discrete">
        <Real start="0"/>
        <Annotations> ... </Annotations>
   </ScalarVariable>
   <!--Index = 2-->
   <ScalarVariable causality="output"
                   description="Output Port: Output"
                   initial="calculated"
                   name="Output"
                   valueReference="2"
                   variability="discrete">
        <Real/>
        <Annotations> ... </Annotations>
   </ScalarVariable>
  <!--Index = 3-->
</ModelVariables>
...
```

Listing 3. *An excerpt of the modelDescription.xml file contained in the FMU. Segments denoted by … are omitted for brevity*

Running the Simulink-generated FMU requires a valid Simulink license on the workstation which the FMU model checks out automatically provided that an FMU session has been invoked in MATLAB. The process could be automated but for the purposes of this experiment, an instance of MATLAB was manually started up, followed by issuing the relevant command which starts the co-simulation session [61].

ModelConductor currently supports the loading and co-simulation of FMUs by implementing a wrapper class for the open source FMPY library [62]. According to the ModelConductor architecture, the experiment consists of periodically polling a database for new data, and whenever new data is available it is fed forward to the connected model. However, as in this case the raw data resides in a plain csv file, a separate mechanism was developed to simulate periodical measurements being written to a database. The Python implementation of a function called `simulate_writes` is given in Appendix A.

Essentially, upon calling `simulate_writes` the code begins going through the input data csv file row by row at an interval defined by the function's only argument, writing the rows to a test database along the way. This is meant to approximate the process of measurements coming in from an actual physical process. At each round of the data receiving loop, the database is polled for the most recent entry. In the context of this artificial example, the "most recent" entry is naively interpreted as the one having the largest integer primary key auto-generated by `simulate_writes` which hasn't yet been utilized by the experiment.

Now, with the MATLAB FMU session in place along with a means to simulate incoming measurement data, a ModelConductor experiment can be set up. Closely approximating the one-to-one use case behavior described in Figure 18, the purpose will be to watch out for the incoming data and to make inferences from the FMU model. The process is rather straightforward utilizing the built-in methods contained in ModelHandler classes, as illustrated by Jupyter notebook snippet in Appendix A. When the notebook is run, the Simulink instance is activated programmatically and the user is able to visually inspect that simulation steps are actually taken by viewing the simulation results in a real-time plotter window. Similarly, it can also be verified that the model works by examining the debug output from the `ex.run` method (Appendix A). On a final note, the results are saved to an output csv file for later examination and/or post-processing with arbitrary tools. Figure 23 displays a plot of the final results produced in MATLAB environment.

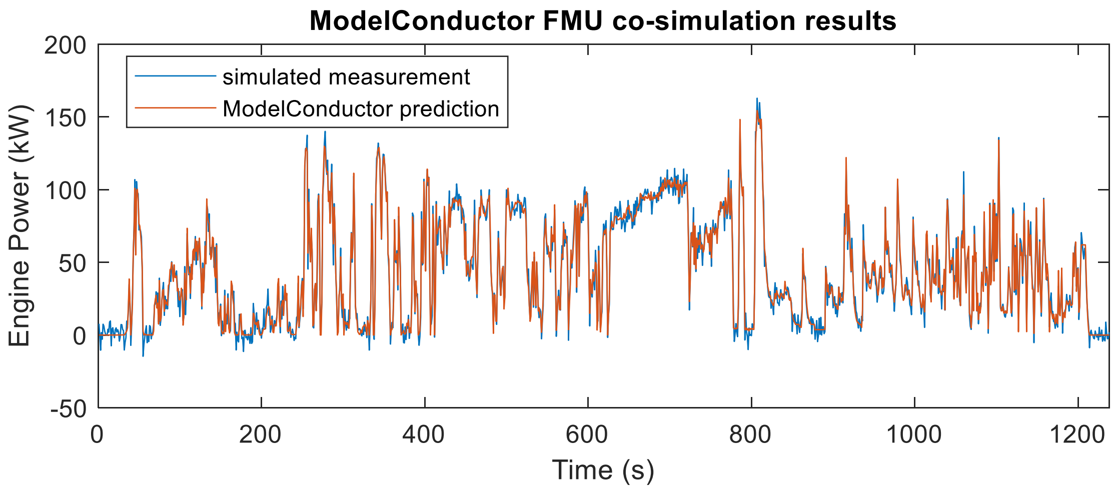


Figure 23. *ModelConductor one-to-one Experiment results*

5.3 Online prediction of engine NO_x emissions by machine learning

The simplified development process of an internal combustion engine can essentially be viewed as an optimization problem of maximizing thermal efficiency, i.e., the ratio of mechanical energy obtained from the crankshaft to the energy input contained in the fuel consumed during the same period. The set of feasible engine configurations is constrained by various factors, such as the tolerance of the materials used against mechanical and thermal stress. Perhaps most importantly, an engine manufacturer needs to consider the maximum acceptable emission levels imposed by the authorities. The tools the development engineer has at hand to approach this problem include, but are not limited to, the selection of an appropriate fuel injection strategy as well as implementing various types of exhaust gas aftertreatment devices.

In particular, in diesel engines it has proven a challenging task to control Nitrogen Oxides (NO_x) levels while also maintaining an acceptable level of Particulate Matter (PM) emissions and adequate fuel efficiency. In order to implement optimal emission control strategies, using fast-running real-time models in the ECU to estimate NO_x from engine parameters has attracted plenty of discussion recently (see, e.g., [63]–[67]). This serves as a motivation behind our following example of ModelConductor digital twin instantiation, the main features of which are highlighted in subsequent discussion. Moreover, the experiment's second part is documented in a Jupyter notebook format in Appendix B.

Along with the other emission components such as carbon monoxide and particulate mass, the NO_x levels are routinely registered during the engine test runs conducted at TUAS ICEL. In this experiment, real-world engine measurement data is utilized, taken from a NRTC cycle ran on a 4.9 liter heavy duty diesel engine, the key characteristics of which are given in Table 1.

Table 1. *Key characteristics of the target engine for simulation*

Attribute	Description
Engine type	In-line, 4-cylinder, 4-stroke, turbocharged & intercooled
Displacement (dm^3)	4.9
Bore (mm)	108
Stroke (mm)	134
Max. Torque (Nm)	860
Rated power (kW)	148
Fuel type and fuel injection system type	Diesel, common rail

The raw data was manually extracted from the laboratory's MySQL database using a LabView based export tool routinely used by the research personnel in the laboratory to perform basic data analysis tasks. The tool in question is developed for the laboratory by a third party contractor with its source code undisclosed. Consequently, many details regarding data pre-processing and the actual query made towards the database remain unclear. This by no means is an optimal situation, but can still be considered acceptable for the purposes of the example.

Eventually, the export process resulted in a data set of 12,390 samples of 158 variables, with each sample corresponding to a snapshot of the variables' values over an averaging window of 0.1 seconds. Hence, the total duration of the cycle is inferred to be 1,239 seconds, matching the value given in the official NRTC specification [59]. A preliminary exploratory analysis of the data was performed, with the intent of finding the most interesting independent variables in terms of predicting the dependent variable 'NOx_Left', the distribution of which is presented in Figure 24.

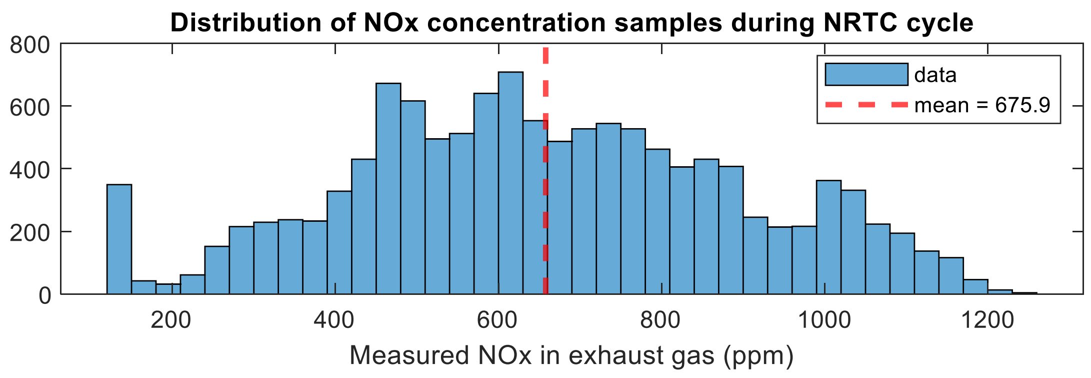


Figure 24. *Distribution of NO_x concentration during the NRTC cycle*

As a result of a semi-formal pre-processing phase including, e.g., the examination of linear correlations of paired variables as well as applying common domain knowledge about internal combustion engines, the data was eventually narrowed down to 49 variables showing most potential. Subsequently, the data was randomly split into training and test sets using a 67/33 split and various machine learning models were fitted to the data for quick comparison utilizing the scikit-learn library. The results obtained from the various models trained are summarized in Table 2 as Mean Absolute Error (MAE) computed for the training and test sets.

Table 2. *Results from training various ML models for the NOx prediction problem*

Model type	Training Error[6] (MAE)	Test Error[7] (MAE)
Linear regression	93.7	96.5
Polynomial regression (2nd order)	35.6	43.5
Random forest (100 trees, maximum depth 25)	4.6	12.1

For brevity, the models themselves are treated as generic black-box learning agents and any further discussion about the implementational technicalities involved is omitted, as well as the possible reasons why some models might perform better than others. The fact is also disregarded that obviously the data faces an overfitting problem as the model complexity is increased, manifested specifically in the case of the random forest by the large difference between training and test error[8].

For further developments, the random forest model was the obvious choice, since the provided data fit can be, at least for the contemporary purpose, considered excellent. This is perhaps best illustrated by observing from Figure 24 that the target variable roughly ranges between 150 ppm and 1200 ppm, while the model on the average only makes an error of 4.6 (12.1) ppm in the training set (test set). It bears repeating, however, that for any industrial or scientific applications beyond demonstrative purposes the obvious overfitting issue would definitely be something the data scientist should look into, perhaps by introducing some sort of regularization technique [68, Ch. 7].

Two separate experiments were conducted with the chosen model. The first one utilized an SQLite database as means for getting the input data by issuing periodical SQL polls, at intervals defined by a `polling_interval` attribute[9] of the `IncomingMeasurementPoller` class. Meanwhile, a mechanism similar to that described in Chapter 5.2 (function `simulate_writes` in Appendix A) is used to simulate the measurements being concurrently written to the same database. The second experiment, on the other hand, utilized a continuous data stream over a TCP socket. For both experiments, the simulation results in comparison to the measured NO_x values (Figure 25) were asserted to be identical.

[6] Rounded to nearest single decimal point.
[7] Rounded to nearest single decimal point.
[8] For a thorough discussion on these topics, the interested reader is referred to the comprehensive presentation in [68].
[9] The meaning of `polling_interval` will be discussed in more detail in Chapter 5.4.

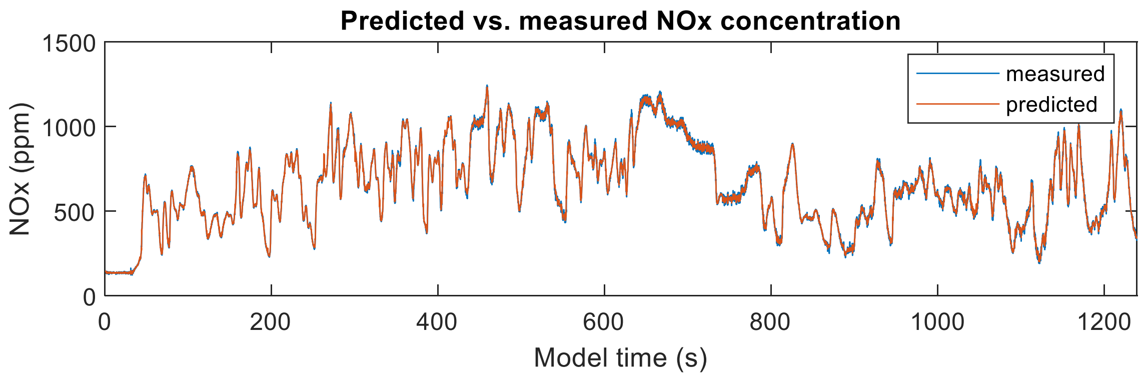


Figure 25. *Simulation results*

For the database experiment, the SQLite file was initially empty, and physically stored on a commodity SSD disk with a maximum reading speed of 540 MB/s and a writing speed of 250 MB/s as reported by the manufacturer[10]. A baseline for database write speed was first established by executing only the write loop without any concurrent read operations being executed towards the database. By setting the `delay` parameter (see definition of `simulate_writes` in Appendix A) to 0.1 seconds, the whole dataset was iterated in 1886 seconds, resulting in the duration distribution for a single write operation plotted in Figure 26.

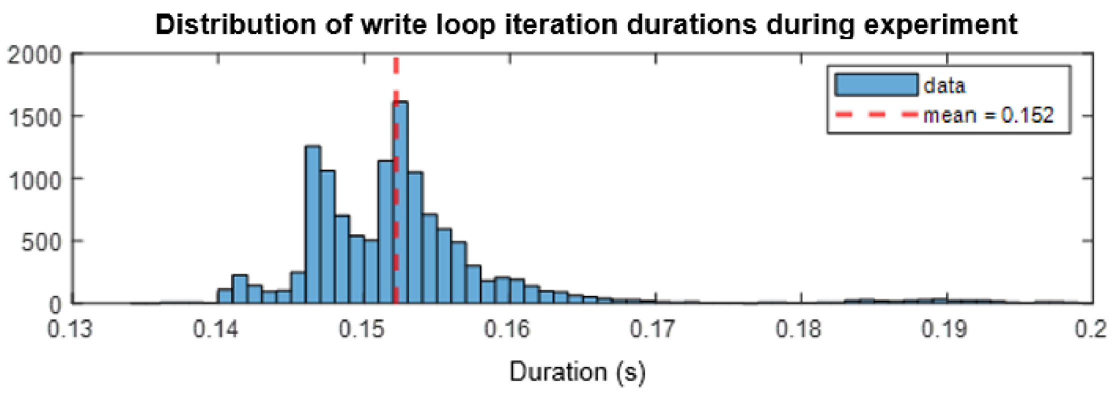


Figure 26. *Distribution of durations of the database write operations*

After setting up a baseline, the actual concurrent read/write experiment was then repeated with various values of `polling_interval` parameters while the write interval was, to the extent possible, maintained constant. As explained in Chapter 4.4.1, a timestamped entry is written to an output log file each time a `ModelResponse` object is received from the `SklearnModelHandler` instance. Having a priori knowledge about how much the simulated time advances between two consecutive `ModelResponses` (in this case, 0.1 seconds) allows to define the following:

[10] A performance benchmark conducted shortly after the experiment with a tool provided by the manufacturer reported a reading speed of 505 MB/s and a writing speed of 180 MB/s.

$$t = \text{simulated time elapsed since beginning of the experiment} \tag{11}$$

$$\tau = \text{wall-clock time elapsed since beginning of the experiment} \tag{12}$$

$$t/\tau = \text{real-time performance coefficient} \tag{13}$$

The ratio t/τ becomes an important performance metric[11] for these types of experiments and is utilized extensively also in subsequent examples of the present work. As it is illustrated in Figure 27, the real-time performance in terms of t/τ was, in the database experiment, modest at best. However, the data suggests that best results are obtained by setting `polling_interval` just slightly larger than the write rate.

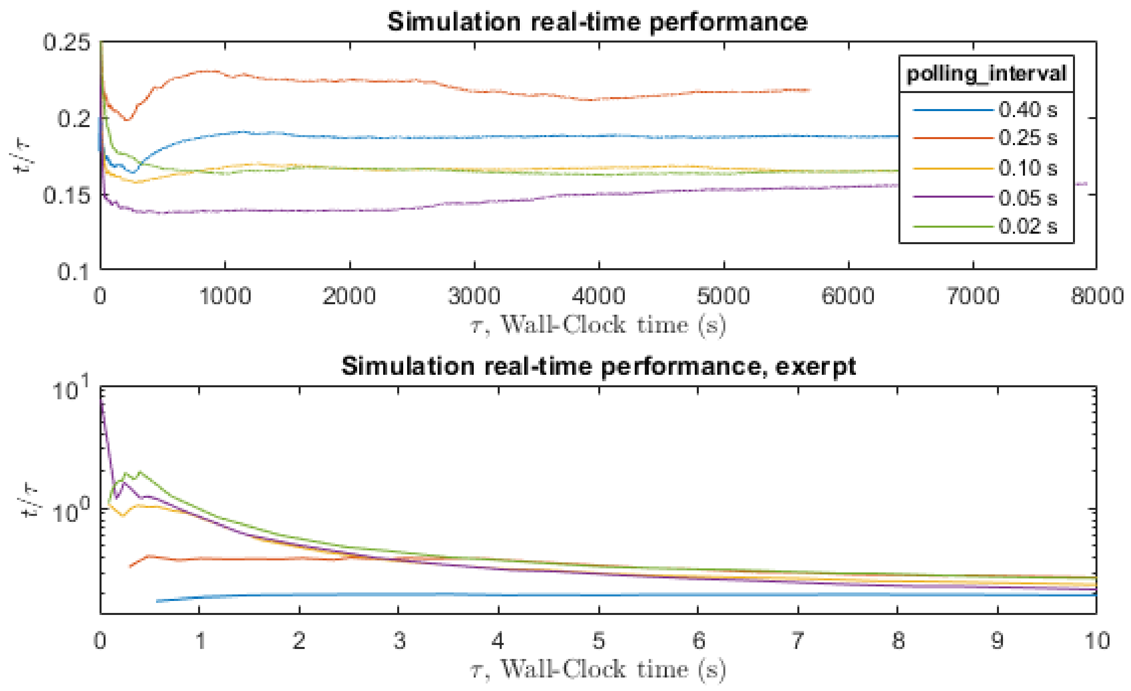


Figure 27. *Simulation real-time performance, poll-based experiment*

An additional viewpoint to the experiment's performance is obtained by looking at Figure 28 where the effect of various `polling_interval` values on the SQLite's ability to write data is compared.

[11] Situations exist when t/τ can *not* be considered a valid performance metric — see Chapter 3.1 for complementary remarks.

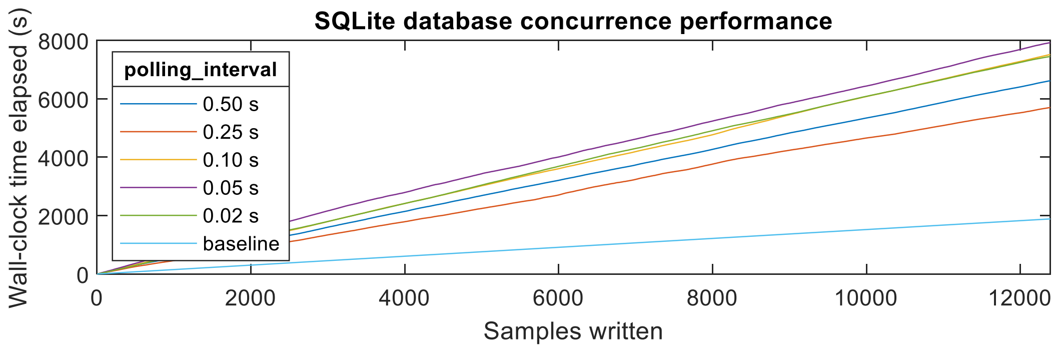


Figure 28. *Wall-clock time elapsed vs. samples written, poll-based experiment*

The second test utilized a client–server approach. Specifically, it consisted of setting up a remote data stream simulation using a Raspberry Pi unit and a corresponding receiver component using ModelConductor library components. On the client side, the data was read from a csv file row by row and subsequently transmitted using the approach outlined in Listing 4. The result will be a stream of JSON-formatted strings (JavaScript Object Notation), discretely spaced in time by call to Python's `time.sleep` method inside the main loop. Each message is prepended with a fixed-length header portion containing information about the message's length, allowing the messages to be correctly parsed at the receiving end.

```
"""Script for simulating IOT measurement stream to
ModelConductor experiment."""

#---Import statements omitted for brevity

#---Get the raw data
data = pandas.read_csv('nrtc1_ref_10052019.csv',
                       delimiter=';')

#---Connection parameters
HOST = "192.168.1.5"
PORT = 33003
ADDR = (HOST, PORT)

client_socket = socket(AF_INET, SOCK_STREAM)
client_socket.connect(ADDR)

#---Main loop

def send(my_msg, event=None):
    """Handles sending of messages."""
    client_socket.send(bytes(my_msg, "utf8"))
    if my_msg == "{quit}":
        client_socket.close()

for _, row in data.iterrows():
    my_msg = row.to_json()
    # add message length to header
```

```
        my_msg = "{:<10}".format(str(len(my_msg))) + my_msg
        print(my_msg[0:50], "...") # debug
        send(my_msg)
        sleep(0.1)

```

Listing 4. *The client side Python script utilized in TCP streaming experiment*

At the receiving end, the `IncomingMeasurementListener` class [32] assumes the responsibility of establishing the TCP connection to the client and parsing the JSON strings to `Measurement` objects. Herein, an original high-level protocol for parsing the messages was implemented in the `handle_client` function of the `server` module [32], the purpose of which is to ensure that exactly one JSON string received from the client ends up being interpreted as exactly one `Measurement` object that is added to the buffer. The TCP layer fortunately takes care of low-level tasks such as data ordering and ensuring the uniqueness of each transmitted segment [69], making this task substantially simpler.

The experiment was repeated with two different network configurations: One with the Raspberry in the same local network set up via Ethernet, and one where the data was transmitted over the Internet by connecting the Raspberry to a 4G hotspot hosted from an Android phone. As it is illustrated in Figure 29, both approaches resulted in t/τ values close to 1, which can be considered an excellent result. As was expected, on the red line in Figure 29 slight artefacts from the network's unpredictable behavior can be observed. Although the overall data throughput did remain satisfactory in this instance, this obviously might not always be the case.

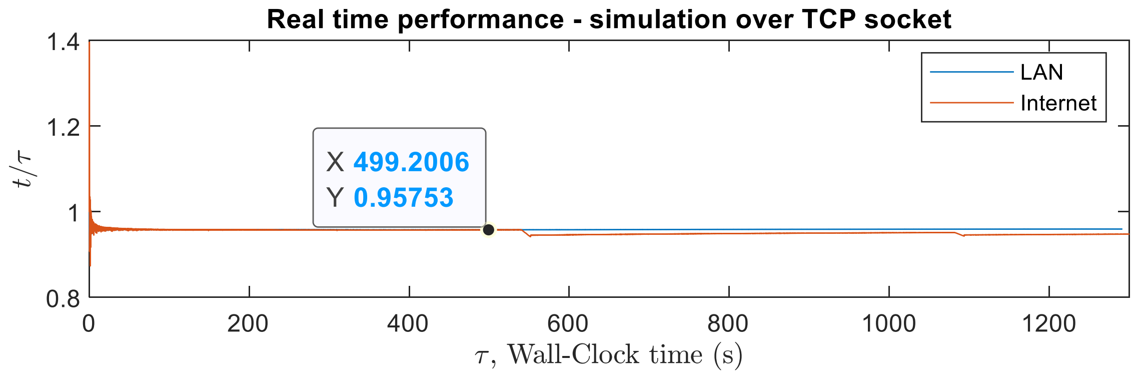


Figure 29. *Real time performance when streaming input data over TCP socket from a remote client*

In the very beginning of the "internet" experiment (and, to a lesser extent, LAN experiment also) the performance appears to fluctuate wildly. As illustrated by a zoomed-in version of the graph (Figure 30), in the internet experiment t/τ initially peaks at approximately 1.4 and then oscillates for a brief moment over and under 1 before stabilizing. In other words, in terms of simulated time, it *appears* that the data is momentarily being utilized *faster* in computations than what it takes in real time to send and receive the datapoint over the TCP socket, convert it to ModelConductor's internal datatype and append to buffer, re-convert to another data format that is expected by the model, send it to connected Sklearn ModelHandler and return the result. Herein the word *appears* is emphasized, since carefully examining the experiment's composition reveals that this cannot actually be true. For illustration, consider that in the client side main send loop (Listing 4) the execution is paused at each iteration for 0.1 seconds. This also happens to be exactly the same interval by which the simulation time advances between two consecutive time steps. Hence, one is able to arrive at the logical conclusion that at any given time t/τ can be *at most* 1, even without the unavoidable overhead resulting from network delay.

Although it could not be confirmed in the scope of the present work, it is argued that the observed counter-intuitive fluctuations in the beginning are mainly consequences of the real-time properties of the start-up phase of the experiment when it is run by the Python interpreter. Specifically, in the `run` method of `OnlineOneToOneExperiment` the data receiving loop is instantiated *before* the data utilizing loop. A situation could hence occur that the buffer already has some data at the time the `ModelHandler` component is initiated and has access to that data. As a result, it obviously becomes questionable whether t/τ, as it has been defined, can be considered a completely valid performance metric, specifically in the beginning phase of an experiment. It is, however, argued that t/τ still provides an useful estimate as the experiment progresses. Since t/τ is computed from cumulative values of simulated and real time passed, the fluctuations are expected to eventually even out.

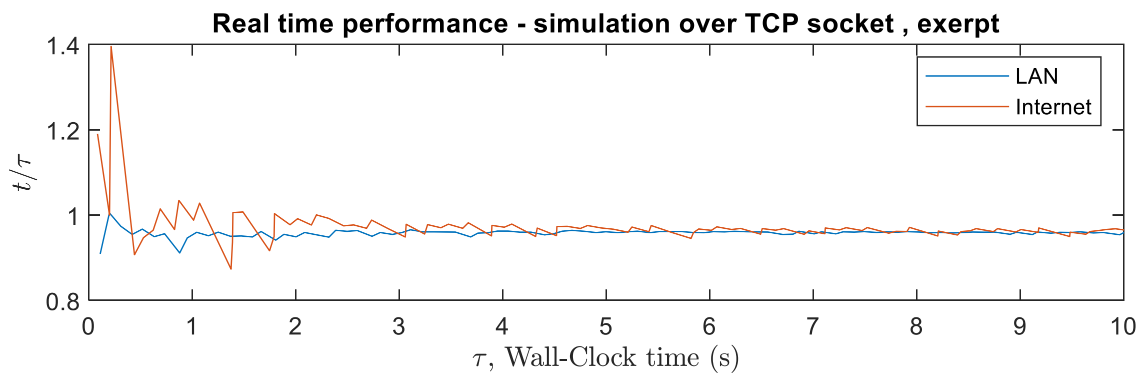


Figure 30. *The beginning of real time performance in streaming experiment*

5.4 Real-world engine test data simulation with GT-SUITE

The two previous examples used data that was created and/or accessed in an artificial manner. With respect to FEDS (Chapter 2.2), the third and last example brings us closer to a naturalistic approach in the chosen evaluation strategy. Specifically, a demonstration is presented of integration of the ModelConductor framework to a database that is in productional use in the thesis' target organization.

5.4.1 Access to input data

The data resulting from the engine test runs conducted at TUAS ICEL is persisted in a MySQL database on a server maintained locally on TUAS premises. On the MySQL server, each of the four test cells in the engine lab are represented as a separate database. As for the parts relevant for the purpose of the contemporary work, the database's table scheme is depicted in Figure 31.

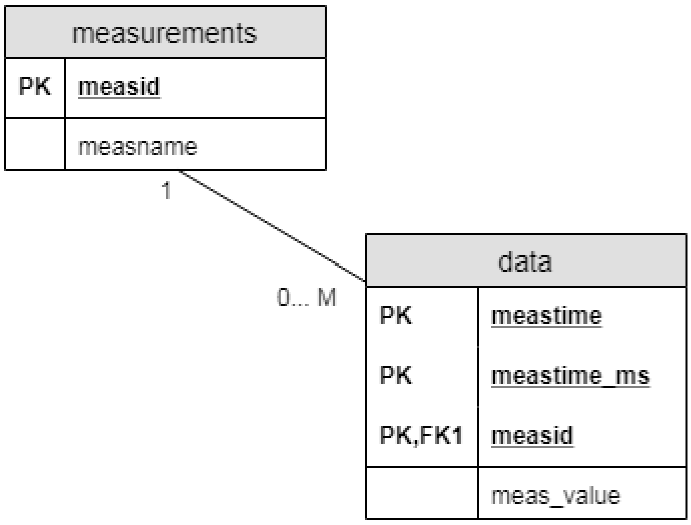


Figure 31. *An excerpt of table scheme at TUAS ICEL's MySQL database*

According to Figure 31, the raw data originating from sensors and other equipment connected to a test engine is warehoused in a single 'data' table. Each record is uniquely identified by a composite primary key consisting of a timestamp (further split into a string of the format DD-MM-YYYY HH:mm:ss plus an integer millisecond part) and a `measid` integer identifier of the measurement channel (variable). At query time, if the actual human-interpretable identifier of the measured variable is desired, it must be resolved separately for each record in 'data' via a join operation to another table called 'measurements'. Furthermore, since all the variables are independent in terms of their time axes, getting the data into a more usable format requires some further preprocessing steps. Listing 5 displays an illustrative example of such a query.

```
select meastime as Time,
sum(case when measid = 191 then meas_avg end) 'Torque',
sum(case when measid = 156 then meas_avg end) 'Speed',
from
(
select data.meastime, data.measid, measurements.measname,
     avg(data.meas_value) as meas_avg
from data
inner join measurements
on data.measid = measurements.measid
where meastime >= '2019-08-20 10:25:00'
and meastime < '2019-08-20 11:00:00'
group by data.measid, data.meastime
) as measwindow
group by meastime
```

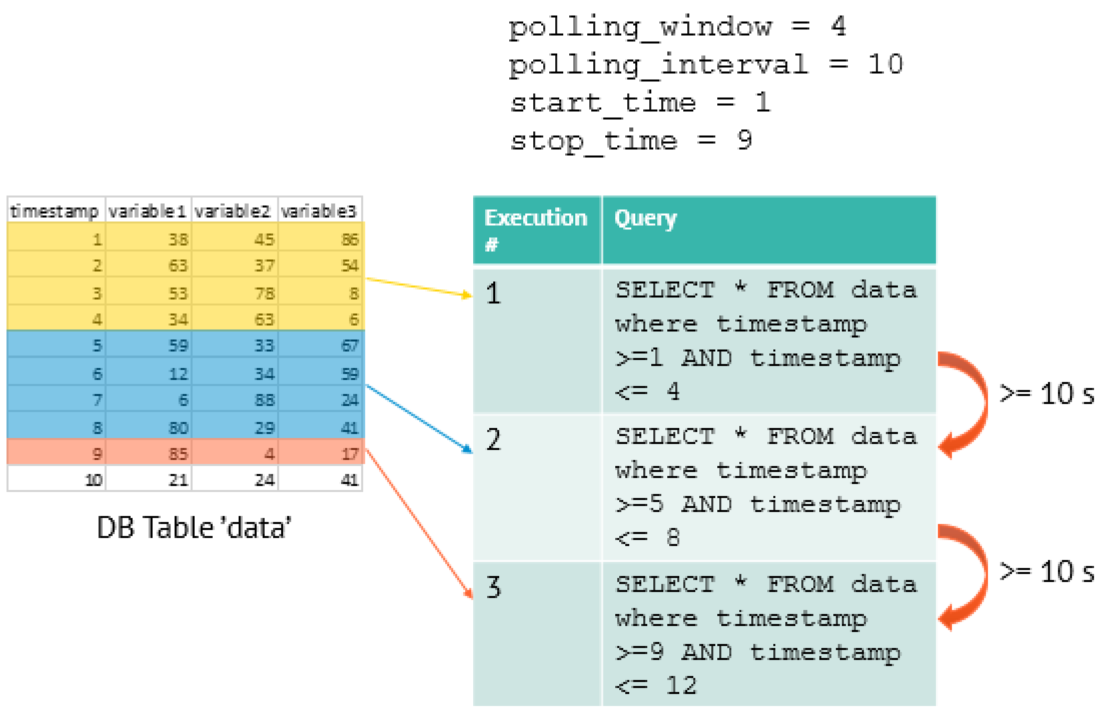
Listing 5. *An example query*

The example query, which will be used as a template for the actual experiment, results in a multivariate time series with an interval of one second, similar to that depicted in Table 3.

Table 3. *Example result from query at MySQL database at TUAS ICEL*

Time	Torque	Speed
2019-08-20 10:25:00	84.00	1499.99
2019-08-20 10:25:01	83.99	1500.03
…	...	...
2019-08-20 10:59:59	121.23	1600.34

The query is quite costly even with only a few variables; in the example case of Listing 5 the query took approximately 76 seconds. Therefore, the number of executions should be limited during experiment time. Such functionality is implemented in the `Incoming-MeasurementBatchPoller` class, which provides means to stepwise iterate through SQL tables with predetermined “window” and “interval” arguments, as well as global start/stop arguments. Figure 32 illustrates a generic use case. While this simplified example omits some further complexities involved in the general use case of transforming the table data into the desired format, it is useful for understanding the basic functionality. The data is taken in as windowed batches of four records, dynamically injecting the required timestamps in to the SQL query strings at each iteration. The resulting rows are then transformed into `Measurement` objects (ModelConductor’s internal data format) and put to the FIFO buffer queue. An additional parameter `polling_interval` allows the user to fine-tune the stepping process as it defines the minimum wall-clock time interval at which queries are to be executed. The actual interval between two consecutive queries is, then, taken as the maximum of the first query’s runtime and `polling_interval`.



```
polling_window = 4
polling_interval = 10
start_time = 1
stop_time = 9
```

timestamp	variable1	variable2	variable3
1	38	45	86
2	63	37	54
3	53	78	8
4	34	63	6
5	59	33	67
6	12	34	59
7	6	88	24
8	80	29	41
9	85	4	17
10	21	24	41

DB Table 'data'

Execution #	Query
1	`SELECT * FROM data where timestamp >=1 AND timestamp <= 4`
2	`SELECT * FROM data where timestamp >=5 AND timestamp <= 8`
3	`SELECT * FROM data where timestamp >=9 AND timestamp <= 12`

Figure 32. *Simplified example of IncomingMeasurementBatchPoller operation*

By varying the composition of `polling_interval`, `polling_window`, `start_time` and `stop_time`, a wide selection of real-time data reading configurations can be realized. Most importantly, the relationship between `polling_interval` and `polling_window` should be chosen accordingly to whether the experiment's data throughput is constrained on the data receiving or data utilizing end of the framework. If the rate at which data is being written to the target database is known a priori, this could be used as a good starting point for `polling_interval` and `polling_window`. As explained, the query complexity also has implications to actual performance.

For the actual experiment, a dynamic engine cycle occurring on October 3rd, 2019 with a duration of roughly 45 minutes was chosen. The actual ModelConductor experiment was preceded by an exploratory study of the input data, where the complete dataset was obtained in offline manner from in a single query made to the MySQL database. The main findings are summarized in Table 4 and Figure 33. For non-disclosure restrictions, the numerical values of the different attributes are not published within the thesis. In general, it can be said however that the data was found to be of good quality and no further preprocessing was done at the online experiment phase, except for treating the missing values in the InjPressure variable (Table 4) and making the necessary unit conversions. For the first purpose, ModelConductor's `Experiment` class implements a strategy to store the last valid values of each variable as a backup in case NaN values are confronted during runtime. Unit conversions were handled in the SQL query (Listing 6).

Table 4. *Descriptive statistics of input data to the experiment (NDA restrictions apply)*

	InjQuantity	ECUTiming	InjPressure	BoostPressure	ExhBackPressure	PowerMeasured	SpeedControl	TorqueControl
Count	2700	2700	2470	2700	2700	2700	2700	2700
NaN	0	0	230	0	0	0	0	0

Illustrated in Figure 33 is the data's highly dynamic nature. Save for a brief warm-up period, the operation characteristics of the engine are constantly fluctuating.

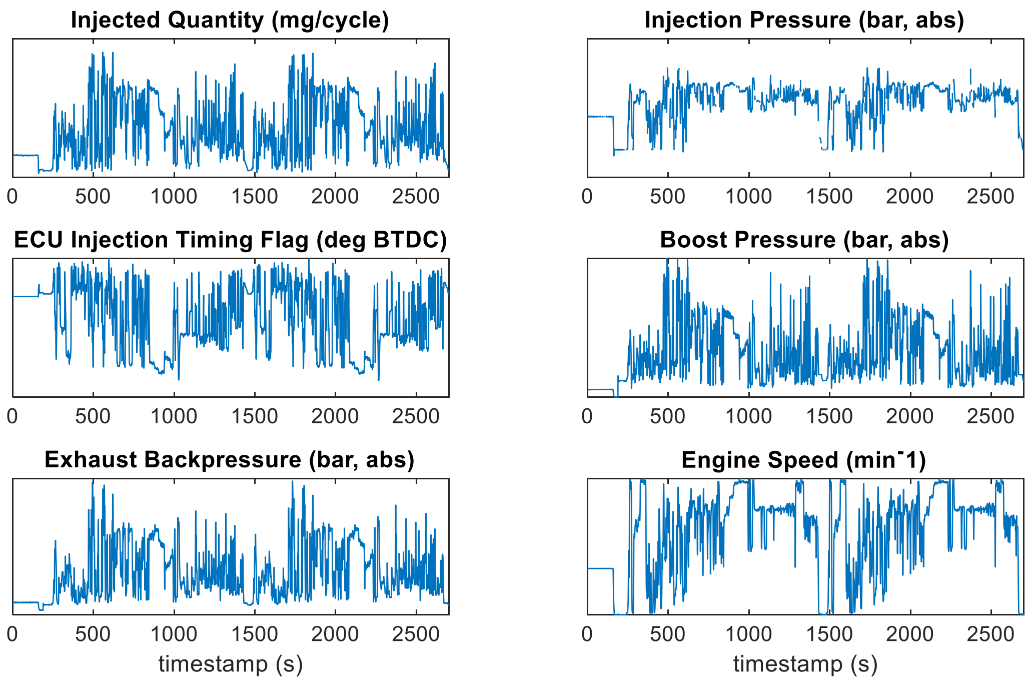


Figure 33. *Input signals to GT-SUITE FMU simulation (y-axes values undisclosed for NDA restrictions)*

During the online experiment, input variables were periodically read from the MySQL database as moving average values with a window of one second using the query outlined in Listing 6. Along with the input variables, the measured engine power is recorded for future reference and result validation. In a couple of cases, slight preprocessing is applied to accommodate for the necessary unit conversions.

```
 1  select meastime as Time,
 2  sum(case when measid = 211 then meas_avg end)
 3  'InjQuantity',
 4  sum(case when measid = 197 then meas_avg end) * -1
 5  'EEMTiming',
 6  sum(case when measid = 222 then meas_avg end) * 10
 7  'InjPressure',
 8  sum(case when measid = 137 then meas_avg end)
 9  'BoostPressure',
10  sum(case when measid = 135 then meas_avg end)
11  'ExhBackPressure',
12  sum(case when measid = 141 then meas_avg end)
13  'PowerMeasured',
14  sum(case when measid = 156 then meas_avg end)
15  'SpeedControl',
16  sum(case when measid = 191 then meas_avg end)
17  'TorqueControl'
18
19  from
20  (
21  select data.meastime, data.measid, measurements.measname,
22       avg(data.meas_value) as meas_avg
23  from data
24  inner join measurements
25  on data.measid = measurements.measid
26  where meastime >= '{}' and meastime < '{}'
27  group by data.measid, data.meastime
28  ) as measwindow
29  group by meastime
```

Listing 6. *SQL query used in the experiment*

As explained earlier (p. 63), on row 26 of Listing 6 curly braces are used to denote the placeholders for injecting the query window's starting and ending timestamps at execution time. The resulting online data stream will, after the experiment has concluded, total up to a 9-column matrix similar to that depicted in Table 3, with starting and ending timestamps:

$$t_{start} = \text{2019-10-03 10:15:00}$$

$$t_{end} = \text{2019-10-03 11:00:00}$$

5.4.2 Engine modeling preliminaries

In the present example use case, a dynamic engine model was built in GT-SUITE software and supplied with a stream of inputs originating from real-world measurements to demonstrate the digital twin experiment capabilities of ModelConductor. GT-SUITE is a proprietary multiphysics modeling and simulation tool provided by Gamma Technologies, Inc. used extensively in, e.g., automotive, marine and off-road mobile machinery industries [70]. At TUAS ICEL, it is routinely utilized for engine modeling and simulation tasks that supplement experimental measurements conducted at the laboratory’s engine test stands. Figure 34 provides an illustration of what a dynamic model map in GT-SUITE’s GUI might look like.

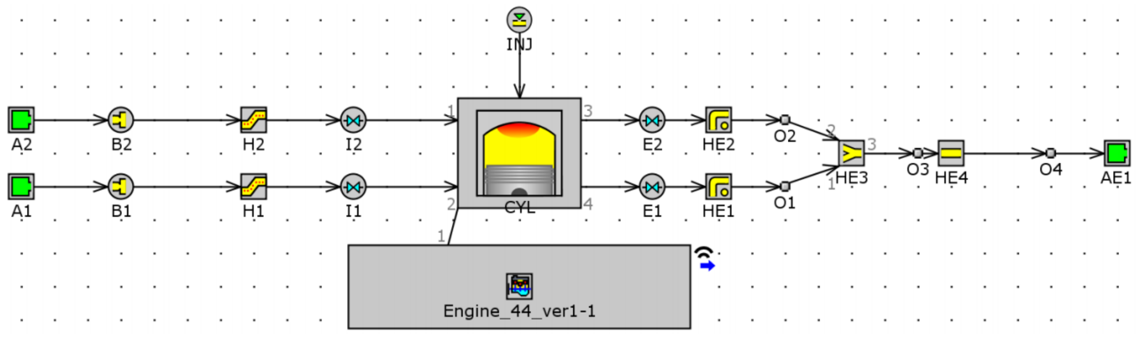


Figure 34. *Example of a simple one cylinder engine model in GT-SUITE built for another project carried out at TUAS ICEL* [71]

In typical automotive (or similar) applications, a one-cylinder engine such as the one being modeled in Figure 34 would be a rare occurrence. Nevertheless, it provides a useful starting point for understanding how GT-SUITE can be used in the industry as a design tool for understanding the implications of various engine configurations and choice of key parameters. This particular model’s experimental frame (Chapter 3.1) could be summarized as follows. The main components being modeled are the intake and exhaust ports, the intake and exhaust valves, the cylinder, the injector and the crankshaft. At the intake side, the system boundary is placed at the point where the intake air enters the intake port, after being compressed by the turbocharger and subsequently cooled down by an air-to-water intercooler. Since in this simplified model the detailed descriptions of the turbocharger-intercooler assembly are omitted, the pressure and temperature of air entering the intake ports are variables that must be experimentally determined. On the model map, they are represented by the specific components ‘A1’ and ‘A2’. Similarly, on the exhaust side, the system boundary is placed right before exhaust gases would enter the turbine side of turbocharger in the real engine. The thermodynamic conditions at this point must, again, be described by the user within the ‘AE1’ component.

A full working cycle of a 4-stroke reciprocating engine consists of intake, compression, work and exhaust strokes, corresponding to two full revolutions of the crankshaft. During intake stroke, the piston is moving downwards, and a pressure differential causes fresh air to be introduced into the cylinder via the intake valves (I1, I2) that open and close in relation to the rotation of the crankshaft at times pre-determined by the user. During compression, piston moves up, causing the pressure and temperature of air to increase. When the piston is nearing the top end of the cylinder (Top Dead Center, TDC), a fine-grained spray of liquid fuel is introduced via the injector component, which then mixes with the air and ignites due to the high temperature. During the working stroke, the pressure in the cylinder rapidly rises due to combustion. The pressure is transferred as mechanical energy to downwards movement of the piston, the rotation of the crankshaft and, consequently, the power output of the engine. Finally, the burnt gases are expelled from the cylinder by the upwards momentum of the piston via the exhaust valves (E1, E2) that open and close during the exhaust stroke according to a predetermined process.

When the above process is being simulated, the main governing inputs are the target RPM of the engine and the fuel injection parameters such as quantity (mg / work cycle) and the timing when injection is started (expressed as crankshaft degrees before reaching TDC, BTDC). It should be noted that fuel injection is not a discrete event, but rather a process that has a finite duration. An injection rate expressed in mg/s is dynamically being computed from the input parameters. According to Heywood [60] an approximate estimate for the duration of the injection in crankshaft degrees $\Delta\theta$ can be obtained from the equation:

$$m_f = C_d A_n \sqrt{2\rho_f \Delta p}\frac{\Delta\theta}{360N} \tag{14}$$

Where m_f is the injected fuel quantity, A_n is the injector nozzle area, C_d is a flow discharge coefficient, ρ_f is the fuel density, Δp is the pressure drop across the nozzle and N is the revolution rate in RPM. Consequently, when the total injected quantity is supplied to the simulator, it is able to infer the required static injection rate. This approximation results in a simple box-shaped injection profile as a function of the crank angle, an idealized version of injection strategies used in actual modern engines. While this approximation is sufficient for the example's purposes, it will not produce accurate estimates of exhaust emission characteristic. For this reason, the present example strictly examines only the output power of the engine and not the emissions.

5.4.3 Dynamic engine simulation model

The engine model implemented for the purposes of this experiment is a model of a four cylinder heavy duty diesel engine, similar to that described in Table 1. An overview of the engine's corresponding model on the GT-SUITE map is depicted in Figure 35.

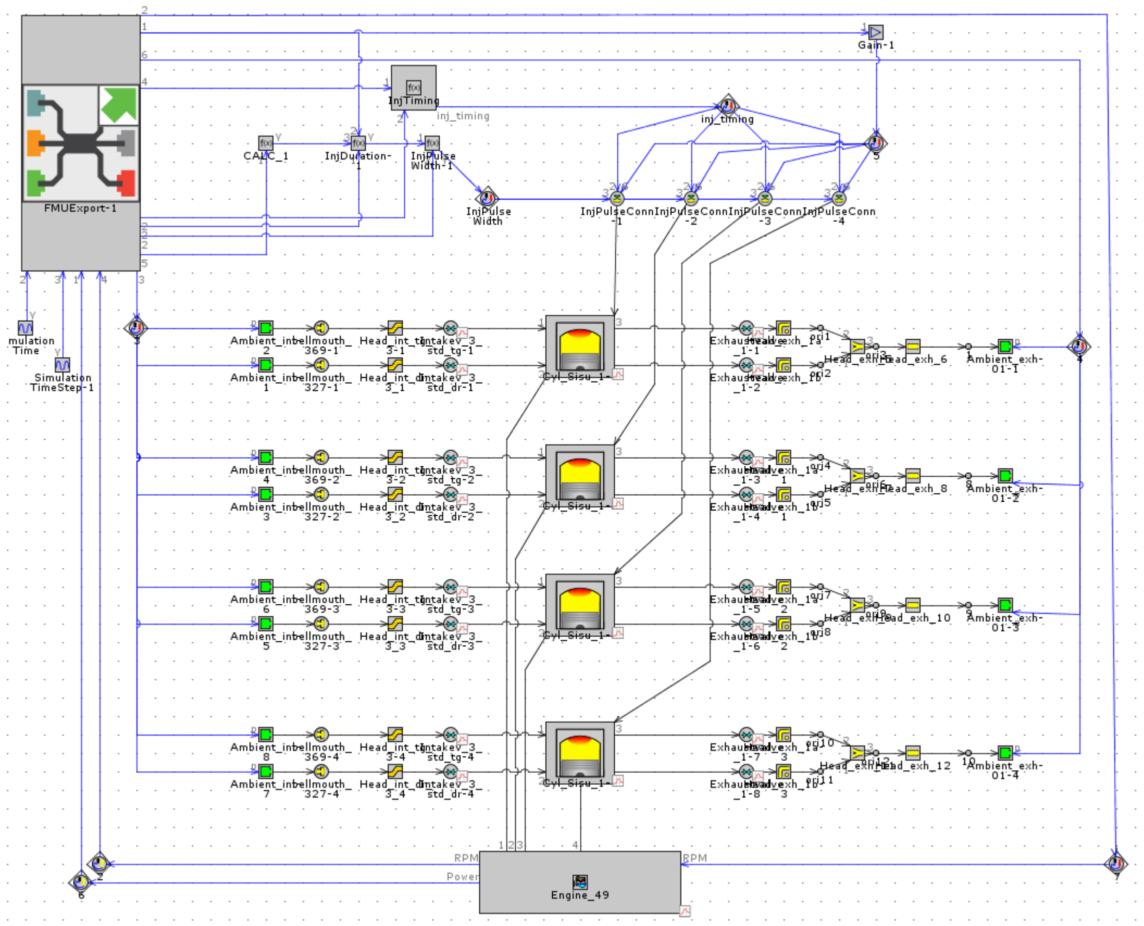


Figure 35. *The implemented GT-SUITE model of the target engine*

As Figure 35 illustrates, the four cylinder components and their flow paths are separately modeled, as well as the fuel injectors connected to each one separately. In the upper left corner, the FMUExport component is displayed, the main functionality of which is to describe the model's I/O mappings, i.e., what data is to be exchanged with the outside world. The FMUExport component also facilitates the actual compiling of the FMU module utilizing GT-SUITE's internal subroutines which, unfortunately, are not disclosed in any further detail by the software vendor.

The control strategy of the model is as follows. At each time step, the model receives as inputs:

- Injection quantity m_f (mg / work cycle).
- Target engine speed N (min^{-1}).
- Boost pressure p_{int}, i.e., intake air pressure at the beginning of the intake ports (bar, abs.).
- ECU timing $\theta_{0,E}$, i.e., injection timing flag originating from the ECU (°BTDC).

- Injection pressure Δp, i.e., the fuel pressure in the common rail system (bar, abs.).
- Exhaust back pressure p_{exh}, i.e., pressure in the exhaust line just before the turbocharger's turbine (bar, abs.).

The outputs are defined in a similar fashion. For the purposes of this experiment, a minimalistic approach is taken. At each time step, the following variable values are exposed to outside agents utilizing the model:

- Engine power (kW).
- Engine speed $(\min^{-1})$.
- Simulated time (s) — cumulative model time from the beginning of the experiment.
- Simulation time step (s) — the mean internal time step used by the ODE solver.

GT-SUITE utilizes ordinary differential equations (ODEs) to dynamically solve the state trajectory (flow and mechanical energy output) of the engine with respect to the inputs. The control logic of the ODE solver works by computing the actual start of injection time θ_0 at each working cycle as:

$$\theta_0 = \theta_{0,E} + AN - B \tag{15}$$

where A, B are coefficients experimentally determined at TUAS ICEL. Subsequently, the duration of injection $\Delta\theta$ is given as

$$\Delta\theta\ [°] = \frac{\omega\ \left[\frac{°}{\mathrm{s}}\right] \cdot m_f\ [\mathrm{kg}]}{C_d A_n [\mathrm{m}^2] \sqrt{2 \cdot \rho_f \left[\frac{\mathrm{kg}}{\mathrm{m}^3}\right] \cdot \Delta p\ \left[\frac{N}{m^2}\right]}} \tag{16}$$

where

$$\omega = \frac{N}{60} \left[\frac{1}{\mathrm{s}}\right] \cdot 360\ [°] \tag{17}$$

Hence, a basic box-shaped injection profile is obtained with start point θ_0, end point $\theta_0 + \Delta\theta$ and a constant injection rate $m_f/\Delta\theta$ expressed in SI-terms as $kg/°$. The obtained injection profile is then superimposed for each cylinder in turn, which causes the crankshaft to rotate and the engine to produce power via the mechanisms discussed in Chapter 5.4.2. The power output by the simulation model can then be compared with that measured from a real engine process, constituting a proof of concept scenario for the contemporary ModelConductor experiment.

5.4.4 Setting up and running the experiment

The inputs that GT-SUITE model expects are described in the FMUExport component's (Figure 35) properties, also allowing the tweaking of some additional settings such as initial value of each variable and the duration the initial value is held, as well as input data interpolation options. No interpolation was used in the contemporary experiment, and the duration at initial output was set to "ignore", with the presupposition that appropriate initial values are provided to the model at runtime.

The export options allow the selection of the compiler to be used alongside with the target environment of the binary executable to be nested in the FMU (Windows 64-bit, Windows 32-bit and Linux are available), as well as selecting the FMI standard version. Additionally, selection between standalone FMU and one requiring a valid GT-SUITE license is provided. Standalone version was selected since it provides more freedom to run experiments on more powerful workstations as well, although it does come with the drawback of not being able to utilize GT-SUITE's GUI and post-processing capabilities at simulation time.

During the export process, stability issues were encountered that required manually altering the FMU built by GT-SUITE's internal subroutine. The first issue was the incorrect placement of a boolean `needsExecutionTool` attribute inside the `modelDescription.xml` file. The FMI specification explicitly defines that this attribute should be located in either the `ModelExchange` [21, p. 89] or `CoSimulation` [21, p. 109] node depending on the operating mode of the FMU. Instead, it was observed that the GT-SUITE process output the attribute directly to parent node `fmuModelDescription`, preventing the FMU to be loaded by external tools such as the FMPY library on which ModelConductor heavily depends. Fortunately, this issue could easily be resolved by simply manually editing `modelDescription.xml` and then recompressing the archive and changing the filename suffix to .fmu.

The apparent second issue was identified to be that two required files `*.dat` and `*.sim` were missing from the resources directory in the compiled FMU package. Instead, they were erroneously written to the root of the working directory during the export process and auto-deleted after exiting the process. Eventually, it turned out to be possible to work around the problem by copying the files to a temporary location *while* the export process was still running and then appending the files to the compiled FMU after the process had exited. Together with the process described earlier to manually alter `modelDescription.xml`, a stable version of the FMU was produced in this manner.

Subsequently, a Python script was developed in a Jupyter notebook environment that was used to invoke the necessary ModelConductor components and start the FMU Co-Simulation session. The script is given in its entirety in Appendix C. Without delving into the details, the script consists of setting the data source, data model and experiment parameters, and finally calling an `Experiment.run` method to start the asynchronous operation of the receive and model loops (Chapter 4.1). While still working in development mode without a GUI to visualize the experiment's progression, ModelConductor's standard debug output is observed during runtime instead which confirms that data is being successfully fetched from the database, fed to the FMU simulation and the model responses are received. For each time step, the values of all the variables (i.e., those denoted by `target_keys`, `input_keys`, `control_keys` and `timestamp_keys`) are written to a plain csv file for further processing. Figure 36 depicts the experiment results, i.e., the engine power simulated by the GT-SUITE based FMU, compared with the control values obtained from the MySQL database.

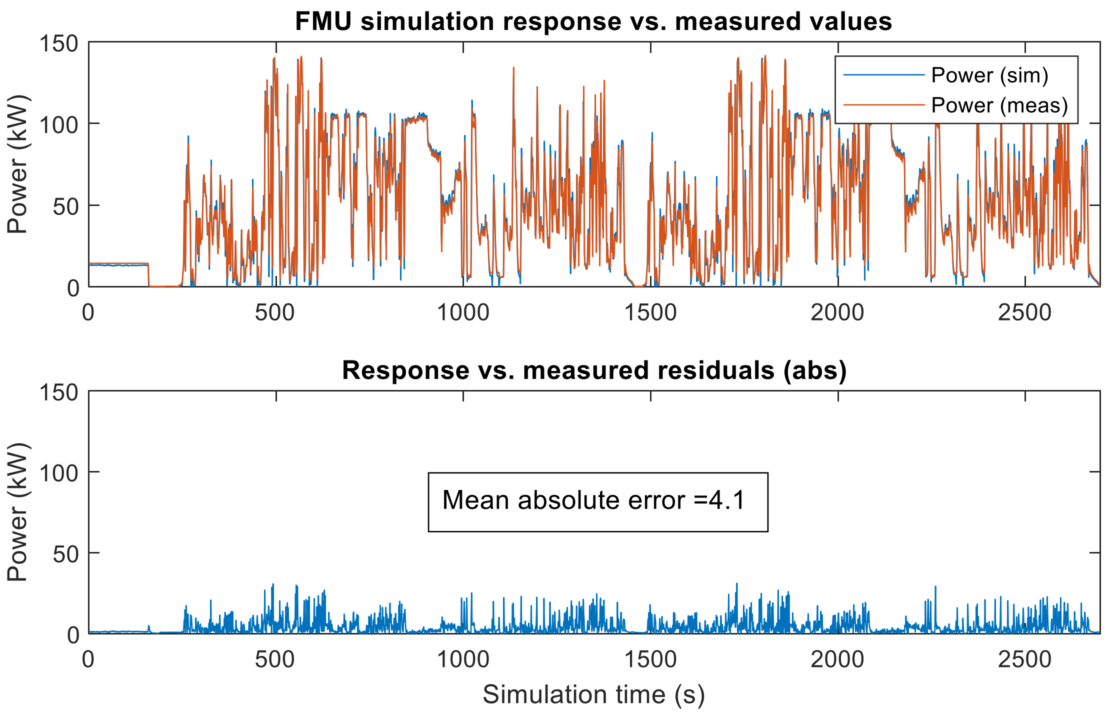


Figure 36. *The simulation results compared with real-world control measurements*

The simulation successfully went through the whole duration expected by the global starting and ending boundaries, i.e., from 2019-10-03 10:15 to 11:00 during little more than 11 hours of wall-clock time. The experiment was run on a commodity laptop workstation (Intel Core i5 2.30 GHz) so the ODE solver performance was not expected to be great and, indeed, it was not. In total, the simulation of this 2700-second experiment took 11 hours, 6 minutes and 53 seconds to finish, making the simulation roughly 15 times slower than real time. Figure 37 illustrates the proceeding of both time axes during the simulation along with t/τ parameter similarly to what was discussed in Chapter 5.3.

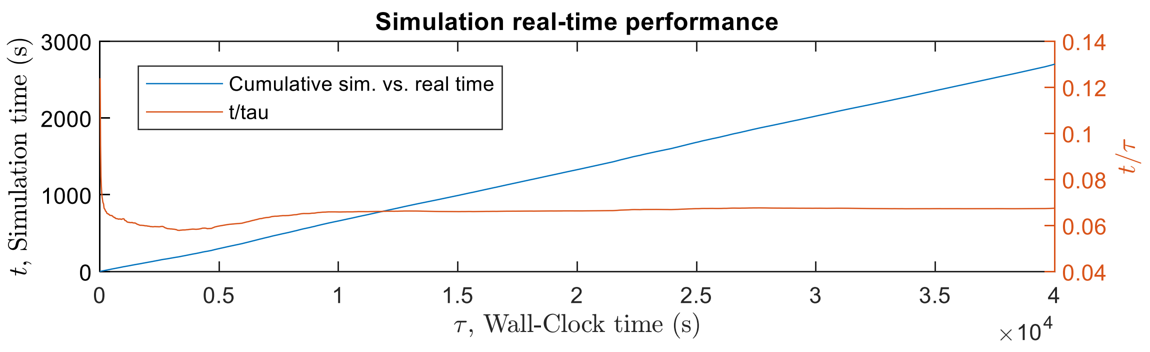


Figure 37. *The passing of simulated time vs. real time during the simulation*

6. DISCUSSION

The aim of the work reported in this thesis was to seek out the essence of software tools that could be used to build, run and evaluate digital twin experiments. The objective was to tackle two distinct research questions, specifically:

- What kind of standards, theoretical constructs and/or "best practices" are reported to facilitate the development of digital twin middleware solutions?
- What features should such a solution entail and how should they be implemented in order to facilitate rapid prototyping and experimentation?

During the process, the subject was discovered to be considerably more elaborate than one would think. A major reason is that the very definition of a digital twin varies wildly across different engineering disciplines. For some, even a simple static CAD model could constitute a digital twin, for some it means an extremely accurate dynamic model of some complex machinery, and some authors even include services and other intangible assets as plausible targets for digital "twinning". Obviously, it follows that only a very narrow segment in the space of all the possible digital twin variants can be appropriately addressed in the scope of a thesis.

Regarding the first research question, an initial demarcation was made to strictly consider dynamical models or ensembles thereof. By this, such digital artefacts are referred to that are able to produce behavior traces by means of simulation, as opposed to static models which do state the structure, but not the behavior of an entity. The review of published literature on digital twins revealed that while research activity on the topic has definitely increased recently, most of the published work can be classified as i) ontological discussions on what digital twins are and what are their implications for organizations or ii) descriptions of specific applications claiming to be digital twins, with varying interpretations of the term itself. To date, few authors have directly attempted to establish a concise theoretical space on digital twins that could directly be utilized in designing digital twin tools. Correspondingly, the subject was, in the theoretical part of the thesis, approached in an indirect manner through the concepts of simulation, co-simulation and tool integration.

The second research question was, to a large extent, addressed via a constructive approach, with Chapter 4 documenting the main features of the empirical work's outcome, namely the ModelConductor library. Subsequently, in Chapter 5 evidence was provided of the developed framework's utility through demonstrating it in action across a variety of use cases, beginning from a simple artificial setup and ending with a complex dynamical model being used in conjunction of a data stream originating from a productional database. Arising as an intrinsic property of the chosen research methodology (i.e., action research), the organizational context of these examples obviously reflects the author's professional affiliation during the period that the work was conducted. However, at the same time the use of general-purpose technologies such as the FMU or MySQL gives rationale for the result's extensibility, at least to some degree.

In the final remaining sections, the main contributions and limitations of the present work are reflected, and consequently some ideas for further study are proposed.

6.1 Contribution

Initially, the work reported in the work's empirical part was necessitated by the requirements of TUAS ICEL to develop tools for sensor data integration onto high-fidelity internal combustion engine simulation models. A prototype of such a tool has now been developed and demonstrated to function satisfactorily in the target organization's context. By no means is the work complete, but specifically the demonstration discussed in Chapter 5.4 does provide strong evidence that the objective of creating an MVP solution has indeed been achieved. The main building blocks for what such a solution should entail have now been laid out, providing a good standpoint for further developments.

On the other hand, the architecture of the developed software MVP demonstrates that the object-oriented approach is suitable for constructing digital twin experiments that make use of online data. A proof of concept has been established that the process and simulation agents, as they were discussed, can be expressed in terms of parent classes that only implement high-level abstractions for the most generic functionalities. The extensibility arising thereof to support a multitude of technologies of individual assets/simulations has been demonstrated by implementations of a few selected cases targeted towards the example experiments. From a software engineering point of view, this implies that similar solutions could, in the future, be built also by professionals with a background in coding, rather than exclusively by domain experts. Furthermore, the example use cases have demonstrated that having implemented the required syntax for the relevant formalisms, the basic tasks of designing, configuring, and running digital twin experiments are straightforward. The script syntax exemplified in notebook examples of Appendices A–C solidifies the justification by which something similar to ModelConductor could become a valuable asset in, e.g., the contemporary machine learning engineer's toolkit.

Finally, it is pointed out that the conducted literature survey and the synthesis constructed of the findings can possibly provide a basis for further theoretical considerations as per how should digital twin frameworks be constructed, used and evaluated. The main contribution of the thesis, in this regard, is the idea of viewing a digital twin as a network of interconnected agents interacting either with physical assets or simulation models. While similar ideas have been proposed before, for instance in the context of High Level Architecture, in the present work it has been described how the discussed theoretical constructs can be derived directly from the theory of co-simulation and tool integration.

6.2 Limitations and future work

Regarding the developed MVP, it is obvious that the software is still in a very early prototype phase. Although reasonable effort was made in the present work to, e.g., ensure that most of the codebase is unit tested, there still are gaps and definitely not all the exceptional situations have been appropriately addressed. While a good level of stability was observed throughout the experiments reported in the work's empirical part, the software should still be considered experimental. While rudimentary readme-files and examples have been provided in the accompanying GitHub repository, anyone considering to use the framework as a part of their project should have a reasonable familiarity with Python syntax in order to ensure the code's suitability for their purposes.

Moreover, it is obvious that the empirical experiments carried out in scope of the thesis represent only the simplest use cases amongst all the conceivable real-world configurations. Most importantly, the functioning of the many-to-many use case has not yet been confirmed by experimental studies, although the MVP architecture in principle does allow this operation mode by means of nested threading. For instance, in the architecture it remains an open question how should communication from one simulation unit to another be facilitated, considering the most general use case where the computations of various submodels could be distributed to multiple workstations. Clearly, this sort of operability would be a prerequisite for moving on to more complex co-simulation setups in the study of online digital twins.

A possible line for future work could be to consider both simulation agents (i.e. the `ModelHandler` class) and process agents (the `MeasurementStreamHandler` class) a subclass of an even more generic Node class, which could implement some high-level interface for node communication by utilizing ModelConductor's internal data model, regardless of the nodes' types. This would allow the discussed complex co-simulation approaches with multiple computation environments, but perhaps even more importantly the development of control-oriented applications wherein data is not only fed from physical assets to simulation models, but also the other way around. This is motivated by recalling that the discussion in this thesis originated from the requirements for intelligent life cycle services, heavily interconnected with disciplines such as AI and deep learning.

Following this discussion, future research efforts should definitely aim for proof of concept demonstrations that a framework such as ModelConductor can be utilized in real-world scenarios in a closed-loop manner. Specifically, evidence is still required from scenarios where the accumulated data over time makes the simulation models more representative and, hence, they are able to provide useful recommendations to their physical counterparts, resulting in some measurable improvement in the physical process.

A somewhat parallel issue is that it still remains unclear how can the contemporality feature discussed in Chapter 2 be realized in practice. In the general case this would mean that the experimental frame of a digital twin experiment is not fixed in the beginning of an experiment. Rather, it should be flexible in the sense that simulation agents can be added, maintained, modified and removed similarly to what can be done for physical machinery. Regarding this, a nontrivial open question for a digital twin framework is how this functionality can be implemented in a manner that is both stable and user friendly in terms of providing a high-level syntax similar to the examples provided in the Appendices A–C. This issue should be addressed in further study in order to provide a more solid basis for digital twin frameworks.

Since physical assets' lifetime could extend to years, open questions remain also regarding the persistence and fault tolerance of a digital twin instantiation within the ModelConductor framework. For obvious reasons, such long-term experiments are challenging to carry out during the scope of a master's thesis. Future study should consider validating the contemporality behavior by running a long digital twin experiment, possibly introducing artificial anomalous situations which could interfere with the process, and making sure that the system is robust enough to recover from those anomalies. Clearly, this would be an enormous task to undertake in a manner that establishes rigorous proofs, but a proof of concept could be realistic to achieve within some specific digital twin context.

The purpose of this thesis has been to shed light on the notion of how digital twins can be built. On final note, it is observed that further theoretical work is required in order to better understand *why should* they be built. Arguably, the construction of any digital artefact can only be justified by its means to improve some physically measurable performance metric, or by its merit to otherwise improve the general well-being in society (including, but not limited to, e.g., educational or entertainment value). In an industrial context it is clear that for digital twin frameworks to gain substantial popularity, a sound theoretical basis should exist that allows practitioners to accurately define digital twins as closed-loop optimization problems with respect to the physical processes' performance, constrained by the associated simulation models' complexity. By extension, such a theory should consider the general many-to-many case, where a multitude of physical processes could be executed, and via the simulation parts of the experiment, optimized in tandem. If developed, such a theoretical tool could be a valuable asset for the research of intelligent fleet management applications, as well as for the more traditional intelligent life cycle services.

REFERENCES

[1] Pricewaterhouse Coopers, “Industry 4.0: Building the digital enterprise,” 2016.

[2] A. Diez-Olivan, J. Del Ser, D. Galar, and B. Sierra, “Data fusion and machine learning for industrial prognosis: Trends and perspectives towards Industry 4.0,” *Inf. Fusion*, vol. 50, no. October 2018, pp. 92–111, 2019.

[3] M. Rüßmann *et al.*, “Industry 4.0: The future of productivity and growth in manufacturing industries,” 2015.

[4] C. Gomes, C. Thule, D. Broman, P. G. Larsen, and H. Vangheluwe, “Co-Simulation: A Survey,” *ACM Comput. Surv.*, vol. 51, no. 3, pp. 49:1--49:33, 2018.

[5] IEEE Standards Association, “IEEE 1516-2010 - IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)-- Framework and Rules,” 2010. [Online]. Available: https://standards.ieee.org/standard/1516-2010.html.

[6] Modelica Association, “Functional Mock-up Interface,” 2019. [Online]. Available: https://fmi-standard.org/. [Accessed: 01-Aug-2019].

[7] E. Glaessgen and D. Stargel, “The Digital Twin Paradigm for Future NASA and U.S. Air Force Vehicles,” in *53rd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics and Materials Conference - Special Session on the Digital Twin*, 2012, pp. 1–14.

[8] G. Nguyen *et al.*, “Machine Learning and Deep Learning frameworks and libraries for large-scale data mining: a survey,” *Artif. Intell. Rev.*, vol. 52, no. 1, pp. 77–124, 2019.

[9] J. Hale, “Deep Learning Framework Power Scores 2018,” *Towards Data Science*, 2018. [Online]. Available: https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a. [Accessed: 02-Aug-2019].

[10] B. Rodič, “Industry 4.0 and the New Simulation Modelling Paradigm,” *Organizacija*, vol. 50, no. 3, pp. 193–207, 2017.

[11] R. Hoda, N. Salleh, and J. Grundy, “The Rise and Evolution of Agile Software Development,” *IEEE Softw.*, vol. 35, no. 5, pp. 58–63, 2018.

[12] R. Stark, S. Kind, and S. Neumeyer, “Innovations in digital modelling for next generation manufacturing system design,” *CIRP Ann. - Manuf. Technol.*, vol. 66, no. 1, pp. 169–172, 2017.

[13] K. Ponomarev, N. Kudryashov, N. Popelnukha, and V. Potekhin, “Main principals and issues of digital twin development for complex technological processes,” *Ann. DAAAM Proc. Int. DAAAM Symp.*, pp. 523–528, 2017.

[14] J. S. DAVID, “Development of a Digital Twin of a Flexible Manufac- Turing System for Assisted Learning,” *Master Thesis*, no. February, 2018.

[15] R. Rosen, G. Von Wichert, G. Lo, and K. D. Bettenhausen, “About the importance

of autonomy and digital twins for the future of manufacturing," *IFAC-PapersOnLine*, vol. 28, no. 3, pp. 567–572, 2015.

[16] F. Tao *et al.*, "Digital twin-driven product design framework," *Int. J. Prod. Res.*, vol. 57, no. 12, pp. 3935–3953, 2019.

[17] B. Schleich, N. Anwer, L. Mathieu, and S. Wartzack, "Shaping the digital twin for design and production engineering," *CIRP Ann. - Manuf. Technol.*, vol. 66, no. 1, pp. 141–144, 2017.

[18] K. Upamanyu and G. Narayanan, "Improved Accuracy, Modelling and Stability Analysis of Power Hardware In Loop Simulation with Open-Loop Inverter as Power Amplifier," *IEEE Trans. Ind. Electron.*, vol. PP, no. c, pp. 1–1, 2019.

[19] S. Huang, W. Wang, M. R. Brambley, S. Goyal, and W. Zuo, "An agent-based hardware-in-the-loop simulation framework for building controls," *Energy Build.*, vol. 181, pp. 26–37, 2018.

[20] J. C. Yepes, M. A. Portela, Á. J. Saldarriaga, V. Z. Pérez, and M. J. Betancur, "Myoelectric control algorithm for robot-assisted therapy: A hardware-in-the-loop simulation study," *Biomed. Eng. Online*, vol. 18, no. 1, pp. 1–29, 2019.

[21] Modelica Association, "Functional Mock-up interface of Model Exchange and Co-Simulation." pp. 1–126, 2014.

[22] A. Banks, E. Briggs, K. Borgendale, and R. Gupta, "MQTT Version 5.0," 2019. [Online]. Available: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html. [Accessed: 06-Oct-2019].

[23] M. D. Myers and D. E. Avison, *Qualitative Research in Information Systems : A Reader*. London, UNITED KINGDOM: SAGE Publications, 2002.

[24] R. N. Rapoport, "Three Dilemmas in Action Research," *Hum. Relations*, vol. 23, no. 6, pp. 499–513, 1970.

[25] P. Järvinen, "Research Questions Guiding Selection of an Appropriate Research Method Pertti Järvinen Research Questions Guiding Selection of an Appropriate Research Method," in *Proceedings of European Conference on Information Systems 2000, 3-5 July.*, 2000.

[26] V. Vaishnavi, B. Kuechler, and S. Petter, "Design Science Research in Information Systems," no. 1, pp. 1–66, 2004.

[27] R. Baskerville, A. Baiyere, S. Gregor, A. Hevner, and M. Rossi, "Design science research contributions: Finding a balance between artifact and theory," *J. Assoc. Inf. Syst.*, vol. 19, no. 5, pp. 358–376, 2018.

[28] J. Venable, J. Pries-Heje, and R. Baskerville, "FEDS: A Framework for Evaluation in Design Science Research," *Eur. J. Inf. Syst.*, vol. 25, no. 1, pp. 77–89, 2016.

[29] J. Venable, J. Pries-Heje, and R. Baskerville, "A Comprehensive Framework for Evaluation in Design Science Research BT - Design Science Research in Information Systems. Advances in Theory and Practice," 2012, pp. 423–438.

[30] J. Pries-Heje, R. Baskerville, and J. R. Venable, "Strategies for Design Science Research Evaluation," in *European Conference on Information Systems*, 2008.

[31] K. Beck *et al.*, “Manifesto for Agile Software Development,” *Manifesto for Agile Software Development*, 2001. [Online]. Available: http://www.agilemanifesto.org/.

[32] P. Aho, “ModelConductor,” 2019. [Online]. Available: https://github.com/donkkis/modelconductor.

[33] S. Raczynski, *Modeling and Simulation : The Computer Science of Illusion*. Somerset, UNITED KINGDOM: John Wiley & Sons, Incorporated, 2006.

[34] H. Vangheluwe, J. de Lara, and P. Mosterman, “An introduction to multi-paradigm modelling and simulation,” *Proc. AIS'2002 Conf.*, 2002.

[35] B. P. Zeigler, *Theory of Modeling and Simulation*. 1984.

[36] W. A. Wymore, *A mathematical theory of systems engineering: The elements*. New York: Wiley., 1967.

[37] H. L. M. Vangheluwe, “DEVS as a Common Denominator for Multi-formalism Hybrid Systems Modelling,” in *Proceedings of the 2000 IEEE International Symposium on Computer-Aided Control System Design*, 2000.

[38] C. Thule, K. Lausdahl, C. Gomes, G. Meisl, and P. G. Larsen, “Maestro: The INTO-CPS co-simulation framework,” *Simul. Model. Pract. Theory*, vol. 92, no. January, pp. 45–61, 2019.

[39] R. Fujimoto, *Parallel and distributed simulation*. 2000.

[40] C. Gomes, C. Thule, D. Broman, P. G. Larsen, and H. Vangheluwe, “Co-simulation: State of the art,” no. February, 2017.

[41] G. Karsai, “Web-Based Open Tool Integration Framework,” 2006.

[42] J. Sztipanovits, T. Bapty, X. Koutsoukos, Z. Lattmann, S. Neema, and E. Jackson, “Model and Tool Integration Platforms for Cyber-Physical System Design,” *Proc. IEEE*, vol. 106, no. 9, pp. 1501–1526, 2018.

[43] J. Sifakis, “System Design Automation: Challenges and Limitations,” *Proc. IEEE*, vol. 103, no. 11, pp. 2093–2103, 2015.

[44] C. Gomes, J. Denil, H. Vangheluwe, and P. G. Larsen, “Co-simulation of Continuous Systems : A Tutorial,” no. ii, pp. 1–32, 2018.

[45] T. Jung and M. Weyrich, “Synchronization of a ‘plug-and-Simulate’-capable Co-Simulation of Internet-of-Things-Components,” *Procedia CIRP*, vol. 79, pp. 367–372, 2019.

[46] G. Karsai, A. Lang, and S. Neema, “Design patterns for open tool integration,” *Softw. Syst. Model.*, vol. 4, no. 2, pp. 157–170, 2005.

[47] J. A. Bergstra and P. Klint, “The discrete time ToolBus — A software coordination architecture,” *Sci. Comput. Program.*, vol. 31, no. 2–3, pp. 205–229, 1998.

[48] G. A. Papadopulos, “Distributed and parallel systems engineering in MANIFOLD,” *Parallel Comput.*, vol. 24, pp. 1137–1160, 1998.

[49] J. S. Dahmann, “High level architecture for simulation,” *Proc. - IEEE Int. Symp.*

Distrib. Simul. Real-Time Appl. DS-RT, pp. 9–14, 1997.

[50] T. Clark, A. Evans, S. Kent, and P. Sammut, “Middlesex Middlesex University Research Repository :,” in *First Workshop on Language Descriptions, Tools and Applications (LDTA 2001)*, 2001.

[51] J. E. Rivera, “On the Semantics of Real-Time Domain Specific Modeling Languages,” University of Malaga, 2010.

[52] G. Karsai, A. Lang, and S. Neema, “Tool Integration Patterns,” in *Proceedings of Workshop on Tool Integration in System Development, European Software Engineering Conference*, 2003, pp. 33–38.

[53] J. S. Dahmann and M. K. L. Morse, “High Level Architecture for Simulation: An Update Dr. Judith S. Dahmann,” *Update*, 1998.

[54] Modelica Association, “Functional Mock-up interface 2.0.” 2014.

[55] D. H. Gelernter, *Mirror Worlds : Or the Day Software Puts the Universe in a Shoebox... How It Will Happen and What It Will Mean*. Cary, UNITED STATES: Oxford University Press, Incorporated, 1993.

[56] Python Software Foundation, “threading - Thread-based parallelism,” 2019. [Online]. Available: https://docs.python.org/3/library/threading.html.

[57] Stackoverflow, “Multiprocessing vs Threading Python,” 2014. [Online]. Available: https://stackoverflow.com/questions/3044580/multiprocessing-vs-threading-python.

[58] Python Software Foundation, “Python Package Index,” 2019. [Online]. Available: https://pypi.org/.

[59] Dieselnet, “Nonroad Transient Cycle (NRTC),” 2019. [Online]. Available: https://www.dieselnet.com/standards/cycles/nrtc.php.

[60] J. B. Heywood, *Internal combustion engine fundamentals*. New York : McGraw-Hill, [1988] ©1988, 1988.

[61] Mathworks, “Export a Model as a Tool-Coupling FMU,” 2019. [Online]. Available: https://se.mathworks.com/help/simulink/ug/export-model-as-tool-coupling-fmu.html.

[62] Dassault Systèmes, “FMPY,” 2019. [Online]. Available: https://fmpy.readthedocs.io/en/latest/.

[63] S. d’Ambrosio, R. Finesso, L. Fu, A. Mittica, and E. Spessa, “A control-oriented real-time semi-empirical model for the prediction of NOx emissions in diesel engines,” *Appl. Energy*, vol. 130, pp. 265–279, Oct. 2014.

[64] R. Vihar, U. Ž. Baškovič, and T. Katrašnik, “Real-time capable virtual NOx sensor for diesel engines based on a two-Zone thermodynamic model,” *Oil Gas Sci. Technol. – Rev. d’IFP Energies Nouv.*, vol. 73, p. 11, Apr. 2018.

[65] J. Chung, H. Kim, and M. Sunwoo, “Reduction of transient NOx emissions based on set-point adaptation of real-time combustion control for light-duty diesel engines,” *Appl. Therm. Eng.*, vol. 137, pp. 729–738, 2018.

[66] R. Finesso, E. Spessa, and Y. Yang, “Fast estimation of combustion metrics in di diesel engines for control-oriented applications,” *Energy Convers. Manag.*, vol. 112, pp. 254–273, 2016.

[67] S. A. Provataris, N. S. Savva, T. D. Chountalas, and D. T. Hountalas, “Prediction of NOx emissions for high speed DI Diesel engines using a semi-empirical, two-zone model,” *Energy Convers. Manag.*, vol. 153, pp. 659–670, 2017.

[68] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. .

[69] DARPA Internet Program, “Transmission Control Protocol.” 1981.

[70] Gamma Technologies, “GT-SUITE Overview.” [Online]. Available: https://www.gtisoft.com/gt-suite/gt-suite-overview/. [Accessed: 01-Sep-2019].

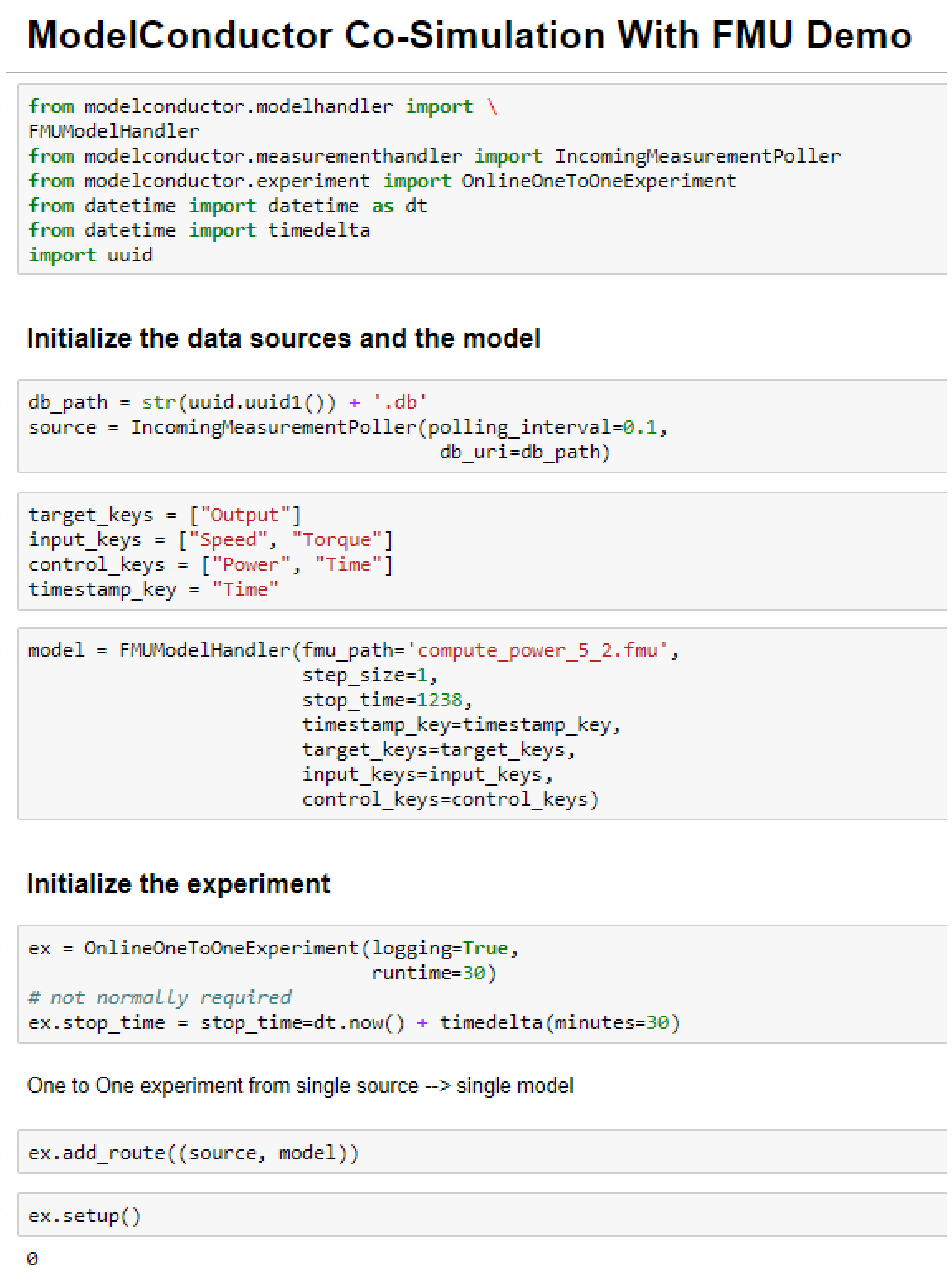
[71] E. Immonen, M. Lauren, L. Roininen, and S. Särkkä, “Multiobjective optimization of the diesel injection rate profile by machine learning methods,” *Unpubl. Manuscr.*, 2019.

[72] K. Främling, J. Holmström, T. Ala-Risku, and M. Kärkkäinen, “Product Agents For Handling Information About Physical Objects,” 2003.

[73] Turku University of Applied Sciences, “e3Power - A systems engineering approach to designing and optimizing hybrid/electric powertrains and vehicle management,” 2019. [Online]. Available: http://www.e3power.fi.

[74] A. J. Martyr and M. A. Plint, *Engine Testing. The Design, Building, Modification and Use Of Powertrain Test Facilities*, 4th editio. Butterworth-Heinemann, 2012.

APPENDIX A: Prediction of engine power using FMU Simulink model



ModelConductor Co-Simulation With FMU Demo

```
from modelconductor.modelhandler import \
FMUModelHandler
from modelconductor.measurementhandler import IncomingMeasurementPoller
from modelconductor.experiment import OnlineOneToOneExperiment
from datetime import datetime as dt
from datetime import timedelta
import uuid
```

Initialize the data sources and the model

```
db_path = str(uuid.uuid1()) + '.db'
source = IncomingMeasurementPoller(polling_interval=0.1,
                                    db_uri=db_path)
```

```
target_keys = ["Output"]
input_keys = ["Speed", "Torque"]
control_keys = ["Power", "Time"]
timestamp_key = "Time"
```

```
model = FMUModelHandler(fmu_path='compute_power_5_2.fmu',
                        step_size=1,
                        stop_time=1238,
                        timestamp_key=timestamp_key,
                        target_keys=target_keys,
                        input_keys=input_keys,
                        control_keys=control_keys)
```

Initialize the experiment

```
ex = OnlineOneToOneExperiment(logging=True,
                              runtime=30)
# not normally required
ex.stop_time = stop_time=dt.now() + timedelta(minutes=30)
```

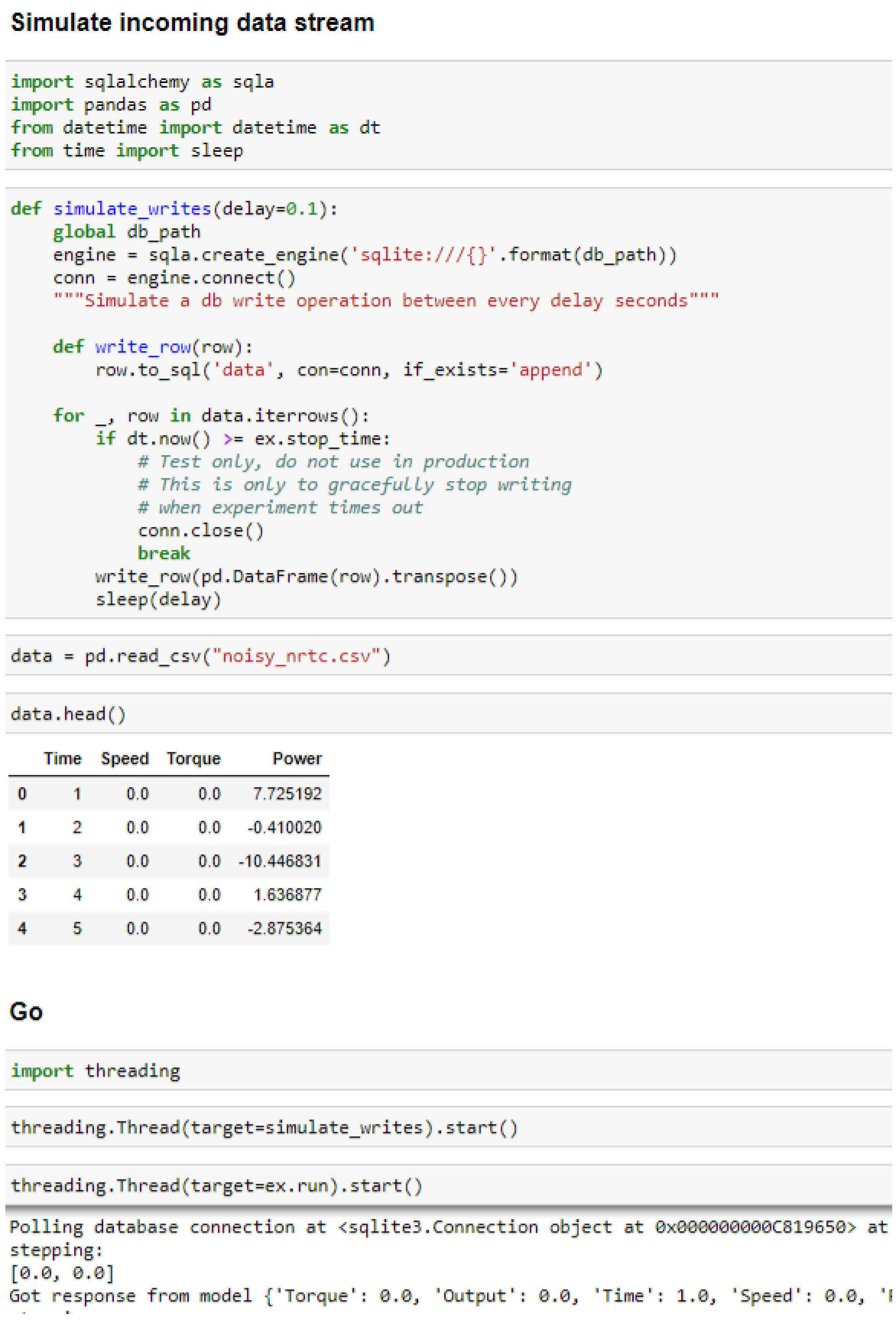
One to One experiment from single source --> single model

```
ex.add_route((source, model))
```

```
ex.setup()
```

```
0
```

APPENDIX A: Prediction of engine power using FMU Simulink model



Simulate incoming data stream

```
import sqlalchemy as sqla
import pandas as pd
from datetime import datetime as dt
from time import sleep
```

```
def simulate_writes(delay=0.1):
    global db_path
    engine = sqla.create_engine('sqlite:///{}'.format(db_path))
    conn = engine.connect()
    """Simulate a db write operation between every delay seconds"""

    def write_row(row):
        row.to_sql('data', con=conn, if_exists='append')

    for _, row in data.iterrows():
        if dt.now() >= ex.stop_time:
            # Test only, do not use in production
            # This is only to gracefully stop writing
            # when experiment times out
            conn.close()
            break
        write_row(pd.DataFrame(row).transpose())
        sleep(delay)
```

```
data = pd.read_csv("noisy_nrtc.csv")
```

```
data.head()
```

	Time	Speed	Torque	Power
0	1	0.0	0.0	7.725192
1	2	0.0	0.0	-0.410020
2	3	0.0	0.0	-10.446831
3	4	0.0	0.0	1.636877
4	5	0.0	0.0	-2.875364

Go

```
import threading
```

```
threading.Thread(target=simulate_writes).start()
```

```
threading.Thread(target=ex.run).start()
```

```
Polling database connection at <sqlite3.Connection object at 0x000000000C819650> at
stepping:
[0.0, 0.0]
Got response from model {'Torque': 0.0, 'Output': 0.0, 'Time': 1.0, 'Speed': 0.0, '
```

APPENDIX B: Online prediction of engine NO_x emissions by machine learning

ModelConductor demo - prediction of engine-out Nox from simulated online measurement stream with a pretrained SkLearn model

Import Modelconductor components

```python
from modelconductor.measurementhandler import IncomingMeasurementListener
from modelconductor.modelhandler import SklearnModelHandler
from modelconductor.experiment import OnlineOneToOneExperiment
```

Setup data source

By default, call to stream.receive starts a TCP socket at port 33003 onto which JSON bytestrings can be relayed by remote clients

```python
stream = IncomingMeasurementListener()
```

Setup model

Load our pretrained random forest regressor model

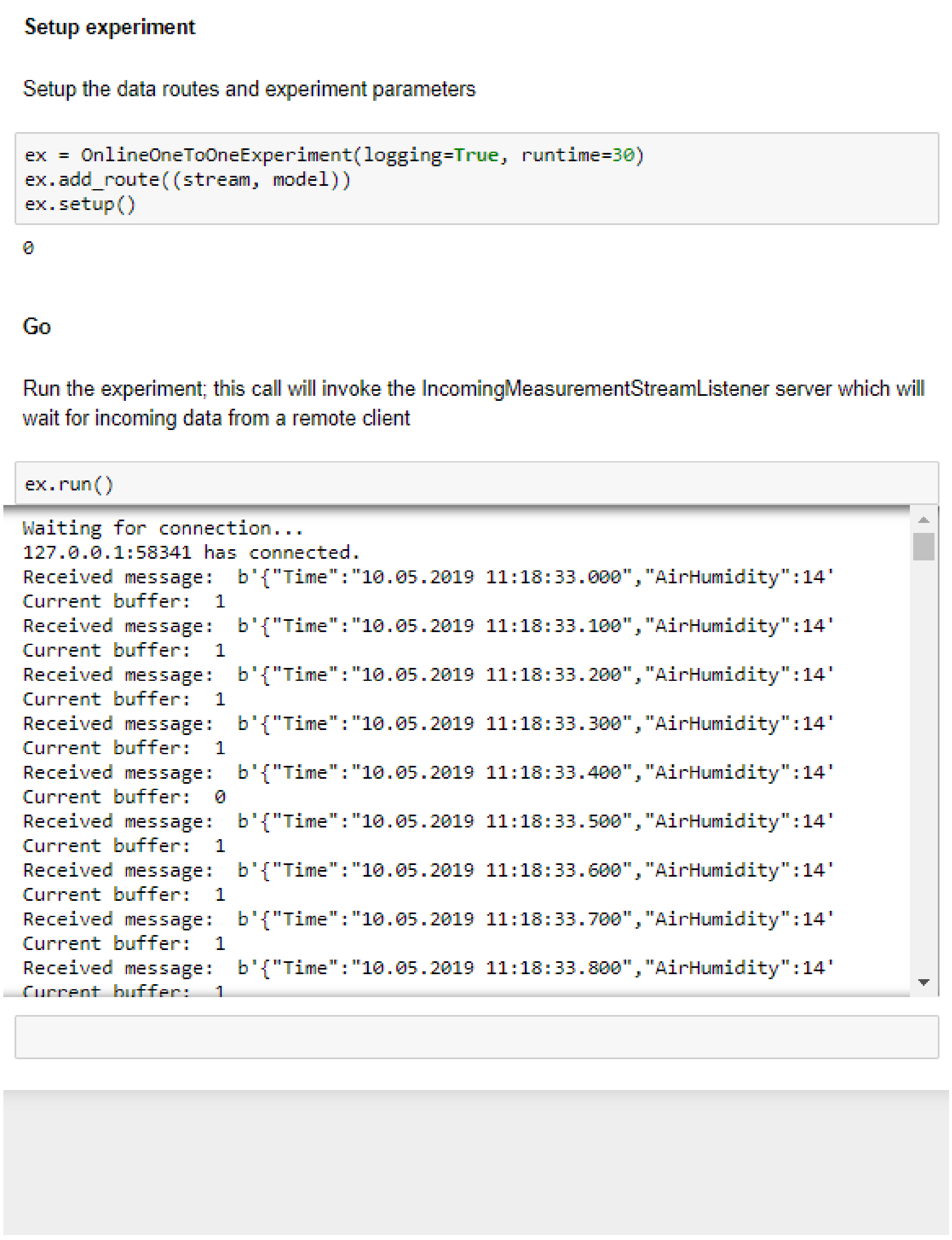
```python
import pickle
# Load input variable names for the pretrained sklearn model
with open('nox_idx.pickle', 'rb') as f:
    idx = pickle.load(f)

input_keys = idx
target_keys = ["Left_NOx_pred"]
control_keys = ["Left_NOx"]

model = SklearnModelHandler(model_filename='nox_rfregressor.pickle',
                            input_keys=input_keys,
                            target_keys=target_keys,
                            control_keys=control_keys)
```

```
C:\Users\Panu\Anaconda3\lib\site-packages\sklearn\base.py:251: UserWarning: Try
ing to unpickle estimator DecisionTreeRegressor from version 0.19.1 when using
version 0.20.0. This might lead to breaking code or invalid results. Use at you
r own risk.
  UserWarning)
C:\Users\Panu\Anaconda3\lib\site-packages\sklearn\base.py:251: UserWarning: Try
ing to unpickle estimator RandomForestRegressor from version 0.19.1 when using
version 0.20.0. This might lead to breaking code or invalid results. Use at you
r own risk.
  UserWarning)
```

APPENDIX B: Online prediction of engine NO_x emissions by machine learning



Setup experiment

Setup the data routes and experiment parameters

```
ex = OnlineOneToOneExperiment(logging=True, runtime=30)
ex.add_route((stream, model))
ex.setup()
```

```
0
```

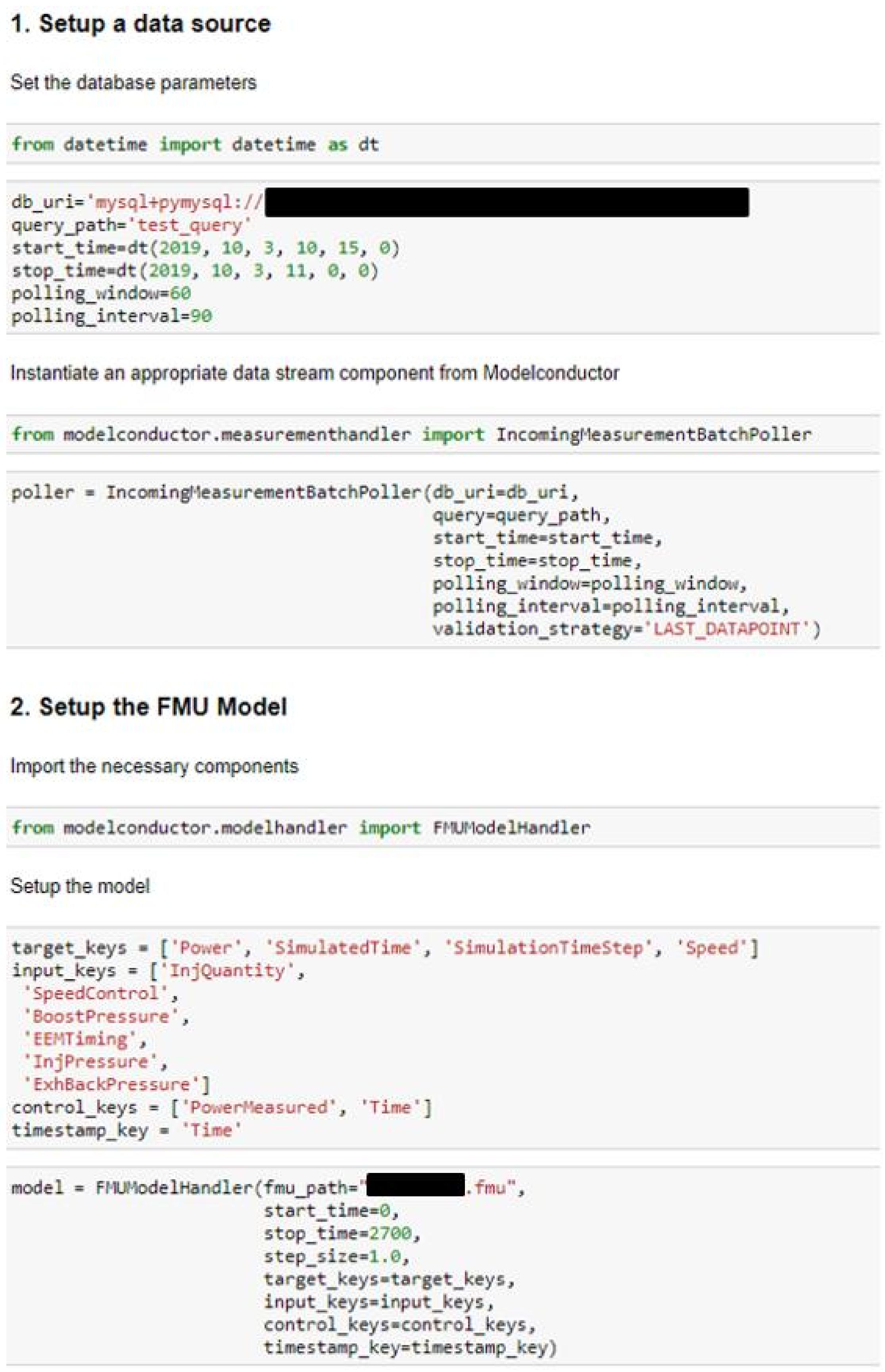
Go

Run the experiment; this call will invoke the IncomingMeasurementStreamListener server which will wait for incoming data from a remote client

```
ex.run()
```

```
Waiting for connection...
127.0.0.1:58341 has connected.
Received message:  b'{"Time":"10.05.2019 11:18:33.000","AirHumidity":14'
Current buffer:  1
Received message:  b'{"Time":"10.05.2019 11:18:33.100","AirHumidity":14'
Current buffer:  1
Received message:  b'{"Time":"10.05.2019 11:18:33.200","AirHumidity":14'
Current buffer:  1
Received message:  b'{"Time":"10.05.2019 11:18:33.300","AirHumidity":14'
Current buffer:  1
Received message:  b'{"Time":"10.05.2019 11:18:33.400","AirHumidity":14'
Current buffer:  0
Received message:  b'{"Time":"10.05.2019 11:18:33.500","AirHumidity":14'
Current buffer:  1
Received message:  b'{"Time":"10.05.2019 11:18:33.600","AirHumidity":14'
Current buffer:  1
Received message:  b'{"Time":"10.05.2019 11:18:33.700","AirHumidity":14'
Current buffer:  1
Received message:  b'{"Time":"10.05.2019 11:18:33.800","AirHumidity":14'
Current buffer:  1
```

APPENDIX C: Real-world engine test data simulation with GT-SUITE



1. Setup a data source

Set the database parameters

```
from datetime import datetime as dt
```

```
db_uri='mysql+pymysql://[redacted]
query_path='test_query'
start_time=dt(2019, 10, 3, 10, 15, 0)
stop_time=dt(2019, 10, 3, 11, 0, 0)
polling_window=60
polling_interval=90
```

Instantiate an appropriate data stream component from Modelconductor

```
from modelconductor.measurementhandler import IncomingMeasurementBatchPoller
```

```
poller = IncomingMeasurementBatchPoller(db_uri=db_uri,
                                        query=query_path,
                                        start_time=start_time,
                                        stop_time=stop_time,
                                        polling_window=polling_window,
                                        polling_interval=polling_interval,
                                        validation_strategy='LAST_DATAPOINT')
```

2. Setup the FMU Model

Import the necessary components

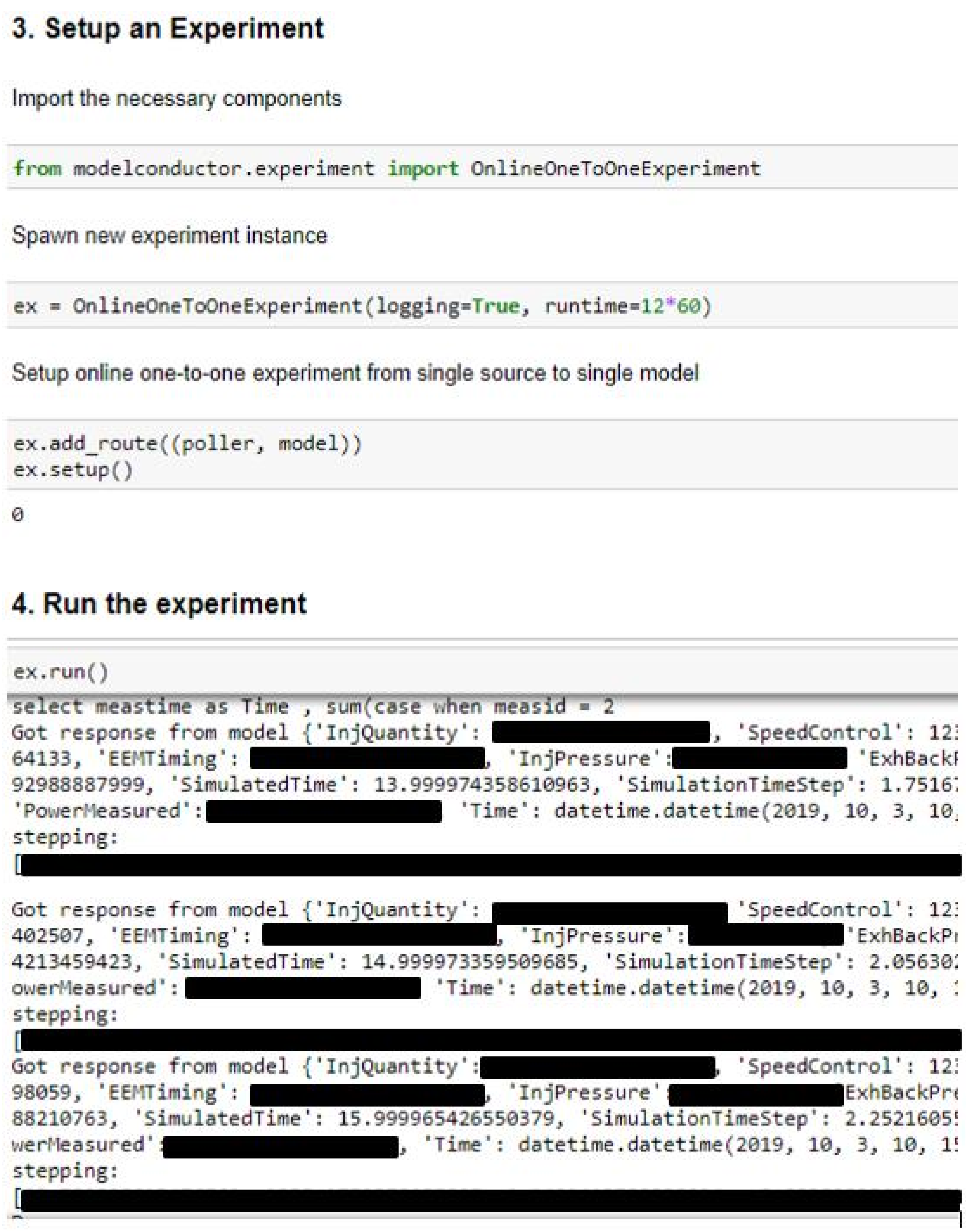
```
from modelconductor.modelhandler import FMUModelHandler
```

Setup the model

```
target_keys = ['Power', 'SimulatedTime', 'SimulationTimeStep', 'Speed']
input_keys = ['InjQuantity',
 'SpeedControl',
 'BoostPressure',
 'EEMTiming',
 'InjPressure',
 'ExhBackPressure']
control_keys = ['PowerMeasured', 'Time']
timestamp_key = 'Time'
```

```
model = FMUModelHandler(fmu_path="[redacted].fmu",
                        start_time=0,
                        stop_time=2700,
                        step_size=1.0,
                        target_keys=target_keys,
                        input_keys=input_keys,
                        control_keys=control_keys,
                        timestamp_key=timestamp_key)
```

APPENDIX C: Real-world engine test data simulation with GT-SUITE



3. Setup an Experiment

Import the necessary components

```
from modelconductor.experiment import OnlineOneToOneExperiment
```

Spawn new experiment instance

```
ex = OnlineOneToOneExperiment(logging=True, runtime=12*60)
```

Setup online one-to-one experiment from single source to single model

```
ex.add_route((poller, model))
ex.setup()
```

```
0
```

4. Run the experiment

```
ex.run()
```

```
select meastime as Time , sum(case when measid = 2
Got response from model {'InjQuantity': [redacted], 'SpeedControl': 12
64133, 'EEMTiming': [redacted], 'InjPressure':[redacted] 'ExhBack
92988887999, 'SimulatedTime': 13.999974358610963, 'SimulationTimeStep': 1.7516
'PowerMeasured':[redacted] 'Time': datetime.datetime(2019, 10, 3, 10
stepping:
[[redacted]

Got response from model {'InjQuantity': [redacted] 'SpeedControl': 12
402507, 'EEMTiming': [redacted], 'InjPressure':[redacted]'ExhBackP
4213459423, 'SimulatedTime': 14.999973359509685, 'SimulationTimeStep': 2.05630
owerMeasured': [redacted] 'Time': datetime.datetime(2019, 10, 3, 10,
stepping:
[[redacted]
Got response from model {'InjQuantity':[redacted], 'SpeedControl': 12
98059, 'EEMTiming': [redacted], 'InjPressure'[redacted]ExhBackPr
88210763, 'SimulatedTime': 15.999965426550379, 'SimulationTimeStep': 2.2521605
werMeasured'[redacted], 'Time': datetime.datetime(2019, 10, 3, 10, 1
stepping:
[[redacted]
```