UNIVERSITÉ DU LUXEMBOURG

PhD-FSTC-2016-43
The Faculty of Science, Technology and Communication

Dissertation

Defence held on 08/11/2016 in Luxembourg

to obtain the degree of

Docteur de l'Université du Luxembourg en Informatique

by

Thomas Hartmann
Born on 13th November 1981 in Kaufbeuren (Germany)

Enabling Model-Driven Live Analytics For Cyber-Physical Systems: The Case of Smart Grids

Dissertation defence committee

Prof. Dr. Nicolas Navet, chairman
Professor, University of Luxembourg, Luxembourg, Luxembourg

Dr. François Fouquet, vice-chairman
Research Associate, University of Luxembourg, Luxembourg, Luxembourg

Prof. Dr. Yves Le Traon, supervisor
Professor, University of Luxembourg, Luxembourg, Luxembourg

Prof. Dr. Jordi Cabot, member
Professor, Universitat Oberta de Catalunya, Castelldefels (Barcelona), Spain

Prof. Dr. François Taïani, member
Professor, Université de Rennes 1, Rennes Cedex, France

Dr. Jacques Klein, expert
Senior Research Scientist, University of Luxembourg, Luxembourg, Luxembourg

Abstract

Advances in software, embedded computing, sensors, and networking technologies will lead to a new generation of smart cyber-physical systems that will far exceed the capabilities of today's embedded systems. They will be entrusted with increasingly complex tasks like controlling electric grids or autonomously driving cars. These systems have the potential to lay the foundations for tomorrow's critical infrastructures, to form the basis of emerging and future smart services, and to improve the quality of our everyday lives in many areas. In order to solve their tasks, they have to continuously monitor and collect data from physical processes, analyse this data, and make decisions based on it. Making smart decisions requires a deep understanding of the environment, internal state, and the impacts of actions. Such deep understanding relies on efficient data models to organise the sensed data and on advanced analytics. Considering that cyber-physical systems are controlling physical processes, decisions need to be taken very fast. This makes it necessary to analyse data in live, as opposed to conventional batch analytics. However, the complex nature combined with the massive amount of data generated by such systems impose fundamental challenges. While data in the context of cyber-physical systems has some similar characteristics as big data, it holds a particular complexity. This complexity results from the complicated physical phenomena described by this data, which makes it difficult to extract a model able to explain such data and its various multi-layered relationships. Existing solutions fail to provide sustainable mechanisms to analyse such data in live.

This dissertation presents a novel approach, named **model-driven live analytics**. The main contribution of this thesis is a multi-dimensional graph data model that brings raw data, domain knowledge, and machine learning together in a single model, which can drive live analytic processes. This model is continuously updated with the sensed data and can be leveraged by live analytic processes to support decision-making of cyber-physical systems. The presented approach has been developed in collaboration with an industrial partner and, in form of a prototype, applied to the domain of smart grids. The addressed challenges are derived from this collaboration as a response to shortcomings in the current state of the art. More specifically, this dissertation provides solutions for the following challenges:

First, data handled by cyber-physical systems is usually dynamic—data in motion as opposed to traditional data at rest—and changes frequently and at different paces. Analysing such data is challenging since data models usually can only represent a snapshot of a system at one specific point in time. A common approach consists in a discretisation, which regularly samples and stores such snapshots at specific timestamps to keep track of the history. Continuously changing data is then represented as a finite sequence of such snapshots. Such data representations would be very inefficient to analyse, since it would require to mine the snapshots, extract a relevant dataset, and finally analyse it. For this problem, this thesis presents a temporal graph data model and storage system, which consider time as a first-class property. A time-relative navigation concept enables to analyse frequently changing data very efficiently.

Secondly, making sustainable decisions requires to anticipate what impacts certain actions would have. Considering complex cyber-physical systems, it can come to situations where hundreds or thousands of such hypothetical actions must be explored before a solid decision can be made. Every action leads to an independent alternative from where a set of other actions can be applied and so forth. Finding the sequence of actions that leads to the desired alternative, requires to efficiently create, represent, and analyse many different alternatives. Given that every alternative has its own history, this creates a very high combinatorial complexity of alternatives and histories, which is hard to analyse. To tackle this problem, this dissertation introduces a multi-dimensional graph data model (as an extension of the temporal graph data model) that enables to efficiently represent, store, and analyse many different alternatives in live.

Thirdly, complex cyber-physical systems are often distributed, but to fulfil their tasks these systems typically need to share context information between computational entities. This requires analytic algorithms to reason over distributed data, which is a complex task since it relies on the aggregation and processing of various distributed and constantly changing data. To address this challenge, this dissertation proposes an approach to transparently distribute the presented multi-dimensional graph data model in a peer-to-peer manner and defines a stream processing concept to efficiently handle frequent changes.

Fourthly, to meet future needs, cyber-physical systems need to become increasingly intelligent. To make smart decisions, these systems have to continuously refine behavioural models that are known at design time, with what can only be learned from live data. Machine learning algorithms can help to solve this unknown behaviour by extracting commonalities over massive datasets. Nevertheless, searching a coarse-grained common behaviour model can be very inaccurate for cyber-physical systems, which are composed of completely different entities with very different behaviour. For these systems, fine-grained learning can be significantly more accurate. However, modelling, structuring, and synchronising many fine-grained learning units is challenging. To tackle this, this thesis presents an approach to define reusable, chainable, and independently computable fine-grained learning units, which can be modelled together with and on the same level as domain data. This allows to weave machine learning directly into the presented multi-dimensional graph data model.

In summary, this thesis provides an efficient multi-dimensional graph data model to enable live analytics of complex, frequently changing, and distributed data of cyber-physical systems. This model can significantly improve data analytics for such systems and empower cyber-physical systems to make smart decisions in live. The presented solutions combine and extend methods from model-driven engineering, models@run.time, data analytics, database systems, and machine learning.

Keywords: Model-driven engineering, Models@run.time, Data analytics, Cyber-physical systems, Internet of Things, Graph databases, Machine learning, Temporal data, What-if analysis, Distributed reasoning

Acknowledgments

This work has been funded by the National Research Fund Luxembourg (grant 6816126) and Creos Luxembourg S.A. under the SnT-Creos partnership program.

The PhD experience goes beyond research, experimentations, and paper writing. It is indeed a challenging life experience that started in August 2013 and which outcome owes much to the support and help of many people.

First of all, I want to express my sincere thanks to my supervisor Prof. Dr. Yves Le Traon for giving me the opportunity to pursue my PhD studies within his group and under his supervision. He always encouraged me, had a permanent confidence in me, and supported me throughout these years. I have learned a lot from his rigorous scientific guidance as a researcher and from his positive, motivating, and open-minded attitude as a team leader. I am equally grateful to my co-supervisor Dr. Jacques Klein for his advice, optimism, countless discussions, guidance, and for always emphasising the bright side of things.

My special thanks goes to my daily advisor Dr. François Fouquet for his patience, advice, and flawless guidance throughout the sometimes daunting world of academia. He taught me how to do research, conduct rigorous experiments, write scientific papers, and always pushed me a step further. I am very happy about the friendship we have built up during the years.

I am grateful to the members of my dissertation committee, Prof. Dr. Jordi Cabot, Prof. Dr. François Taïani, and Prof. Dr. Nicolas Navet, for their time to review my work and for providing interesting and valuable feedback.

My sincere thanks also goes to Yves Reckinger and Robert Graglia from Creos for the many fruitful discussions and the time they found to collaborate with us. It was really useful and rewarding to me to be able to apply my research on a concrete industrial case.

I would also like to express my warm thanks to all current and former members of the SerVal team for the plenty good coffee breaks, discussions, and the great times we had. I wish all of them the very best. In particular, I want to thank Dr. Assaad Moawad, Dr. Grégory Nain, and Matthieu Jimenez for their continuous feedback and proofreading. I also want to express my thanks to all my co-authors and to the people I have been in touch with during my PhD journey and that are not explicitly mentioned here.

Finally, and more personally, I would like to express my heartfelt thanks to my family and my friends for their continuous and unconditional support during these last years. The role they all played in this journey is more important than they can possibly know.

Contents

List of abbreviations and acronyms

AJAX asynchronous JavaScript and XML. 141

AMI advanced metering infrastructure. 6

AMR automated meter reading. 6

AP asynchronous parallel. 54, 56, 59

API application programming interface. 32, 34, 45–47, 49, 50, 53, 56, 57, 62, 64, 90, 92, 134, 135, 141, 154, 156, 192

BAP barrier-less asynchronous parallel. 54, 59

BG base graph. 106

BGL Boost Graph Library. 58

BSON binary JSON. 35, 131

BSP bulk synchronous parallel. 54, 56, 58–60, 74, 149

CASE computer-aided software engineering. 32

CDN content delivery network. 132–135

CDO connected data objects. 73

CEP complex event processing. 24, 57, 59

CGI compact graph index. 69

CMOF Complete MOF. 28

CORBA Common Object Request Broker Architecture. 28

CPS cyber-physical system. 2, 3, 10–15, 23, 31, 33, 34, 42, 45, 47, 48, 53, 58, 66, 72–74, 76, 82, 100, 128, 130, 138, 146, 157, 161, 189, 191

DAG directed acyclic graph. 71

DHT distributed hash table. 132, 135

DSML domain-specific modelling language. 26–28, 31

EBNF Extended Backus–Naur Form. 28

ELT extract-load-transform. 24, 25

EMF Eclipse Modeling Framework. 30–34, 73, 135, 150, 157

EMOF Essential MOF. 28, 32, 33, 147

ETL extract-transform-load. 24, 25, 48

GAS gather, apply, and scatter. 55, 57–59, 74

GMM Gaussian mixture model. 178

GPL general-purpose language. 26

GPS Graph Processing System. 58

GWIM Global World Index Map. 113, 114

HDFS Hadoop Distributed File System. 44–46, 49

HGS Historical Graph Store. 68–70

ICT information and communication technologies. 4

IDE integrated development environment. 33

IoT Internet of Things. 2, 23, 34, 42, 47, 66, 72, 76, 100, 102, 144, 146, 157, 161, 188, 189

ITT Index Time Tree. 112, 114, 116, 118, 119

JSON JavaScript object notation. 35, 54, 89, 95, 112, 131

KDE kernel density estimate. 178

KMF Kevoree Modeling Framework. xv, 32–34, 90, 91, 97, 101, 106, 112, 114, 115, 129, 135–137, 146, 156, 157, 179

KPI key performance indicator. 92, 94

LABS locality aware batch scheduling. 66, 67

LDP large deep prediction. 94

LSB least significant bit. 133

LU large update. 92, 94

LWIM Local World Index Map. 113, 114

LWP large wide prediction. 94

MAD magnetic, agile, and deep. 43, 44

MDA model-driven architecture. 27

MDE model-driven engineering. 14, 15, 26, 27, 29–32, 47, 141, 156

MIW Massive Insertion Workload. xvii, 117, 118

MOA Massive Online Analysis. 75

MOEA multi-objective evolutionary algorithm. 190

MOF meta object facility. xv, 27–29, 72, 89, 147

MSB most significant bit. 132

MTGL MultiThreaded Graph Library. 58

MU minor update. 92, 94

MWG many-world graph. xvii, 101, 105, 108–113, 115–124

OCL object constraint language. 141, 153, 191

OLAP online analytical processing. 25, 42–44, 71, 76, 77

OLTP online transactional processing. 25

OMG Object Management Group. 27, 28

OWL web ontology language. 26

PDF probability density function. 178

PLC powerline communication. 8, 166

PSW parallel sliding window. 56, 59

RDD resilient distributed dataset. 45, 46, 67

RDF resource description framework. 26, 62, 89

SCADA supervisory control and data acquisition. 8

SDP small deep prediction. 94

SIW Single Insertion Workload. xvii, 117, 118

SPC Stream Processing Core. 51, 52

SWP small wide prediction. 94

TAF Temporal Graph Analysis Framework. 68

TG temporal graph. 106–108

TGI Temporal Graph Index. 68

UI user interface. 166, 167

UML Unified Modeling Language. 27, 28, 31, 32, 150, 151, 191

UoW unit of work. 115

WIM World Index Map. 120

XMI XML metadata interchange. 32, 73

XML extensible markup language. 27, 32, 35, 54

List of figures

List of tables

List of algorithms and listings

1

Introduction

This chapter begins with the context of this dissertation, followed by an introduction of smart grids, which are used throughout this thesis as motivation and main case study as well as a representative example of a cyber-physical system. Next, important terminology is defined. Then, this chapter sets out the challenges addressed in this thesis and introduces an approach, named model-driven live analytics, designed to tackle these challenges. Finally, an overview of the contributions and the structure of this thesis is presented.

Contents

1.1 Context

Cyber-physical systems (CPSs) are engineered systems that tightly couple computational algorithms and physical components. Continuous improvements in processing power and device miniaturisation allow us to embed advanced computing and communication capabilities in more and more physical objects, diluting the border between the digital and physical world. CPSs "interact with their environments via sensors and actuators, and monitor and control physical processes, using feedback loops, where physical processes and computations affect each other" [221]. Advances in software, embedded computing, sensors, and networking technologies will lead to a new generation of smart cyber-physical systems that will far exceed the capabilities of today's embedded systems. These systems have the potential to lay the foundations for our critical infrastructures of tomorrow, form the basis of emerging and future smart services, and improve the quality of our everyday lives in many areas. Examples of cyber-physical systems include smart grids, smart buildings and cities, medical systems, robotic systems, self-driving cars, and what is often referred to as Industry 4.0 [222]. Applications of CPSs are expected to have significant societal and economical implications. They will transform the way how we interact with the physical world around us [268], where computation is no longer decoupled from its environment. Cyber-physical systems are closely related to the Internet of Things (IoT) and the terms CPS and IoT are often used interchangeably, depending on the context they are used in [241].

CPSs are entrusted with increasingly complex tasks and are on the cusp of controlling critical processes of our physical world, like it is the case for smart grids or self-driving cars. In order to meet the demands accompanied by such tasks, these systems need to become more and more intelligent [267]. Russel and Norvig describe an intelligent system or agent as "a system that perceives its environment and takes actions that maximize its chances of success" [278]. To be able to make such decisions, these systems need a deep understanding of their environment, internal state, impacts of their actions, and their goals. This relies on efficient data models to organise the sensed data and on advanced analytics to extract conclusions from the examined data. Therefore, CPSs continuously collect data measured by sensors, reason about this data, make decisions based on it, and if necessary take corrective actions via actuators. The ability to analyse the sensed data and to draw conclusions out of it is a major prerequisite for taking smart actions.

Considering that CPSs are controlling critical physical processes, decisions usually need to be taken very fast, *e.g.,* in seconds or even milliseconds. This requires systems to analyse data in live, as opposed to conventional batch analytics. However, the complex nature combined with the massive amount of data generated by such systems impose fundamental challenges. While data in the context of cyber-physical systems has similar characteristics as big data, it has a particular complexity. This complexity results from the complicated physical phenomena described by this data, which makes it difficult to extract a model able to explain such data and its various multi-layered relationships. Existing solutions fail to provide sustainable mechanisms to analyse such data in live.

This is the area this dissertation contributes to. More specifically, this thesis aims

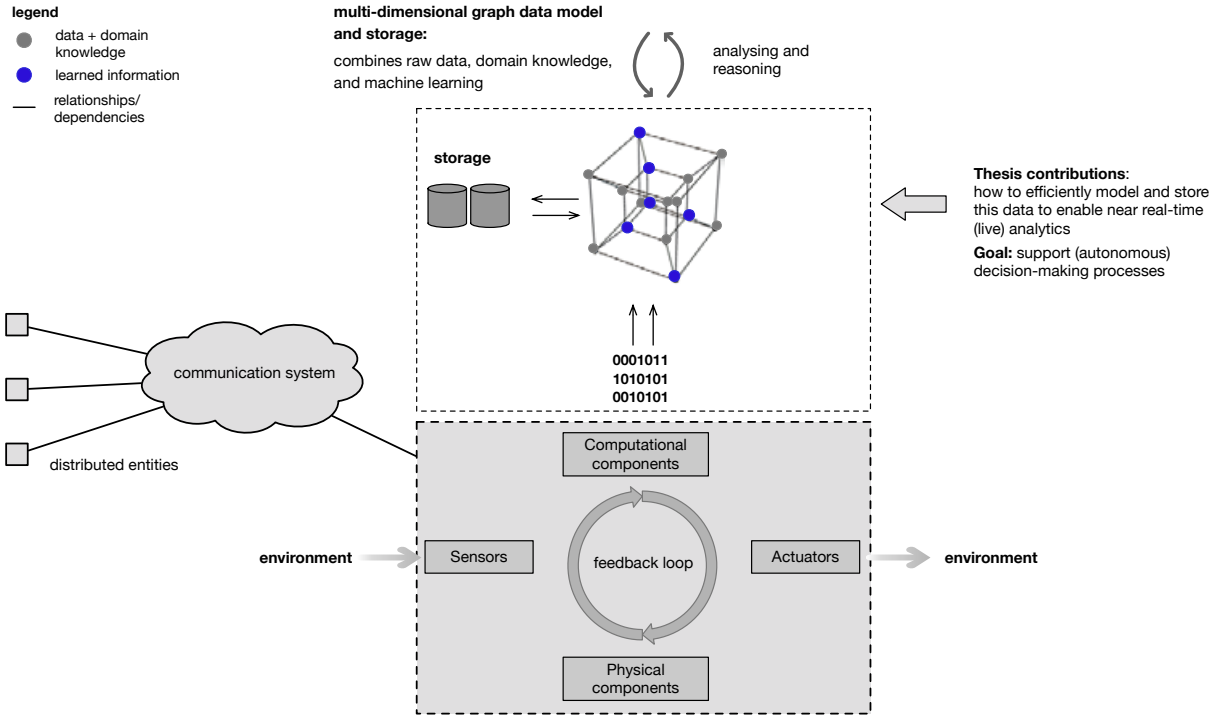


Figure 1.1: Working principle of cyber-physical systems and area of thesis contributions

to provide the means to enable live analytics of complex CPS data, with the goal to support decision-making processes. In the context of this dissertation the terms live and "near real-time" are used in opposition to batch processing, emphasising that computations are expected to be executed "reasonably fast". What reasonably fast is, depends on the application domain. In the context of smart grids, and therefore in this work, near real-time means in the range of seconds. However, no hard real-time guarantees are made. In contrast, Altmeyer *et al.,* [74] discuss challenges and an approach to model and ensure hard real-time requirements, such as those often found in embedded systems. This thesis investigates how complex CPS data can be structured and organised to enable live analytics, given the specific constraints and requirements of such systems. **The main contribution of this thesis is a multi-dimensional graph data model and storage that brings raw data, domain knowledge, and machine learning together in a single model, which in turn can drive live analytic processes**. Figure 1.1 depicts the working principle of cyber-physical systems and shows the area of the contributions of this thesis.

Throughout this dissertation smart grids are used as the main case study. Covering whole countries, smart grids are not just very complex, they are also one of the largest CPSs currently developed. Smart grids allow utilities to collect and analyse massive amounts of data, with the goal to drive operational and customer values from it. This includes, for example, energy theft and anomaly detection, demand response, variable pricing, consumer classification, and advanced monitoring and failure detection. Since these systems are time-critical, it is often not sufficient to collect this data in big data centres and analyse it in batch mode. Instead, decisions about taking action often must be reached quickly and on a local basis, *i.e.,* close to the source of data [277]. In the

literature this is sometimes called "edge computing" [280] or "fog computing" [100]. This, together with the diversity, complexity, and the amount of data collected in smart grids makes them a representative case study for the context of this dissertation.

This thesis has been conducted in an industrial context in collaboration with Creos Luxembourg S.A.[1], the main electricity grid operator in the country. The addressed challenges are derived from this collaboration as a response to shortcomings in the current state of the art (*cf.* Chapter 3). A prototype of the proposed solutions has been developed and applied. The collaboration with Creos made it possible to discuss and refine the research ideas behind this thesis based on actual requirements and, wherever possible, to evaluate them based on data from a real system.

1.2 The smart grid case study

This section first introduces the main ideas and the vision behind smart grids. It then describes smart grids in the context of this thesis and presents a conceptual smart grid model, which is used throughout this dissertation.

1.2.1 The smart grid vision

The traditional electricity grid was designed for the demand of the 20th century. In order to keep pace with the rising demand for energy it must undergo substantial changes. The vision of the smart grid aims to increase efficiency and reliability of the electricity grid by transforming today's centralised electricity grid into a distributed, self-adaptive, and self-healing smart grid of tomorrow. In future, renewable energies and distributed micro-generations are expected to be seamlessly integrated into the electricity grid. This is depicted in Figure 1.2. To enable this, the smart grid emerges as a convergence of information and communication technologies (ICT) with power system engineering [142]. The backbone of this effort is the facilitation of modern ICT to allow two-way communication and an automated control of devices. This underlying communication infrastructure is essential for the smart grid and is what ultimately enables the smart grid to be smart. In fact, it is what allows many advanced features, such as monitoring and diagnostics, remote meter reading, remote control of devices, and demand-side management.

While there is no clear and generally accepted definition of what a smart grid is, there is a consensus that a two-way flow of both information and electricity is the salient characteristic of smart grids [75], [142], [141], [254]. Farhangi [142] describes a number of additional smart grid features commonly mentioned in literature and compares them with the existing electricity grid. This comparison is summarised in Table 1.1. The vision of the smart grid promises a modernised power grid, which is able to meet the increased electricity demand, is more efficient and reliable, decreases brownouts,

[1] http://www.creos-net.lu/start.html

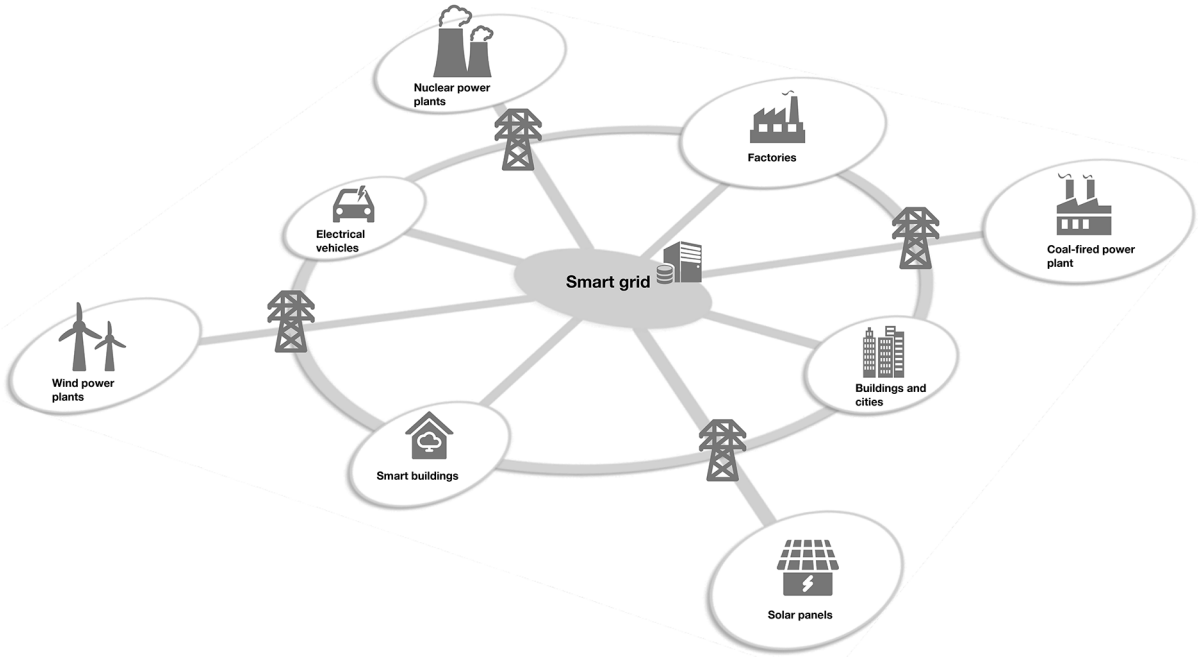


Figure 1.2: The smart grid vision

Table 1.1: Smart grid features compared to the existing electricity grid [142]

Existing electricity grid	**Smart grid**
Electromechanical	Digital
One-way communication	Two-way communication
Centralised generation	Distributed generation
Hierarchical	Network
Few sensors	Sensors throughout
Blind	Self-monitoring
Manual restoration	Self-healing
Failures and blackouts	Adaptive and islanding
Manual check/test	Remote check/test
Limited control	Pervasive control
Few customer choices	Many customer choices

blackouts, and surges whilst reducing costs for energy producers and consumers alike. A power grid, which allows near real-time troubleshooting, autonomously detects issues, and proactively addresses them before they become problems. Such an advanced grid could pave the way for smart devices, smart homes, electric vehicle charging, and would give customers more control over their power bill. It would allow to seamlessly integrate renewable energies and could help to significantly reduce our carbon footprint. However, there is still a long way to go and massive changes are necessary to transform the existing power grid into a smart grid. The main drivers towards this vision are new communication, sensing, and control systems, which are integrated at all levels of the electricity grid. These enable utilities to create new layers of intelligence [118] over current and future infrastructures, by continuously collecting and analysing data throughout the grid, and taking corrective actions when needed.

1.2.2 Smart grids in the context of this thesis

Smart grids are used as the main case study to evaluate the proposed concepts of this dissertation. The objective is to analyse data collected in a cyber-physical system in near real-time, and to ultimately support decision-making processes based on the results of this analysis. In the context of smart grids, it is the communication infrastructure which allows to collect and process the sensed data. It is what enables the cyber part of the smart grid. Therefore, the smart grid communication infrastructure is the primary focus for the context of this thesis. In the following, the relevant entities are introduced.

Smart meters are the cornerstones of the smart grid communication infrastructure. Installed at customers' houses they continuously measure electric consumption, the quality of power supply and regularly report these values to utilities for monitoring and billing purposes. Initially, their main task was essentially automated meter reading (AMR) but they quickly evolved to the so-called advanced metering infrastructure (AMI) [142]. AMI provides a two-way communication system, which allows to send remote commands to meters. Smart meters often control other devices like water and gas meters, or micro generation devices. Usually, these are less powerful devices than smart meters in terms of control abilities, *e.g.,* they have only limited functionalities regarding flow control, compared to the load management of an electrical smart meter. Smart meters are also used as gateways for the smart home [260]. This enables smart home devices to use the communication infrastructure provided by the smart grid. Through AMI, utilities can get nearly instantaneous information about customers' consumption demand. This, together with the ability of smart meters to restrict the maximum allowed consumption and to connect/disconnect specific loads, *e.g.,* electric vehicles, opens the way for intelligent load management. Smart meters can even be used to remotely switch off the electricity of a connected customer.

A second important building block of the smart gird communication infrastructure are so-called *data concentrators*. Data concentrators collect and store consumption data from a number of associated smart meters. Physically, data concentrators are often located at, or near power substations. In regular intervals (typically several times a day, in some settings immediately) they send this data to a *data centre*, where it is

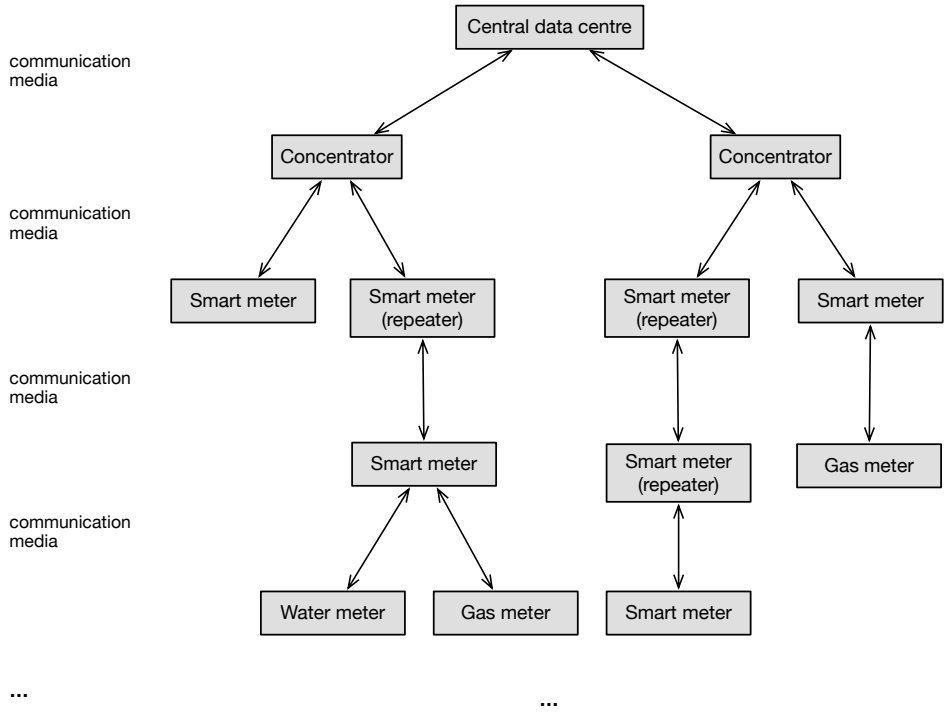


Figure 1.3: Schematic representation of the smart grid communication infrastructure

stored, aggregated and analysed. All smart meters associated to a concentrator are controlled by it. Smart meters can be either directly connected to a data concentrator, or can use any other smart meter as a gateway, *e.g.,* due to long distances, noises, and interference signals. These gateways are regular smart meters. Smart meters acting as gateways are referred to as *repeaters*. Either way, at a time t each smart meter is registered to at most one concentrator, wether directly, or via one or several repeaters. Similar, at a time t, a smart meter which is not directly connected to a concentrator, can use at most one other smart meter as a repeater and so forth. This characteristic forms a communication topology as depicted in Figure 1.3. As shown in the figure, the topology is organised as a tree, where each data concentrator is the root node of a subtree. The individual subtrees are connected to a data centre. Smart meters dynamically reorganise this topology by reconnecting to other smart meters, *e.g.,* if a repeater fails, a connection is broken, or noises disturb the communication. Such changes occur comparatively frequently, *e.g.,* in Luxembourg the average is 30 changes per hour [173]. Concentrators have the ability to send commands, like requesting consumption data or restricting the maximum allowed consumption of a customer. This property divides the smart grid communication infrastructure into smaller, autonomous areas, where each area is controlled by a data concentrator. This is often referred to as the distributed control ability of the smart grid [142]. To do so, compared to smart meters, concentrators are usually equipped with a considerable amount of storage capacity and computation power. Data concentrators are by no means powerful data centres, but have enough storage and computation capacity to perform specific monitoring and analytic tasks. Since data concentrators receive near real-time information about customers' consumption values, as well as grid status updates, and can also control

connected smart meters, data concentrators are ideally suited for analysing this data and taking corrective actions if necessary.

Different communication technologies, wired and wireless, are used for data transmission between the smart grid devices. As discussed in [230], depending on the specific requirements of a smart grid deployment, such as time of deployment, installation and maintenance costs, availability of a technology, or minimum data transfer rate requirements, some technologies are more appropriate than others. Examples are powerline communication (PLC), cellular technologies like GPRS, or internet via copper lines and optical fibre. The smart grid communication topology in Luxembourg is built upon a PLC network. A major advantage of PLC is that the same media that is used for electric power transmission can be reused for establishing the communication network and transmitting data. On the other hand, a major concern with PLC is the amount of electrical noise and disturbances that may be encountered. This requires advanced error detection techniques. Another consequence that results from using PLC is that the communication topology depends on the physical network. Smart grid devices can only communicate with each other, if a physical connection is available. This restricts the possible communication topologies. The communication between data concentrators and a central data centre in Luxembourg relies on GPRS and optical fibre connections.

Figure 1.4 shows a conceptual model of the described smart grid communication infrastructure. In the following we will refer to this model as *"conceptual smart grid model"* or simply *"smart grid model"*. This model contains the relevant notions of smart grids and their relationships in the context of this thesis.

Electricity grids are typically controlled by supervisory control and data acquisition (SCADA) systems, which control electricity production and delivery in real-time. It is the responsibility of these systems to ensure the global stability of the grid. SCADA systems have strong constraints concerning latency to ensure resilience of the grid in case of over-usage, as for example described by Aim *et al.,* [75]. A challenge when designing smart grid infrastructures is the coordination of SCADA systems and the new communication networks across smart meters. SCADA systems typically focus on global electricity production and delivery management, while the new smart grid communication network focuses on local consumption optimisation and management. In the scope of this thesis the proposed solutions are applied aside from already existing SCADA systems. The goal of this work is to leverage the real-time data, which recently becomes available due to the new smart grid technologies, as well as to provide additional monitoring and control abilities on a higher lever, rather than replacing already existing control systems.

1.3 Terminology

This section defines the recurring terminology used in this dissertation. Further details and additional definitions are provided in Chapter 2 of this manuscript.

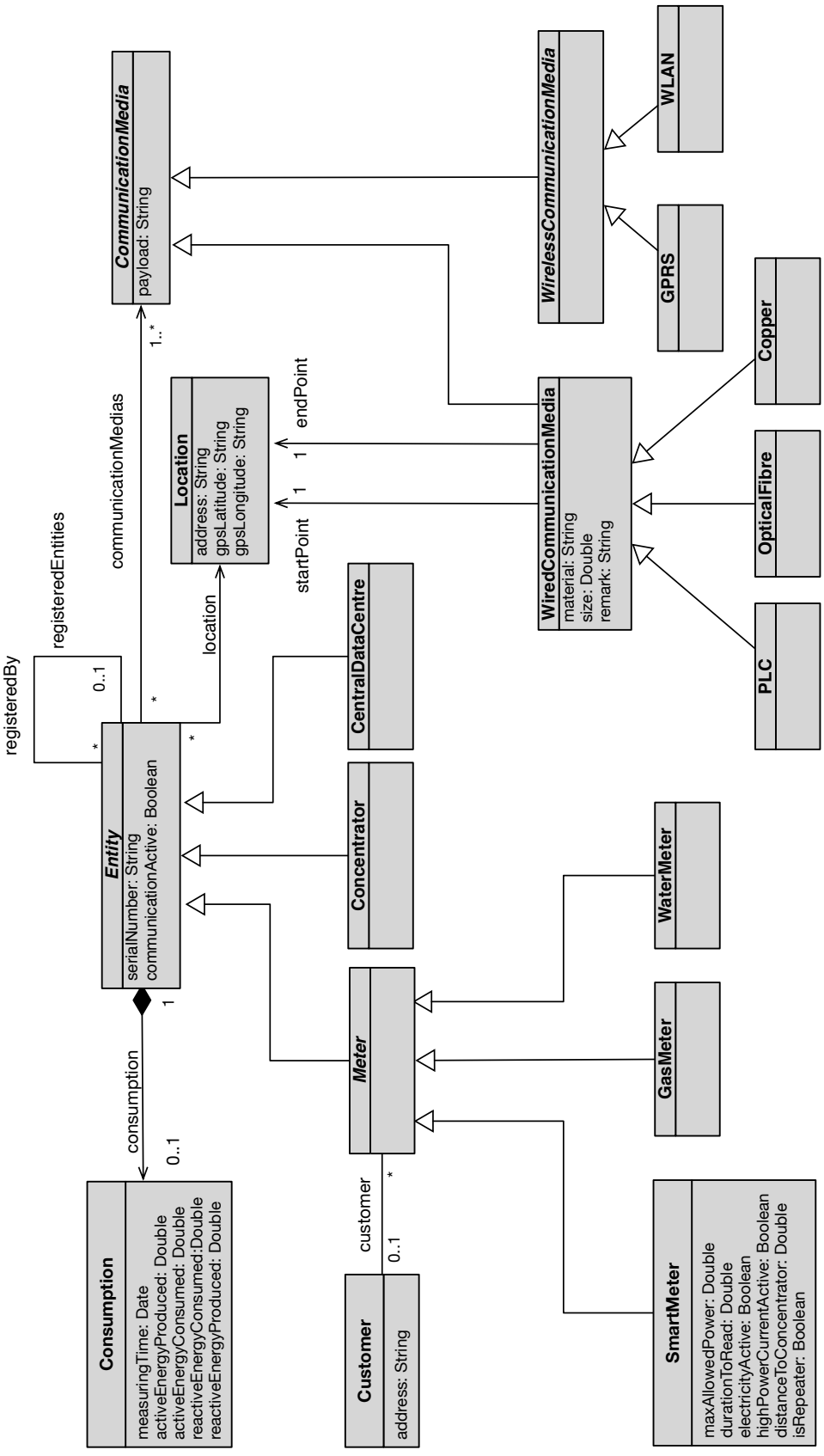


Figure 1.4: Conceptual smart grid model

- The term **model** denotes to an abstraction or simplification of a subject one wants to reason about in some form or another [271]. A subject can be something from the real world, or something virtual. Models are simplifications in the sense that they only represent a certain part of the subject, *i.e.,* they limit the scope of a subject to the relevant parts, given a certain purpose [286]. While models are conceptual constructs, a concrete instance of a model consists of data structures, which implement the semantic of the model. In the context of this thesis, models are used to create abstractions of cyber-physical systems.

  A famous definition of the term model was coined by Jeff Rothenberg, who defines a model as follows: *"A model represents reality for the given purpose; the model is an abstraction of reality in the sense that it cannot represent all aspects of reality. This allows us to deal with the world in a simplified manner, avoiding the complexity, danger, and irreversibility of reality"* [274].

  Another often cited definition is the one from Seidewitz: *"A model is a set of statements about the system under study"* [285].

- A **meta model** defines the structure of a model. A meta model is a model itself, which makes statements about what can be expressed in valid models [285]. It offers the vocabulary for formulating reasonings on top of a given model. Both, models and meta models are models but have different purposes. A meta model describes a model [271]. The term "meta" is relative in the sense that, depending on the perspective, a model can be a model or a meta model. An instance of a meta meta model is a meta model, the instance of a meta model is a model and so forth.

  Seidewitz provides the following definition of a meta model: *"A metamodel is a specification model for which the systems under study being specified are models in a certain modeling language"* [285].

- **Runtime models** are models used during the execution of a system. This dissertation refers to such models as **models@run.time**. A runtime model represents a system under study and is linked in the sense that a runtime model constantly mirrors the system and its current state and behaviour [96]. Models@run.time are used during the execution of a system to reason about its state.

  Blair *et al.,* define a model@run.time as: *"A model@run.time is a causally connected self-representation of the associated system that emphasizes the structure, behavior, or goals of the system from a problem space perspective"* [96].

- **Modelling** denotes to the activity of creating models of a subject.

- Respectively, **meta modelling** refers to the process of creating meta models of a subject.

- Following the context definition of Dey and Abowd [70], in this thesis the internal state together with the surrounding environment of a CPS is referred to as **context** or **system context**. In this dissertation we use models, or more specifically, models@run.time, to represent the context of CPSs and to reason about it.

1.4 Challenges

1.4.1 Overview

In today's complex cyber-physical systems data is generated in very large scale by a wide range of sources, such as sensors and embedded devices [191]. As it turns out, the complexity of this data is at least as much of a challenge in gaining profound insights as is the sheer size of it. In fact, even relatively small but complex datasets can be difficult to analyse. In the context of CPSs the complexity results from the complicated physical phenomena described by this data, which makes it difficult to extract a model able to explain such data and its various multi-layered relationships.

Furthermore, these systems are expected to react quickly, which requires fast, *i.e.,* in near real-time, data analysis. In addition to that CPSs usually have only limited computational capabilities. Even though—driven by advances in embedded systems and their falling prices—the computing power of CPSs is getting more and more powerful, they can most of the time not rely on big clusters and batch processing for their analytic tasks.

The high complexity of CPS data makes it necessary to properly structure and organise data to efficiently analyse it. This requires appropriate models, able to represent the context (internal state and surrounding environment) of a cyber-physical system [181]. Building, storing, and loading appropriate context models to enable efficient live analytics for CPSs is challenging [261]. Existing solutions fail to provide sustainable mechanisms to analyse such data in live.

In the following, an overview of the main challenges addressed in this thesis is presented. In the contribution part of this dissertation each challenge is then discussed in detail. Each challenge corresponds to a concrete need encountered during the collaboration with Creos Luxembourg S.A.

1.4.2 Challenges addressed in this thesis

Analysing data in motion. Data handled by cyber-physical systems is usually dynamic, *i.e.,* it is constantly changing [191], [224]. This is also known as *data in motion*, as opposed to traditional *data at rest*, or as *temporal data* [186]. For example, physical quantities measured by sensors, such as temperature, pressure, speed, and distance are inherently temporal. Moreover, data in CPSs often changes frequently and at very different paces. It is usually not enough to only consider the current data. Instead, reasoning processes typically need to analyse and compare data from the current context with its history [188], [171], [202]. For instance, predicting the electric load for a particular region requires a good understanding of the past electricity production and consumption in this region, as well as recent data, such as current and forecasted weather conditions. However, data models can usually only reflect the context of a CPS at a given point in time, *i.e.,* they only represent a snapshot of a real system at one specific timestamp. Such discretisation leads to a representation of

temporal context data as a finite sequence (potentially distributed) of snapshots (*e.g.,* proposed by [188], [171]). As a consequence, the state of a context model between two snapshots is not defined. This results in losing the semantic of continuously evolving data [289]. To address this problem, it is a common approach to regularly sample and store the context of a system at a very high rate in order to provide analytic algorithms with enough historical data. In order to correlate data from different timestamps, analytic algorithms then need to mine a huge amount of snapshots, extract a relevant dataset, and finally analyse it (*e.g.,* [239], [188], [171]). This requires heavy resources and/or is time-consuming, which stands in conflict with the near real-time response time requirements such systems need to meet.

Challenge #1:
One of the major challenges addressed in this thesis is how data models and associated storage systems can be organised to offer reasoning algorithms an efficient, coherent, and consistent view of temporal data.

Exploring hypothetical actions. Making sustainable decisions requires to anticipate the possible impacts of actions. The exploration of what might happen if this or that action would be taken is referred to as *what-if analysis* [167], [41]. Every action triggers effects which potentially lead to an alternative state from where a set of other actions can be applied and so forth. Considering complex systems, like cyber-physical systems, it can come to situations where hundreds or thousands of alternative actions must be explored before a solid decision can be made (*e.g.,* optimisation and planning tasks [138]). For example, in case of a potential overload situation the smart grid would need to explore numerous chains of different actions, like restricting the maximum allowed load for certain customers or regulating the charging of electric cars, to finally decide for the most appropriate chain of actions. Every action can be interpreted as a divergent point leading to an independent alternative. What-if analysis simulates different actions and tries to find the sequence of actions which leads to a desired alternative [167], [41]. The usefulness of simulating actions based on models in the context of CPSs has, for example, been shown by Fejoz *et al.,* [143] (in this case using the CPAL language [249]). An alternative can be interpreted as a snapshot of a system's context. In order to simulate different chains of actions, every alternative needs to be able to evolve independently—both in space, *i.e.,* leading to additional alternatives, and in time. This leads to different histories in different alternatives, creating a very high combinatorial complexity of alternatives and temporal data.

Challenge #2:
The second major challenge addressed in this dissertation is how data models and associated storage systems can be organised to allow an efficient exploration of a large number of independent alternatives—in space and time—even on a massive amount of data.

Reasoning over distributed data in motion. CPSs are not just getting more and more large-scale and complex but are also increasingly equipped with distributed control and decision-making abilities [267], [221]. Reasoning over distributed data is a complex task [148] since it relies on the aggregation and processing of various dis-

tributed and constantly changing data [232]. In fact, to fulfil their tasks, these systems typically need to share context information between computational nodes (any computer system reading, writing, or processing data in the context of a CPS). Therefore, appropriate data models used to analyse data in a CPS must support such distribution. Data models of complex CPSs can get very large, which makes sharing this information efficiently challenging. For example, the state of a smart grid is continuously updated with a high frequency from various sensor measurements (like consumption or quality of power supply) and other internal or external events (*e.g.,* overload warnings). In reaction to these state changes different actions can be triggered. However, reasoning and decision-making processes are not centralised but distributed over smart meters, data concentrators, and a central system [142], making it necessary to share context information between these nodes. Smart grids, depending on the size of a city or country, can consist of millions of elements and thousands of distributed computational nodes. This challenges the efficiency of sharing context information, especially when taking the near real-time requirements such systems usually need to meet into account.

Challenge #3:
How to handle large-scale, distributed, and frequently changing data to enable efficient analytics over distributed data is the third challenge addressed in this thesis.

Combining domain knowledge with machine learning. In order to meet future needs CPSs need to become increasingly intelligent [267]. On the one hand, some situations CPSs will face are predictable at design time. For example, to react to critical overload situations, the maximum allowed load for customers could be restricted or the charging of electric cars could be balanced accordingly. On the other hand, such systems will also face events that are unpredictable at design time. For instance, the electric consumption of a house depends on the number of people living there, their activities, weather conditions, used devices, and so forth. Despite such behaviour is unpredictable at design time, it is known at design time that this behaviour is unknown [298] and that it can be learned later by observing past situations, once data becomes available. Machine learning algorithms can help to solve this unknown behaviour by extracting commonalities over massive datasets. However, in cases where datasets are composed of independent entities (so-called system of systems [97]) which behave very differently, finding one coarse-grained common behavioural model can be difficult or even inappropriate. For example, the consumption of a factory follows a very different pattern than the consumption of an apartment. Searching for commonalities between these entities would not lead to correct conclusions. Instead, following a "divide and conquer" strategy, learning on finer granularities can be considerably more efficient for such problems [330], [144]. However, learning on fine granularities leads to many fine-grained learning units, which must be synchronised and composed to express more complex behavioural models. Therefore, this requires an appropriate structure to model such learning units and their relationships to domain knowledge.

Challenge #4:
The last challenge addressed in this thesis is how domain knowledge and machine learning can be seamlessly combined to improve data analytics for cyber-physical systems.

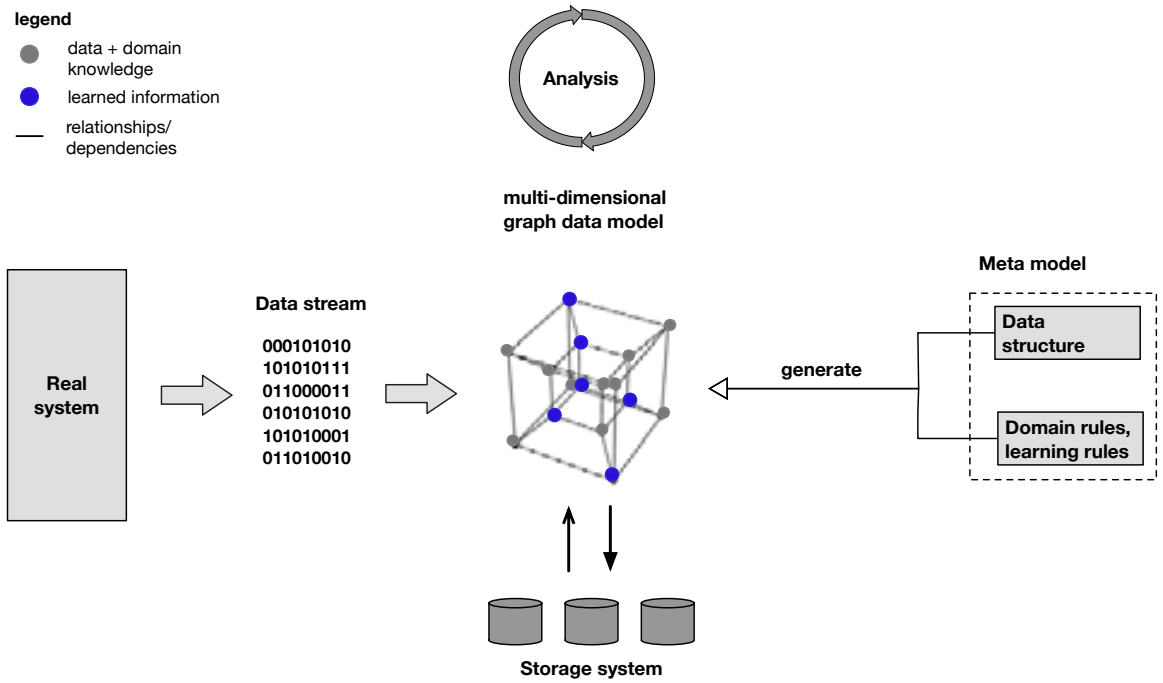


Figure 1.5: Model-driven live analytics

1.5 Approach: model-driven live analytics

In order to address the challenges related to live analytics of data arising in the context of CPSs, this dissertation presents a novel model-based approach to structure and process this data for efficient analytics. The proposed approach combines methods from model-driven engineering (MDE), the models@run.time paradigm, database systems, and machine learning in order to analyse this data in live and turn it into actionable insights. MDE suggests to focus on creating conceptual models of all topics related to a specific problem [195]. These models capture the knowledge of domain experts and can then be exploited in the process of solving this problem. This thesis seeks to extend the applicability of MDE to bring it to another domain: data analytics. The models@run.time paradigm has shown the potential of models to be used during runtime to represent the context of CPSs and to reason about it [96], [246]. Following this paradigm, in this dissertation runtime models are used to structure data of CPSs in order to provide analytic algorithms with efficient abstractions of these systems. These abstractions are then used to drive analytic processes. Modern database technologies, like NoSQL storage systems and graph database methods, are applied to create an efficient storage system for the proposed data model. Machine learning algorithms are seamlessly integrated into this data model. With model-driven live analytics we promote the usage of models as the centre element of analytic processes, where data is structured, analysed, and stored based on a model. This is depicted in Figure 1.5. Model-driven live analytics suggests to connect raw data, domain knowledge, and machine learning into a single structure, which can be leveraged by analytic processes to transform raw data into valuable or actionable insights. This thesis aims to provide the means to build and store such a structure.

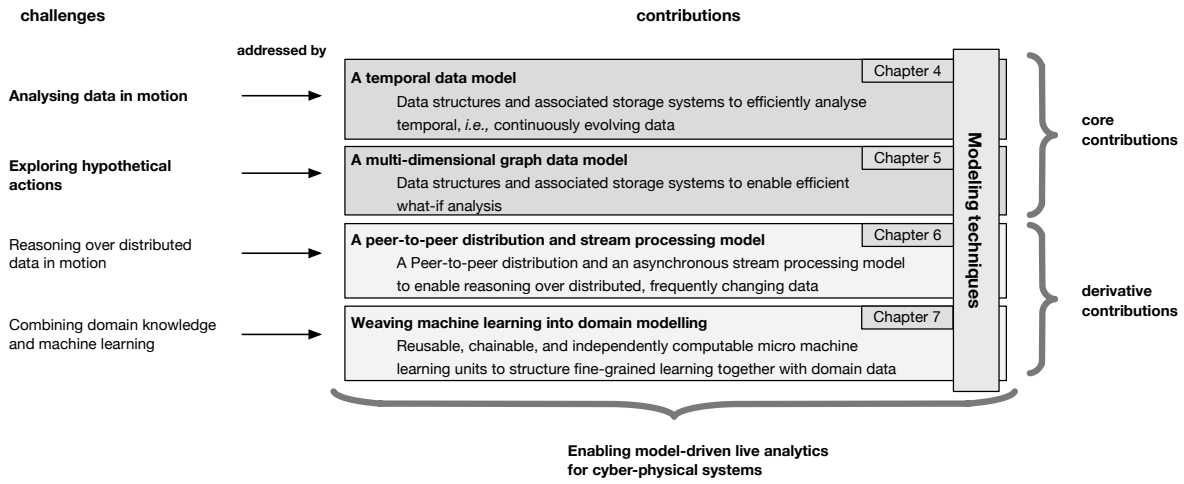


Figure 1.6: Thesis contributions

1.6 Contributions

This dissertation addresses the above-mentioned challenges by extending the applicability of MDE and models@run.time to the domain of live analytics for CPSs. The hypothesis behind this dissertation is that complex and frequently changing data of CPSs can be efficiently, *i.e.,* in near real-time, analysed by organising them in a data model connecting raw data, domain knowledge, and machine learning. This hypothesis is evaluated against a concrete smart grid case study. Figure 1.6 depicts the concrete contributions made in this thesis and shows which of the challenges described in Section 1.4 are addressed by each of these contributions. In the following, a short overview is provided about each contribution.

A temporal data model. The first contribution of this dissertation addresses the challenge of representing and storing temporal, *i.e.,* continuously evolving data. Therefore, it defines a *temporal data model* together with a time-relative navigation concept. The proposed approach considers time as a first-class property crosscutting any context element and any relation between context elements. This contribution also defines an efficient storage concept for the proposed temporal data model. The goal of this approach is to provide analytics with data structures to efficiently reason about massive amounts of continuously evolving data.

This contribution is based on the work that has been presented in the following papers:

- Thomas Hartmann, François Fouquet, Grégory Nain, Brice Morin, Jacques Klein, Olivier Barais, and Yves Le Traon. A native versioning concept to support historized models at runtime. In *Model-Driven Engineering Languages and Systems - 17th International Conference, MODELS 2014, Valencia, Spain, September 28 - October 3, 2014. Proceedings*, pages 252–268, 2014

- Thomas Hartmann, François Fouquet, Grégory Nain, Brice Morin, Jacques Klein, and Yves Le Traon. Model-based time-distorted contexts for efficient temporal reasoning.

In *The 26th International Conference on Software Engineering and Knowledge Engineering, SEKE 2014, Vancouver, BC, Canada, July 1-3, 2014.*, pages 746–747, 2014 (best paper award)

A multi-dimensional graph data model. The second contribution of this thesis tackles the challenge of simultaneously exploring different hypothetical actions. It extends the temporal data model to a *multi-dimensional data model* able to reflect a large number of different alternatives. The suggested data model allows each alternative to evolve independently with its own independent history in order to enable the simultaneous exploration of many different actions. This contribution aims to define an efficient data model able to enable what-if analysis for a large number of independent actions even on a massive amount of (temporal) data.

This contribution is based on the work that has been presented in the following paper:

- under submission at ACM/USENIX EuroSys 2017: Thomas Hartmann, Assaad Moawad, Francois Fouquet, Gregory Nain, Romain Rouvoy, Yves Le Traon, and Jacques Klein. PIXEL: A Graph Storage to Support Large Scale What-If Analysis

A peer-to-peer distribution and stream processing model. A third contribution of this dissertation copes with the challenge of data analytics over massively distributed datasets of frequently changing data. It proposes an approach to transparently *distribute the suggested data model* in a peer-to-peer manner and defines a *stream processing* method to efficiently handle frequent changes. More specifically, it combines ideas from reactive programming, peer-to-peer distribution, and large-scale modelling. The objective of this contribution is to enable efficient analytics over distributed datasets of frequently changing data.

This contribution is based on the work that has been presented in the following paper:

- Thomas Hartmann, Assaad Moawad, François Fouquet, Grégory Nain, Jacques Klein, and Yves Le Traon. Stream my models: Reactive peer-to-peer distributed models@run.time. In *18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MoDELS 2015, Ottawa, ON, Canada, September 30 - October 2, 2015*, pages 80–89, 2015

Weaving machine learning into domain modelling. The fourth and last contribution of this thesis addresses the challenge of modelling and combining domain data and knowledge together with machine learning. It defines so-called *micro learning units*, which decompose learning tasks into reusable, chainable, and independently computable units. The concept presented in this approach extends data models with the ability to represent learned knowledge on the same level as domain data. This contribution aims to weave micro machine learning into data modelling, *i.e.,* to allow to model learning and domain knowledge in the same data models and with the same concepts.

This contribution is based on the work that has been presented in the following papers:

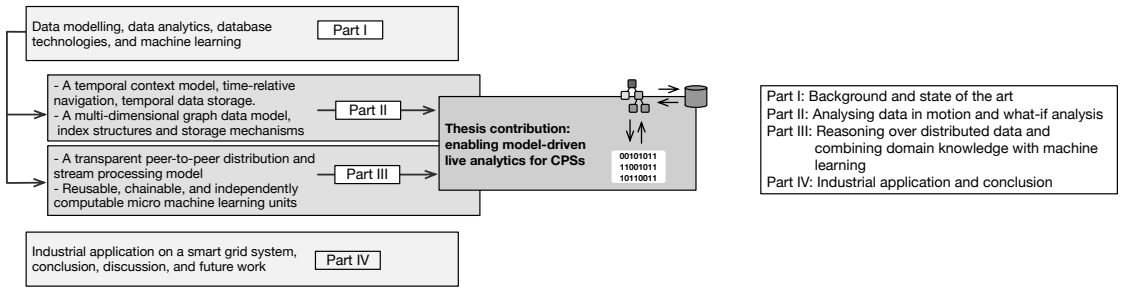


Figure 1.7: Thesis structure

- under submission at International Journal on Software and Systems Modeling (SoSyM): Thomas Hartmann, Assaad Moawad, Francois Fouquet, and Yves Le Traon. The Next Evolution of MDE: A Seamless Integration of Machine Learning into Domain Modeling
- Thomas Hartmann, Assaad Moawad, François Fouquet, Yves Reckinger, Tejeddine Mouelhi, Jacques Klein, and Yves Le Traon. Suspicious electric consumption detection based on multi-profiling using live machine learning. In *2015 IEEE International Conference on Smart Grid Communications, SmartGridComm 2015, Miami, USA, November 2-5, 2015*

1.7 Thesis structure

This thesis encompasses four parts. The first part introduces the technical background of this dissertation and the state of the art. Then, the two main parts of this manuscript propose solutions to address the challenges presented in Section 1.4. Each of the four challenges is addressed in a separate chapter and each of the chapters starts with an introduction to and a motivation for addressing the challenge. In this way, each of these chapters can be read independently. Finally, a fourth part presents a concrete industrial case study and the conclusion of this dissertation. In the following, the structure of the dissertation along with an overview of each part is depicted in Figure 1.7.

Part I: Background and state of the art. In this part the technical background of this thesis is presented in Chapter 2. The background includes data analytics, data modelling techniques, database technologies, and machine learning approaches. The state of the art regarding this dissertation is discussed in Chapter 3.

Part II: Analysing data in motion and what-if analysis. This is the first of two main contribution parts. It defines the core concepts and foundations of this thesis. Chapter 4 introduces a novel approach for representing temporal data together with a time-relative navigation between elements. It defines a data model and storage system for efficiently analysing temporal data. By means of a concrete smart grid load prediction case study, it is shown that this approach can significantly outperform the state of the art. Chapter 5 presents an approach to explore alternative futures,

which inevitably diverge when exploring the impact of different what-if decisions. This chapter extends the temporal data model and storage system with an additional dimension, allowing to explore many different alternatives. The chapter demonstrates that the proposed data model can efficiently fork and update thousands of independent alternatives composed of millions of temporal elements.

Part III: Reasoning over distributed data and combining domain knowledge with machine learning. Part III is the second of two main contribution parts. It builds on top of the data model and storage system described in Part II. First, Chapter 6 introduces an approach for scalable and distributed data models, combining ideas from reactive programming, peer-to-peer distribution, and large-scale modelling. This chapter extends the data model defined in the previous part with a distribution concept and a stream processing model to enable an efficient processing of frequently changing, distributed data. It is shown that these techniques can enable frequently changing, reactive distributed data models that can scale to millions of elements and several thousand computational nodes. Then, Chapter 7 presents an approach to combine machine learning and domain modelling. It suggests decomposing learning into reusable, chainable, and independently computable micro learning units, which are modelled together with and on the same level as domain data. It therefore extends the data model definition presented in the previous chapters with the ability to define fine-grained micro machine learning units. An evaluation based on the smart grid case study illustrates that using micro machine learning for such scenarios can be significantly more accurate than coarse-grained learning while the performance is fast enough to be used for live learning.

Part IV: Industrial application and conclusion. This part presents a concrete industrial application from the smart grid domain and concludes this dissertation. Chapter 8 first describes the case study and how the problems are addressed using model-driven live analytics, before it presents and discusses the results. Finally, this dissertation is concluded in Chapter 9, where possible future research directions are discussed.

Part I

Background and state of the art

2

Background

This chapter presents the technical background for this dissertation, before the state of the art is discussed in the following chapter. It first introduces important terms and techniques for data analytics. Then, the chapter details the modelling background for the present thesis before it introduces background work of database systems. Finally, an overview of machine learning techniques is provided.

Contents

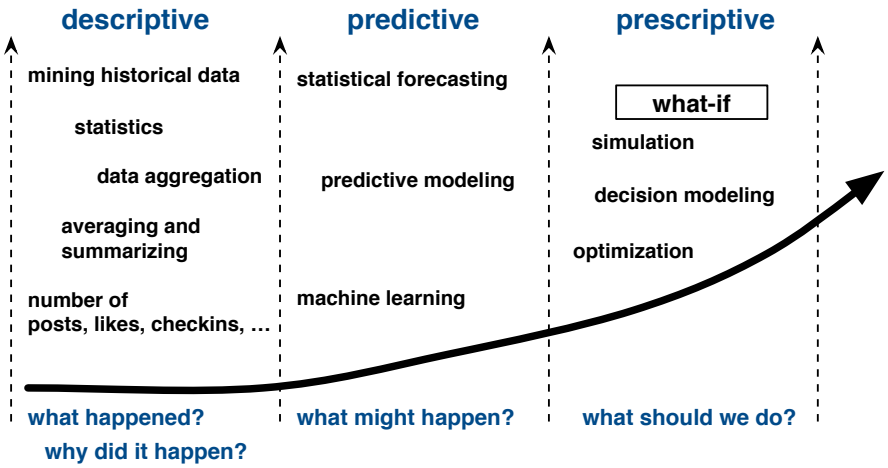


Figure 2.1: Taxonomy of data analytics

2.1 Data analytics

Today, data is generated in very large scale by a wide range of sources, such as sensors, embedded devices, social media, and audio/video. Advances in storage technologies and their continuously falling prices allow to collect and store huge amounts of data for a long time, creating entirely new markets aiming at valorising this data. Recent studies, for example from McKinsey [43], emphasise the tremendous importance of this relatively new field by calling it the *"next frontier for competition"*. Others even compare the value of data for modern businesses with the value of oil, referring to data as *"the new oil"* [53]. However, as it is the case for crude oil, data in its raw form is not very useful. To transform crude oil into value, a long valorisation chain, composed of heterogeneous transformation steps, needs to be applied before oil becomes the essential energy source we all so heavily rely on. Similarly to oil, to turn data into the multi-billion dollar business that some analysts predict it will become [43], we need to process and refine data before we can get valuable insights out of it. This process of turning raw data into valuable insights is referred to as *data analytics*. It has the potential to help us to better understand our businesses, environment, physical phenomena, bodies, health, and nearly every other aspect of our lives. However, turning collected data into competitive advantages remains a big challenge. Analysing data fast enough to support decision-making processes of cyber-physical systems is a main motivation behind this dissertation.

2.1.1 Taxonomy of data analytics

Commonly in literature three main types of data analytics are distinguished [218], [132]. Figure 2.1 summarises these three main types.

The first is **descriptive analytics**. It describes and summarises *"what happened?"*. It also answers the question *"why did it happen?"*. Common techniques for descriptive analytics are data mining, statistics, and other types of post-mortem analysis. For

example, it can provide the number of likes, posts, or check-ins in a social network. Descriptive analytics mainly provide trending information on past or current events.

The next step is **predictive analytics**. It answers the question *"what might happen?"*. It utilises techniques such as statistical forecasting, predictive modelling, and machine learning. Predictive analytics helps to anticipate likely scenarios. Since these methods are probabilistic, it can only forecast what *might* happen based on historical data. An example is producing a credit score of customers to predict the probability of these to pay future credit payments back on time. Another example is sentiment analysis.

The third and most advanced type of analytics is **prescriptive** analytics. It is able to explore multiple potential futures based on the taken actions. Prescriptive analytics is therefore able to answer the question *"what should we do?"* in respect to a certain goal. This goes beyond mere statistical forecasting. Instead, it is necessary to explore what happens if this or that action would be taken, *i.e.,* to explore different alternatives. What-if analysis is an essential but challenging part of prescriptive analytics. The model-driven live analytics approach proposed in this thesis aims at enabling live prescriptive analytics for cyber-physical systems.

2.1.2 Batch and (near) real-time analytics

Traditionally, data analytics, especially for large datasets, is **batch analytics**. Batch data processing is designed to efficiently process high volumes of data (terabytes or even petabytes), where first large datasets are collected over a period of time, then processed, and finally the batch results are produced. Batch analytics process data in a pipeline-based way: first the data to be analysed is extracted from different sources (*e.g.,* databases, social media, or stream emitters), then it is copied into some form of usually immutable data structures, then it is stepwise processed, and finally an output is produced. By parallelising the processing steps, *e.g.,* based on the map-reduce programming model [128], these techniques are able to mine huge amounts of data and can find all kinds of useful correlations. This process is depicted in Figure 2.2. Examples for batch analytics are log analysis, solving of complex optimisation problems, business intelligence, and sorting of huge datasets. Batch processing can take up to hours, or even days. The Apache Hadoop [319] technology stack is an example of a popular batch processing toolset.

In contrary, **(near) real-time analytics** involves a continual data input, process, and output of data. Real-time data analytics requires data to be processed in a short amount of time (up to seconds or minutes). Data is processed as soon (or close to) when data comes into a system. The goal is to enable organisations or systems to take immediate actions. Examples where real-time analytics is needed are IoT, CPSs, or real-time dashboards to reflect business changes throughout the day. Twitter Heron [213] and, to a lesser extent, Apache Spark [327] are examples of real-time analytic frameworks. Most businesses today use batch analytics, in fact, real-time analytics is yet in its infancy.

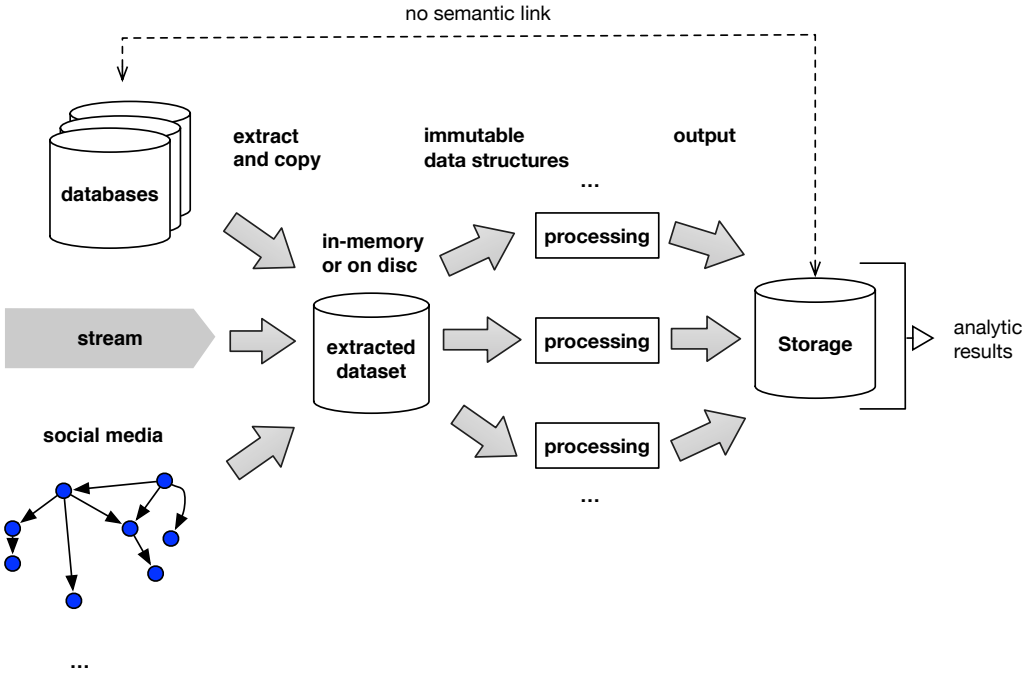


Figure 2.2: Schematic working principle of a typical pipeline-based batch data analytic processes

2.1.3 Complex event processing

Closely related to real-time data analytics is **complex event processing (CEP)** [220]. Basically, the goal of CEP is to identify cause-and-effect relationships among events (in real-time) in streams of data. CEP combines data from multiple sources in order to identify patterns about more complex events. This allows to, instead of querying data for an event, be notified and react in case a complex event happens. Examples where CEP can be applied are social media posts, stock market feeds, traffic reports, and text messages. Operational Intelligence, for instance, combines real-time data processing and CEP to gain valuable insights into operations.

2.1.4 Extract-transform-load and extract-load-transform

Extract-transform-load (ETL) and extract-load-transform (ELT) are two terms often used in today's analytic environments. Therefore, both terms are introduced in the following.

Traditional data warehousing is following an **ETL** process. That is, data is in a first step *extracted*, *i.e.,* copied from source systems to a staging area. In a second step, data is *transformed* and processed. This means, data is reformatted for the warehouse and business calculations are applied. Finally, data is *loaded*. In the loading step data is copied from the staging area into the data warehouse. A key limitation of ETL is that early in the process it must be decided what data is important and should be extracted to the staging area and processed. Only this data will later become available

in the data warehouse. Furthermore, transformed data cannot be modified after. For example, extracted data may no longer exist in the source or staging database. Also, in case any bugs are discovered in the transform phase, or if the transform process should be changed, this cannot easily be achieved in ETL, since the original raw data is not stored. In addition, this multi-layered process delays the availability of data in the warehouse. Nonetheless, ETL is currently the most used approach.

In recent years, the transformation step is more and more pushed towards the data source (database). Since in this approach, the transformation is done after the loading phase, it is called **ELT**. By loading source data directly into the target environments instead of a staging/transformation platform, data becomes available significantly sooner. Data can then directly be processed, *i.e.,* transformed, in the target platform. In ELT, the load process is isolated from the transformation process. This even allows to delay the transform step to query time. ELT makes sense in case the target is a computational powerful computer. Whereas in ETL transformations are processed by ETL tools, in ELT transformations are processed directly by the target datasource.

In short, the main difference between ETL and ELT is where data is processed.

2.1.5 OLAP and OLTP

Online analytical processing (OLAP) is a technique often used for business intelligence, report writing, financial reporting, budgeting, and data mining. It enables users to selectively extract and view data from different points of view. OLAP systems store and manage large amounts of historical data with the goal to analyse this data. The term OLAP was introduced based on the term online transactional processing (OLTP) from traditional databases. OLTP is a class of systems that facilitates transaction-oriented applications. OLTP environments are used to transact and query against data, and support the daily operational needs of the business enterprise [120]. OLAP is discussed in more detail in Chapter 3.

2.2 Modelling

Modelling is a fundamental process in software engineering. The term **model** denotes to an abstraction or simplification of a subject one wants to reason about in some form or another. Models are simplifications in the sense that they only represent a certain part of the subject, *i.e.,* they limit the scope of a subject to the relevant parts, given a certain purpose. A subject can be something from the real world, or something imaginary. Models offer a *simpler, safer and cheaper* [274] means to reason. Benyon [91] defines a model as follows: *"A model is a representation of something, constructed and used for a particular purpose".*

Over time different languages, formalisms, and concepts to model and reason about systems have been developed and used for different purposes [261], [84], [296]. Entity-

relationship models [287], for example, are used as a general modelling concept for describing entities and the relationships between them. They are widely used for defining data models that can be implemented in databases, especially in relational databases. Other examples are ontologies, the resource description framework (RDF) [217], and the web ontology language (OWL) [315], which are particularly used in the domain of the Semantic Web. These allow to describe facts in a *subject-predicate-object* manner and provide means to reason about these facts. A subject is a resource, a predicate denotes aspects of the resource and specifies a relationship between the subject and object. For example, the fact "the table is made of wood" is a triple of a subject (the table), a predicate (is made of), and an object (wood).

Most modelling approaches have in common that they describe a context using a set of concepts (also called: classes, types, elements), attributes (or properties), and the relations between them. We refer to the representation of a context (set of described elements) as a *context model* or simply as *model* and to a single element (concept) as *model element* or simply as *element*.

Closely related to modelling is the concept of **meta modelling**. A meta model is an abstraction of the model itself. It defines the properties of the model. A model conforms to its meta model, comparable to how a program conforms to the grammar of the language it is written in. In this respect, a meta model is a model itself, which makes statements about what can be expressed in valid models. It offers the vocabulary for formulating reasonings on top of a given model. Both, models and meta models are models but have different purposes. A meta model describes a model. The term "meta" is relative in the sense that, depending on the perspective, a model can be a model or a meta model. An instance of a meta meta model is a meta model, the instance of a meta model is a model and so forth.

This thesis builds on the concepts of model-driven engineering and models@run.time, which are detailed in the following.

2.2.1 Model-driven engineering

MDE [195], [101] is a software development methodology to address the increasing domain complexity of today's application domains. Model-driven engineering focuses on the specification of formal models. It promotes to provide abstractions of the problem space that express designs in terms of application domains (*e.g.,* finance, telecom, healthcare) rather than abstractions of the solution space (*i.e.,* the domain of computing technologies). More specifically, MDE addresses platform complexity and expresses domain concepts by combining [281]:

- **Domain-specific modelling languages (DSMLs)** "whose type systems formalise the application structure, behaviour, and requirements within particular domains, such as software-defined radios, avionics mission computing, online financial services, warehouse management, or even the domain of middleware platforms [281]". In contrary to a general-purpose language (GPL), which is broadly applicable across application domains, a DSML is specialised to a par-

ticular application domain. Domain-specific modelling languages can be textual or graphical and are used by developers to build applications in terms of the concepts and relations of a particular domain. They are tailored to match the semantics and syntax of a certain domain. In this respect, a DSML provides a type system for the domain concepts.

- **Transformation engines and generators** "that analyse certain aspects of models and then synthesise various types of artefacts, such as source code, simulation inputs, extensible markup language (XML) deployment descriptions, or alternative model representations [281]". This allows to generate systematic functionalities. Examples are persistence layers, constraint checking, and platform dependent code. Besides saving development time, automated code generation and transformation also ensures what is often referred to as "correct-by-construction", as opposed to conventional "construct-by-correction".

Often claimed advantages of MDE are: abstraction from specific realisation technologies (improved portability, interoperability), automated code generation and transformation (increased productivity, efficiency), model checking and verification (improved quality), separation of application and infrastructure code, reusability, readability, and cost-effectiveness [281], [195] [83]. On the other hand, common critic points of model-driven engineering is the comparatively high initial effort, *e.g.,* for defining DSMLs, scalability issues, poor tool support [195] [83], [320], and organisational challenges to adopt MDE [185]. The two de facto standards for model-driven engineering are model-driven architecture (MDA) promoted by the Object Management Group (OMG) and the Eclipse ecosystem of modelling tools and languages [28]. OMG provides a set of specifications, such as MOF [238] and the Unified Modeling Language (UML) [253], rather than concrete implementations.

Model-driven engineering focuses on creating and exploiting domain models at different levels of abstraction, which are conceptual models of all topics related to a specific problem. Nonetheless, MDE largely applies models to certain parts of the development process. In particular, structural and compositional aspects in the design phase or model checking and verification in the testing phase [281]. The models@run.time paradigm (*cf.* Section 2.2.3) suggests to extend the usage of models from the design phase to the runtime of a system. In this thesis, we apply and extend model-driven engineering and models@run.time techniques to create abstractions of cyber-physical systems, suitable for near real-time analytics of data collected in such systems.

2.2.2 MOF: The Meta Object Facility

Considering the importance and influence the Meta Object Facility (MOF) has on model-driven engineering, this section provides an overview about MOF. MOF is an OMG standard for model-driven engineering. The MOF specification [238] defines its scope as:

"This International Standard provides the basis for metamodel definition in OMG's family of MDA languages and is based on a simplification of UML2's class modeling

capabilities. In addition to providing the means for metamodel definition it adds core capabilities for model management in general, including Identifiers, a simple generic Tag capability and Reflective operations that are defined generically and can be applied regardless of metamodel."

MOF was developed with the intention to offer a type system for entities in Common Object Request Broker Architecture (CORBA) and a set of interfaces to manipulate those types. CORBA is an OMG standard [252] by itself. It has been designed to enable communication between systems deployed on diverse platforms, on different operating systems, programming languages, and computing hardware. Although CORBA itself uses an object-oriented model, systems using CORBA do not have to be object-oriented. It is an example of a distributed object paradigm. MOF enables to define the structure (or abstract syntax) of data or of a language. The primarily purpose of MOF is to provide means to define meta models. In this respect, MOF plays a similar role than Extended Backus–Naur Form (EBNF) plays for defining programming language grammars. In short, MOF is a DSML to define meta models. There are two versions of MOF, Essenntial MOF (EMOF) (a reduced version) and Complete MOF (CMOF). The variant ECore, defined in the Eclipse Modeling Framework (*cf.* Section 2.2.5.1), is strongly based on EMOF.

Despite the fact that various OMG specifications refer to a four layered meta model architecture—shown in Figure 2.3—MOF in fact defines fundamental modelling concepts that can be used to handle any number of layers (**meta levels**) greater than one. Figure 2.3 shows on the left the MOF-layers and on the right side excerpts of the corresponding model and its relationships to the other layers. Due to the widespread use of the four layered meta model architecture in literature, the four layers are presented in the following:

- **M3 or meta meta model level**: represents the language used by MOF to define meta models, called M2-models. MOF can be defined by itself.
- **M2 or meta model level**: this layer represents the UML meta model, *i.e.,* the model that describes UML. The M2-layer is used to define M1-layer models.
- **M1 or model level**: this layer defines user models expressed in UML.
- **M0 or runtime level**: the last layer, called M0-layer, is used to describe real-world objects, *i.e.,* the running system.

The modelling concepts *Classifier* and *Instance* or *Class* and *Object* and the ability to navigate from an instance to its meta object, *i.e.,* its classifier, allows MOF to handle an arbitrary number of layers (at least two). MOF is reflective (can be used to define itself), therefore there is no need to add an additional layer on top of M3. As can be seen in Figure 2.3, what is a model and what a meta model is relative and depends on the viewpoint. If we consider a M1-layer model, a M2-layer model is the meta model of it, but—on the same time—the M1-layer model is a meta model of a M0-layer model.

Models@run.time seek to extend the applicability of model-driven engineering, which is mainly applied at the design time of a system, to the runtime environment. This

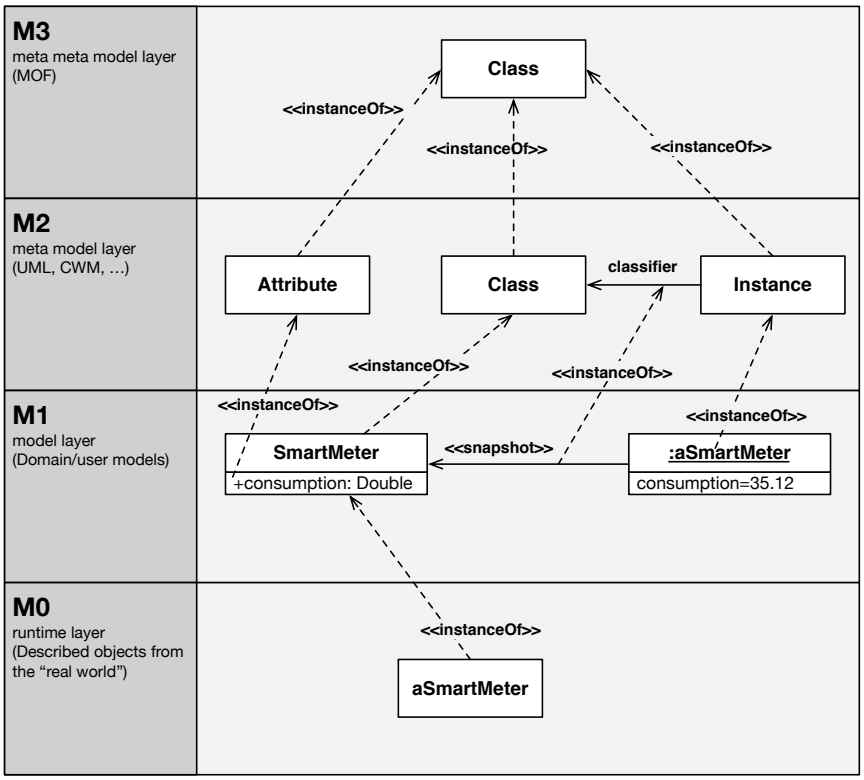


Figure 2.3: The four layered meta model hierarchy of MOF

thesis builds on models@run.time concepts, therefore this paradigm is detailed in the following section.

2.2.3 Models@run.time

Over the past few years, an emerging paradigm called *models@run.time* [96], [246] proposes to use models both at design and runtime in order to support self-adaptive systems. Models@run.time empower self-adaptive systems with a model-based abstraction causally connected to their own current state. The model provides up-to-date information about the system, *i.e.,* it reflects the current state of the system. Since the model is causally connected to the system, adaptions can be made at the model level. Similar to model-driven engineering, models@run.time are models at a high level of abstraction and related to the problem space rather than the solution space. Models@run.time are tied to the models produced as artefacts from the MDE process [96]. Blair *et al.,* [96] define a **model@run.time** as: *"a model@run.time is a causally connected self-representation of the associated system that emphasizes the structure, behavior or goals of the system from a problem space perspective".*

At design time, following the model-driven engineering paradigm, models support the design and implementation of the system. The same (or similar) models are then embedded at runtime in order to support reasoning processes. Models provide a semantically rich way to define a context and can be used in reasoning activities. This

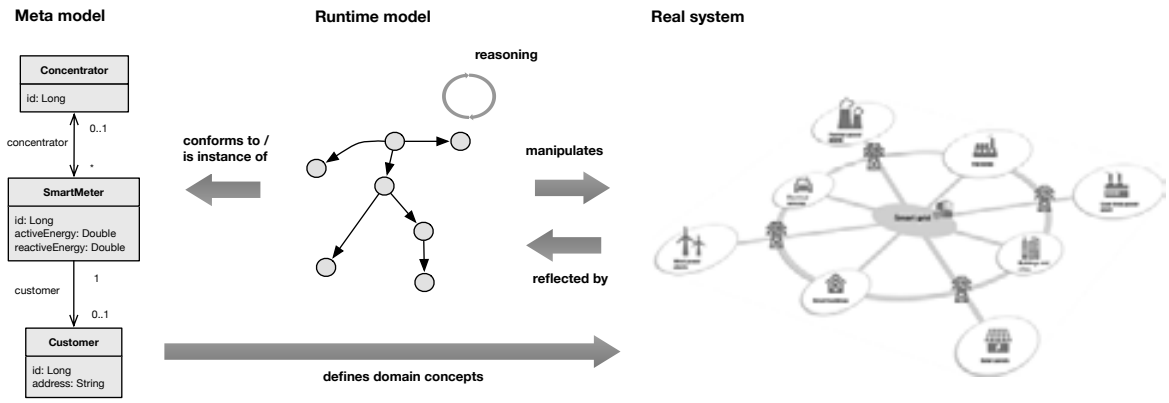


Figure 2.4: Schematic representation of models@run.time

is depicted in Figure 2.4. As can be seen in the figure, a meta model is used to define the domain concepts of the real system, *i.e.,* it specifies the elements which can be used by the model. The actual runtime model is an abstraction of the real system and conforms to (is an instance of) its meta model. Runtime models—as abstractions of the real system—can be used to reason about the state of the real system. Due to the causally link of the model@run.time, it reflects the current state of the real system and vice versa, the real system can be manipulated through the runtime model.

Models@run.time are often mentioned in the context of architectural and variability models [246], [155], [109], where architectural models are used as runtime artefacts to enable architecture-based dynamic adaptation and reconfiguration. For instance, this enables systems to (i) dynamically explore several adaptation options (models) in order to optimise their state, (ii) select the most appropriate one, and (iii) run a set of verifications of viability on new configurations before finally asking for an actual application. As stated by Blair *et al.,* [96], runtime models can also be used "to support dynamic state monitoring and control of systems during execution, or to dynamically observe the runtime behaviour of systems". Following these considerations, in the context of this dissertation, we suggest to use models—as abstractions—of cyber-physical systems (in particular smart grids) during runtime in order to structure and reason about the state of these systems with the goal to support decision-making processes. Therefore, we extend runtime models with a temporal dimension (*cf.* Chapter 4), the ability to explore many different hypothetical actions (*cf.* Chapter 5), to reason about distributed, constantly changing data (*cf.* Chapter 6), and learning (*cf.* Chapter 7). This thesis pursues the idea of model-driven engineering and models@run.time further and brings it to another domain: near real-time data analytics.

2.2.4 Meta models, models, and runtime models in the context of this dissertation

To clarify the used terminology, Figure 2.5 shows the relations between meta models, models, and object graphs in the context of this dissertation. In MDE approaches, we usually first model a domain using a meta model, defined in languages like Eclipse Mod-

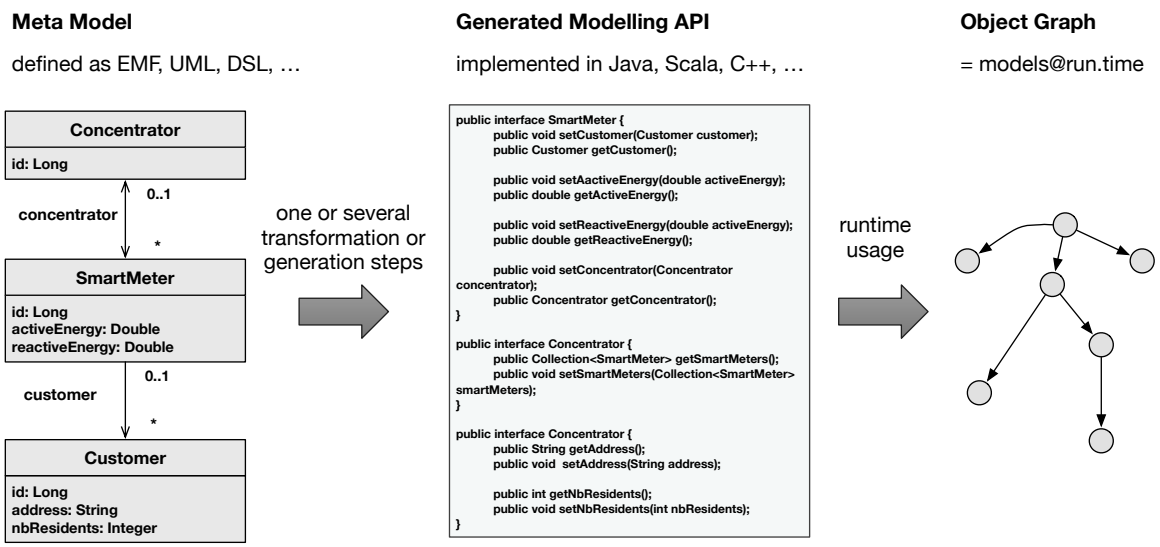


Figure 2.5: Relation between meta models, models, and object graphs in the context of this thesis

eling Framework (EMF), UML, or other graphical or textual domain specific modelling languages. For this purpose, this thesis uses a textual DSML which is conceptually close to EMF but provides a different syntax. The relevant parts of this language, its syntax and semantics, are described throughout this thesis in the context when the language is used. Then, one or several transformation or generation steps transform the meta model into the actual model, usually implemented in an object-oriented programming language, like Java, Scala, or C++. The generator used in this dissertation currently supports Java and JavaScript. This model can then used in the implementation of an application. During system execution, the runtime model can therefore be interpreted as an object graph. In this thesis we use the terms runtime model and object graph synonymously. To refer to a meta model we explicitly use the terms meta model or domain model. We refer to the representation of a context (set of described elements) as a *context model* or simply as *model* and to a single element (concept) as *model element* or simply as *element*.

The object graph (runtime model) is where the main contributions of this thesis are. More specifically, the present dissertation introduces a multi-dimensional graph data model (*cf.* Chapter 5), which allows to represent and analyse the context of complex CPSs in live. In this respect, meta models can be seen as a typing layer on top of the graph data model. Of course, in addition—following default MDE techniques—it allows to add more meta information, which can be used by the object graph during the execution of a system, *i.e.,* at runtime. For example, as presented in Chapter 7 of this thesis, learning rules can be seamlessly defined in the meta model layer and later, during runtime, leveraged by the object graph data model.

2.2.5 Modelling frameworks

MDE frameworks historically emerged from computer-aided software engineering (CASE) tools developed in the 80s. Since then, they evolved into full MDE ecosystems. This section gives an overview about two concrete open source modelling frameworks, 1) the Eclipse Modeling Framework (EMF) and 2) the Kevoree Modeling Framework (KMF). EMF is presented, since it is considered as the de facto standard for modelling frameworks. On the other hand, KMF is discussed because it was specifically developed to support the models@run.time paradigm. For this reason, the concepts presented in this dissertation have been implemented and integrated into KMF.

2.2.5.1 The Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) [294] is a de facto modelling standard and part of the Eclipse ecosystem [29]. On its project website[1] it is described as *"a modelling framework and code generation facility for building tools and other applications based on a structured data model. From a model specification described in XML metadata interchange (XMI), EMF provides tools and runtime support to produce a set of Java classes for the model, along with a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor."* EMF allows different ways to define models, *e.g.,* annotated Java, UML, XML, or graphical/textual modelling tools. The specified models can then be imported into EMF. It provides the foundations for interoperability with other EMF-based applications and tools of the Eclipse ecosystem. EMF consists of three main building blocks:

- **EMF.Core**: The framework core contains the meta model (Ecore) for describing models. The core also includes runtime support for models (change notifications, persistence support with XMI serialisation) and a reflective application programming interface (API) for manipulating EMF objects.
- **EMF.Edit**: The EMF.Edit framework contains reusable classes (*e.g.,* content and label provider classes, a command framework) for building editors for EMF models.
- **EMF.Codgen**: The code generation facility provides support to build a complete editor for EMF models. It also contains a UI to specify generation options and which generators should be invoked. EMF.Codgen supports three level of code generation: model (Java interfaces and implementation classes for the model classes, factory and package implementation class), adapters (implementation classes that adapt the model classes for editing and displaying), and editors (a basic generated editor, which can be further customised).

Ecore is the meta model at the base of EMF. It enables to define domain models. Ecore is its own meta model, *i.e.,* it is defined with itself. Despite some differences, EMOF and Ecore concepts are very similar. EMF can be considered as a "tuned Java

[1] https://eclipse.org/modeling/emf/

implementation" of most of EMOF. Over the years, EMF has been used to implement a large number of tools and evolved into a widely used modelling framework.

However, EMF has been mainly developed with design time models in mind and less with runtime models. This leads to some significant shortcomings and limitations when used in the context of models@run.time. Therefore, with the Kevoree Modeling Framework, an alternative to EMF has been developed, specifically to meet the models@run.time requirements.

2.2.5.2 The Kevoree Modeling Framework

The Kevoree Modeling Framework (KMF) [147], [151] is an alternative to EMF. Like EMF, KMF is a modelling framework and code generation toolset for building object-oriented applications based on structured data models. However, while EMF was primarily designed to support design-time models, KMF is specifically designed to support the models@run.time paradigm and targets runtime models. While the basic concepts remain similar, runtime models—especially runtime models of complex cyber-physical systems—usually have higher requirements regarding memory usage, runtime performance, and thread safety. Therefore, EMF faces some limitations in supporting the models@run.time paradigm, which KMF tries to address [147]. More specifically, Fouquet [147] *et al.,* state the following requirements of models@run.time, which have initially lead to the development of KMF:

- **Reduced memory footprint**: the more efficient a modelling framework (its runtime) is, the more devices are capable of running it. This is especially important for models@run.time, which for example are used in the context of CPSs and must run on comparatively limited devices.
- **Dependencies**: the size of needed dependencies should be fairly small, since all dependencies need to be deployed on every device using models@run.time. The tight coupling and integration of EMF into the Eclipse IDE comes with the price of comparable large dependencies.
- **Thread safety**: models@run.time are often used in distributed and concurrent environments. This can lead to concurrent modifications of runtime models and must be efficiently supported by a models@run.time runtime.
- **Efficient model (un)marshalling and cloning**: For verification or reasoning purposes, runtime models need to be cloned locally on a device in order to reason on an independent representation. In addition, runtime models often need to be communicated to other computational nodes (in distributed systems, like smart grids), thus an efficient models@run.time infrastructure must provide efficient model cloning and (un)marshalling.
- **Connecting models@run.time to classical design tools**: an efficient runtime model infrastructure should provide transparent compatibility with design environments. Therefore, the Kevoree Modeling framework comes with plugins for intelliJ for a seamless integration into an integrated development environment (IDE).

KMF promotes the use of models not only for code generation or architectural management but also during runtime as a central artefact. Therefore, KMF was from the ground up developed with strict performance and memory requirements in mind. It provides its own (textual) modelling language. The semantic of this language is close to Ecore, however it is not compatible. We use KMF to integrate the concepts introduced in this dissertation. Besides the fact that KMF has been designed specifically for models@run.time, we decided to use KMF as base for our approach rather than, for example, EMF for several additional reasons. First, we found that KMF is a more lightweight framework, making fundamental changes on the core, *e.g.,* to implement asynchronous calls for read and write operations, easier. Secondly, KMF is more suitable to process large datasets, *i.e.,* has lower memory requirements [147]. Thirdly, KMF can be easily extended with different storage concepts, *e.g.,* key-value stores. This is especially important for our approach, since the large amount of data, which needs to be analysed in the context of CPSs and IoT, usually does not fit completely in memory. Instead, data must be efficiently loaded from external storage to memory and vice versa. Last but not least, KMF is the main modelling framework inside our research group at the SnT.

2.3 Database systems

Analysing data of CPSs in order to support decision-making processes makes it necessary to not only consider the current context of a system but to also take the history into account. Due to the large amount of data generated by these systems, keeping the full data history in main memory is not practical. Instead, if the context model and its history exceed a certain size, it is necessary to store parts of the data in databases and load it only on demand.

Over the past few years, so-called **NoSQL databases** have been brought into focus in the context of big data and real-time analytics and are challenging the dominance of relational databases. The definition of NoSQL, which stands for "Not Only SQL" or "Not Relational", is not generally recognised [106]. Nonetheless, these databases are usually characterised by addressing at least some of the following points: non-relational, horizontally scalable, simple API, capable of managing a huge amount of data, weaker consistency models than classical SQL databases, schema-free, easy replication support, and easily distributable. NoSQL databases were originally motivated by Web 2.0 applications and the accompanying need to scale to thousands or millions of users performing updates as well as reads. A very interesting characteristic of NoSQL databases from a developers point of view is that they allow to develop applications without having to convert in-memory structures to relational structures (the so-called impedance mismatch). NoSQL databases range in functionality from simple key-value stores, over disturbed hash tables, to highly scalable database systems. Often, four different categories of NoSQL databases are distinguished:

- **Document databases**: store data in key/document manner, where a document is a complex data structure. Documents can usually contain many different key-value, key-array, or key-document pairs. An example for a document database

is mongoDB [45]. Common formats for documents are XML, JavaScript object notation (JSON), binary JSON (BSON), and so on.

- **Graph stores**: used to store complex graph-like structured data, *e.g.,* social network connections. Popular examples of graph stores are Neo4j [47] and Giraph [5].
- **Key-value stores**: store data in form of simple key-value pairs. Examples of key-value stores are Riak KV [56] and Berkeley DB [19].
- **Column stores**: column-family databases store data in form of rows that consist of many columns associated with a row key. A group of columns saved together in one row is called column family. Cassandra [3] is a popular example of a column-family database.

In this dissertation, we leverage key-value stores as a main technology of our persistence solution for the presented multi-dimensional graph data model (*cf.* Chapter 5). The data model itself and its storage concepts are inspired by graph stores. Therefore, we present important concepts of key-value stores and graph stores in Section 2.3.3 and 2.3.4 respectively. Before that, we present an important conjecture which is often discussed in the context of NoSQL databases.

2.3.1 The CAP theorem

Brewer's CAP theorem [158] states that for a distributed system it is impossible to simultaneously provide all three of the following guarantees:

- **Consistency**: all nodes see the same data at the same time.
- **Availability**: every request receives a response about whether it succeeded or failed.
- **Partition tolerance**: the system continues to operate despite arbitrary partitioning due to network failures.

Many modern NoSQL databases allow to be configured by developers to tune these three parameters to their needs.

2.3.2 Consistency models: ACID and BASE

While the prevailing consistency model for most relational databases is the ACID model, many NoSQL databases adopt the BASE [264] consistency model.

ACID stands for:

- **Atomic**: all operations of a transaction either succeed or every operation is rolled back.
- **Consistent**: after a transaction is completed, the state of the database is structurally sound.
- **Isolated**: concurrent access to data is managed by the database, each transaction appears to run sequentially.
- **Durable**: results of a completed transaction are permanent, even in the advent of failures.

This strong consistency model comes with the price that it needs usually sophisticated locking techniques, which are often heavyweight and possibly a bottleneck for applications. The basic idea behind the BASE consistency model is that many applications might not need such a strong consistency model, and/or scalability, performance, and resilience are more important requirements than strong data consistency.

BASE stands for:

- **Basic availability**: the database appears to work most of the time.
- **Soft-state**: stores don't have to be write-consistent, nor do different replicas have to be mutually consistent all the time.
- **Eventual consistency** [314]: stores exhibit consistency at some later point (*e.g.,* lazily at read time). Frey *et al.,* [152] discuss an interesting approach of differentiated eventual consistency.

Which consistency model is more appropriate varies from application to application. As stated in the CAP theorem, it is impossible to simultaneously provide consistency, availability, and partition tolerance at the same time, developers and software architects have to find the best trade-off according the requirements of their respective application.

2.3.3 Key-value stores

Key-value stores are simple associative arrays or dictionaries where records are stored and retrieved using keys. This is depicted in Figure 2.6. Values can either be retrieved using a key, put for a specific key, or deleted from the data store using a key. Keys can be hashed, which enables `put(key,value)`, `get(key)`, and `delete(key)` operations to be very efficient. The value in a key-value store is usually some form of a blob, *i.e.,* the data store just stores the value, without specifying or knowing what is inside. It is the responsibility of the application to interpret and understand what is stored inside the blob value. This makes key-value stores appropriate for various data. There exists a wide variety of implementations, for example, some of them support the ordering of

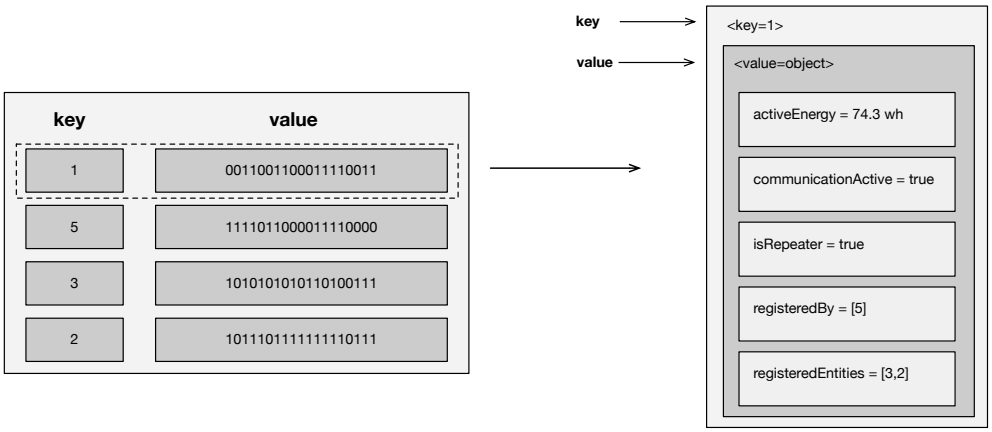


Figure 2.6: A simple key-value example

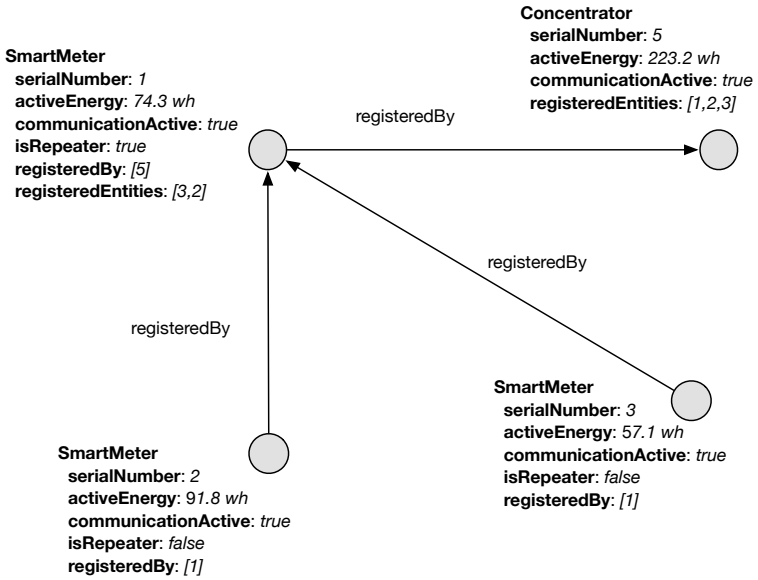


Figure 2.7: A simple graph example

keys. Different implementations support different consistency models, different distribution, and different conflict resolution strategies. Some are plain in-memory stores while others allow to persist data. Key-value stores don't enforce strict schemas and provide very good horizontal scaling characteristics what helped their dissemination in cloud computing and big data technologies. We leverage key-value stores as the base persistence technology behind our approach.

2.3.4 Graph stores

Graphs allow to describe complex domains in terms of rich graph structures. Like mathematical graphs, graphs in database models consist of nodes (entities) and edges (relationships). Entities can be thought of, for example, as an object in an application. Graphs enable it to store entities and relationships between these entities. This maps

closely to many problem domains. Figure 2.7 shows a simple graph example. The figure resembles the example shown in Figure 2.6. In most graph data models nodes can have properties (attributes). Some graph models also allow edges to have attributes. Graph databases embrace the concept of relationships as a core concept in their data models and store them as first class citizens. This allows to query relationships faster than in traditional database models, where relations (especially many-to-many) often need to be expensively computed at query time. Furthermore, the data model of graph databases is closer to the object-oriented model used in many of todays applications, making their integration easier. Graph databases are besides document-oriented databases and key-value stores one major category of NoSQL databases. One of the main ideas behind graph databases is the fact that information in the real world are practically never isolated pieces but rich, interconnected concepts. Neo4j [47] is one of the first and most famous graph data bases.

2.4 Machine learning

Machine learning is an evolution of pattern recognition and computational learning theory in artificial intelligence. It explores the construction and study of algorithms that can learn from and make predictions on data. It uses algorithms operating by building a mathematical model from example inputs to make data-driven predictions or decisions, rather than strictly static program instructions [318]. The essence of machine learning is to create compact mathematical models that represent abstract domain notions of profiles, tastes, correlations, and patterns that: 1) *fit well* the current observations of the domain and 2) are able to *extrapolate well* to new observations [242].

Several categorisations of machine learning techniques are possible. We can divide these techniques according to the nature of the used learning: In *supervised learning* data has predefined and well known fields to serve as expected output of the learning process. While in *unsupervised learning* input data is not labeled and does not have a known field defined as output. Machine learning algorithms try to deduce structures present in the input data to find hidden patterns. In some cases, input data is a mixture of labelled and unlabelled samples. This class of problems is called *semi-supervised learning*. Many machine learning algorithms require some parameters (called *hyper-parameters*) to configure the learning process itself. In some situations, these parameters can also be learned or adapted according to the specific business domain. Thus, they are called *meta learning parameters* and the process of learning such parameters is called **meta learning**. For the rest of the paper we will refer to such parameters simply as *parameters*.

Another categorisation of machine learning techniques is according to the frequency of learning: In *online learning*, for every new observation of input/output, the learning algorithm is executed and its state is updated incrementally with each new observation. This is also known as live, incremental, or on-the-fly machine learning. We speak of *offline learning* or *batch learning* when a whole dataset or several observations are sent in "one shot" to the learning algorithm. We speak of *lazy learning*, or *on-demand learning* when we train a machine learning technique only for the purpose of estimating

the output of a specific input vector. The learning technique is trained using a small batch or a subset of observations similar to the requested input. This type offers a case-based or context-based reasoning because the learning is tailored for the requested input.

Finally, a machine learning module can be composed by combining several machine learning submodules. This is usually called *ensemble methods*. It is often used to create a strong machine learning model from multiple weaker machine learning models that are independently trained. The results of the weaker models can be combined in many ways (voting, averaging, linear combination) to improve the overall learning. Some techniques split the training data over the weaker models, this is called *bagging*. Other techniques split over the features and some split over both data and features. *Random forests* are a powerful example of these techniques, where the global machine learning module is composed by several decision trees, each trained on a subset of data and features. *Neural networks* are another example, where the global network is composed by several neurones, each can be seen as an independent learning unit.

3

State of the art

This chapter discusses work related to the one presented in this dissertation. First, it presents related approaches for analysing data in the context of cyber-physical systems. Then, major data analytics platforms, stream and graph processing frameworks, and graph databases are detailed. Finally, the related work regarding the four challenges addressed in this dissertation is discussed.

Contents

3.1 Analysing data of cyber-physical systems

The specific challenges of data analytics—or big data analytics—in the context of IoT and cyber-physical systems have been discussed and identified as an open issue by several researchers, *e.g.,* [191], [270], [293]. Jara *et al.,* [191] describe existing analytics solutions, discuss open challenges, and provide some guidelines to address these challenges. As a major difference between existing big data analytics and analytics for CPSs, they discuss the need for real-time analytics as a vertical requirement from communication to analytics. They propose a hybrid approach, where real-time analytics is used for control and, in addition, batch processing for modelling and behaviour learning. Ray [270] proposes an architecture for autonomous perception-based decision and control of complex CPS. As two of the main challenges he identifies the complexity of such systems (*e.g.,* stemmed from the complex underlying physical phenomena) and their usually high performance requirements. Ray argues that, therefore, an efficient abstraction is the key for modelling and analysing data of cyber-physical systems. Similar, Stankovic [293] in his discussion about research directions for the Internet of Things, mentions the challenges of converting the continuously collected raw data into usable knowledge, as one of the big challenges.

Thematically close to the context of our work is the work of Šikšnys [291]. Šikšnys focuses on the planning phase in the context of large-scale multi agent CPSs. As a main contribution, Šikšnys provides a definition and a conceptual model (composed of a flexibility model, a prescriptive model, and a decision model) of what he calls "PrescriptiveCPS". Sikšnys contributions remain at the level of a conceptual model, rather than proposing a concrete technical approach for data analytics of CPSs.

In the following we discuss analytics platforms, frameworks, and technologies that are related to our approach, before we present the related work specific for the four challenges addressed in this dissertation. Figure 3.1 presents an overview of the state of the art presented in the following chapters.

3.2 Data analytics platforms

Over the years, data management systems have pushed the limits of data analytics for huge amounts of data further and further.

3.2.1 Online analytical processing (OLAP)

In the 1990's Codd *et al.,* [116] presented a category of database processing, called OLAP, which since then has implemented and widely used in many commercial database systems. It addresses the lack of traditional database processing to consolidate, view, and analyse data according to multiple dimensions, *i.e.,* from multiple perspectives. The term multi-dimensional in the context of OLAP is used in a very

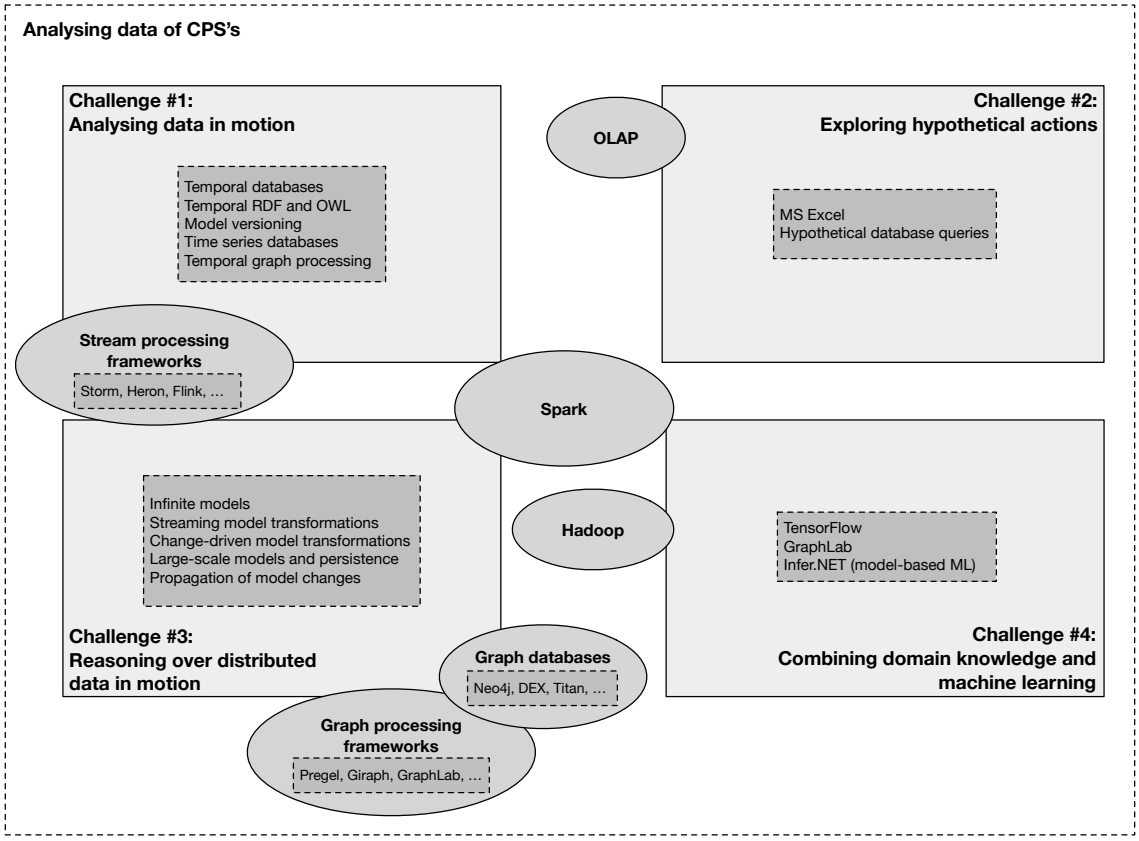


Figure 3.1: Overview of the state of the art related to this thesis

different way than in the context of this dissertation. Whereas in this thesis we refer to "time" and "alternative worlds" as different dimensions of data, a dimension in OLAP environments means a different perspective of the same data, *i.e.,* highlighting different aspects of the same data for different purposes or for different stake holders. OLAP offers three basic analytical operators: *roll-up*, *drill-down*, and *slicing/dicing*. Roll-up is used to consolidate data, this includes for example aggregation of data. In contrary, drill-down is used to enable users to view the details. Slicing and dicing allows users to take out (slice) specific datasets and view (dice) these slices from different perspectives. In OLAP contexts, these perspectives are called *dimensions*. Typical examples of OLAP applications are business reporting, management reporting, marketing, and business process management. OLAP enables complex analytical (ad hoc) queries with fast execution times. The core of OLAP systems are so-called **OLAP cubes** (sometimes referred to as hypercubes). An OLAP cube is basically a multi-dimensional array of data. OLAP cubes are usually filled and prepared in batch mode from different sources. Therefore, OLAP techniques are not suitable for live analytics of complex and frequently changing data of cyber-physical systems. Moreover, OLAP is usually performed on powerful and centralised servers.

In a more recent work about best practices for big data analytics, Cohen *et al.,* [117] present what they call "MAD Skills". They highlight the practice of magnetic, agile, and deep (MAD) data analysis as a departure from traditional data warehousing and business intelligence. This paper mainly discusses suggestions to better address the massive-scale of data, today's analytics has to face. More specifically, they highlight

the need to make analytics more "magnetic", "agile", and "deep". With magnetic they refer to the idea that analytics, to keep pace with ubiquitous data sources, should incorporate many different data sources, regardless of data quality issues. This is opposed to traditional approaches, where data is carefully cleansed before integrated. Agile means to easily ingest, digest, produce, and adapt data at a rapid pace. Finally, deep refers to the ability to integrate sophisticated statistical methods that go well beyond the roll-ups and drill-downs of traditional business intelligence. The ideas presented in this work originated mainly from their development of an analytics platform for Fox Interactive Media, using a large installation of the Greenplum database system. MAD skills provide a new perspective on data analytics and offer many interesting suggestions to tackle the increasing amount of data in the domain of analytics. However, MAD skills are mainly (organisational) recommendations and best practices how to use already existing technologies and methods rather than providing new methods. This means that, like traditional data analytics (think of OLAP), MAD skills are mostly suitable for centralised batch analytics and business intelligence in general.

3.2.2 The Hadoop stack

A very visible approach in the context of big data analytics is the open source Apache Hadoop framework [7]. The Hadoop stack is a framework for distributed storage and distributed processing of very large datasets across computer clusters using simple programming models. It is built to scale from single commodity computers to clusters of thousands of machines. Hadoop consists of four main components:

- **Hadoop Distributed File System (HDFS)**: The HDFS [290] is designed for storing large datasets and to stream those to applications at a high bandwidth. This allows to distribute storage and computation across servers.
- **Hadoop YARN**: YARN [310] is a job scheduling and cluster resource management framework. The main idea of YARN is to decouple the programming model from the resource management infrastructure.
- **Hadoop MapReduce**: A YARN-based implementation of the map-reduce programming model [128].
- **Hadoop Common**: The common utilities supporting the Hadoop stack.

Hadoop was inspired by the Google File System [157] and map-reduce publications [128]. Since then, it has been successfully used in a wide range of projects across many companies, *e.g.,* Yahoo!, Facebook, and LinkedIn. Nonetheless, writing map-reduce jobs is very low level and often leads to code which is hard to maintain and reuse. Therefore, with **Hive** [305] Facebook created an open source data warehousing solution on top of Hadoop, which allows to express queries in a SQL-like declarative language, which are compiled to map-reduce jobs. This language offers a type system with table support containing primitive types, collections (*e.g.,* arrays, maps), and nested compositions. Like Hive, **Pig** [13] enables to execute ad-hoc queries on historical data using a higher level—compared to implementing map-reduce jobs—query

language. With **HBase** [10] an open source column store on top of HDFS has been developed based on the concepts of Google's BigTable to provide the Hadoop stack with a non-relational, distributed database.

The Hadoop stack was designed and optimised to efficiently process massive datasets of structured, semi-structured, and unstructured data—but it was essentially designed as a batch processing system. Despite some recent attempts to add support for interactive SQL queries against Hadoop data, like **Impala** [11] and **Hawq** [9], Hadoop is nonetheless fundamentally a batch operation environment. The fact that streamed data is first stored on disk and then analysed through map-reduce jobs comes with performance limitations, which makes the Hadoop stack less appropriate for real-time analytics and is therefore not well suited for live analytics of CPS data. Moreover, Hadoop doesn't provide native support for any of the four challenges identified in this dissertation: analysing data in motion, exploring hypothetical actions, reasoning over distributed data in motion (though it supports distributed data at rest), combining domain knowledge and machine learning.

3.2.3 The Spark stack

Besides Hadoop, Spark [14] is probably one of the most prominent big data analytic frameworks today. Spark is an open source cluster computing framework. It was originally developed at the University of California at Berkeley's AMPLab. In the meantime, Spark has become an Apache top-level project. It offers an interface for programming clusters with implicit data parallelism and fault-tolerance. Spark makes intensive use of an execution engine that supports cyclic data flow and in-memory computation. Spark developers claim that programs run much faster than Hadoop, especially for in-memory computations [14].

The core of the Spark processing model is a data structure, called **resilient distributed dataset (RDD)** [326]. RDDs are distributed memory abstractions that enable programmers to perform in-memory computations on large clusters in a fault-tolerant way. In contrary to other approaches, like Hadoop, where results can only be reused between several processing iterations (or queries) by writing and reading them to disk, RDDs allow to store results in memory and reuse them across several iterations. Two types of applications motivated the development of RDDs: iterative algorithms and interactive data mining tools. For these cases, keeping data in memory can significantly improve the performance [326]. A RDD can be thought of as an immutable multiset of data items, which are distributed over a cluster of machines. RDDs were developed to address the limitations of the Hadoop map-reduce paradigm, which forces a linear data flow on distributed programs.

Spark SQL [80] is a component of the Spark stack, which integrates relational processing with Spark's functional API. This allows programmers to leverage the benefits of relational processing while working with the Spark stack. A declarative *DataFrame* API seamlessly integrates with procedural Spark code and allows to cross-fertilise relational and procedural processing. Spark SQL enables developers to express complex analytics in form of a rich API.

Spark itself provides only the computing core and data structures. To be used as a full data analytics framework it requires a cluster manager and a distributed storage system. As cluster managers, besides supporting standalone, *i.e.,* native Spark clusters, Spark supports, among others, Hadoop YARN and Apache Mesos [12]. It supports a wide variety of distributed storage systems, *e.g.,* HDFS, Cassandra, Amazon S3, Tachyon [223] (renamed into Alluxio), and many more. The Spark core itself can be seen as an alternative and faster processing model to replace Hadoop's computation model. However, by itself, Spark is, like Hadoop, essentially a batch processing framework. In order to address the increasing need for (near) real-time analytics, Spark comes with a streaming component, called Spark Streaming.

The core concept of **Spark Streaming** is to divide arriving data into mini-batches (short, stateless, deterministic tasks) of data and then to perform RDD transformations and computations on those mini-batches. A big advantage of this is that it allows to write streaming jobs the exact same way than batch jobs. However, on the downside this comes with the limitation that the latency is equal to the mini-batch duration. Spark Streaming is built on so-called *"discretised streams"* (D-Streams), which are proposed by Zaharia *et al.,* [328] as a processing model for fault-tolerant streaming computation at a large-scale. D-Streams aim at enabling a parallel recovery mechanism that should improve efficiency. While Spark's batch-based streaming mechanism works very well for many domains, like business intelligence or social media analytics, for applications that need to (potentially) react to every data record this is problematic. For example, in the context of cyber-physical systems, it is often not appropriate to wait the end of a batch period before reacting. Also, in batch-based processing systems it is often difficult to deal with poor data quality. For example, records can be missing, data streams can arrive out of (creation) time order, and data from different sources may be generated at the same time, but some streams may arrive delayed. These real-time factors cannot be addressed easily in a batch-based processing. This is usually much easier to overcome in a record-by-record stream processing system, where each record has its own timestamp, and is processed individually. The same counts for finding correlations between multiple records over time. Batch-based stream processing is limiting the granularity of a response inherently to the length of a batch.

The **GraphX** [323] component of Spark is especially interesting in the context of this dissertation. Many problem domains, from social networks over advertising to machine learning, can be naturally expressed with graph data structures. However, it is difficult and often inefficient to directly apply existing data-parallel computation frameworks to graphs. To address this, GraphX allows to express graph computations, that can model the Pregel [232] abstraction, on top of Spark. Like Spark itself, GraphX was originally developed at the University of California, at Berkeley's AMPLab. It maps distributed graphs as tabular data structures to fit the graph representations into the Spark processing model. GraphX aims at combining the advantages of data-parallel and graph-parallel systems by allowing to formulate graph computations within the Spark data-parallel framework [323]. With *GraphFrames* [126] the authors present an integrated API for mixing graph and relational queries on top of Spark. It enables users to combine graph algorithms, pattern matching, and relational queries. While representing complex data in form of graphs goes into a similar direction than the approach followed in this thesis, the graph data structure provided in GraphX does

not provide means to represent and analyse data in motion, nor does it support to represent and analyse many different alternatives.

With **MLlib** [237] the Spark stack provides a distributed machine learning library on top of Spark's core. It supports several languages and, therefore, provides a high-level API to simplify the development of machine learning tasks. A wide variety of machine learning and statistical algorithms have been implemented and integrated into MLlib, such as support vector machines, logistic regression, linear regression, decision trees, alternating least squares, k-means, and many more. MLlib greatly benefits from the integration into the Spark ecosystem and can leverage its components, like GraphX and Spark SQL. While MLlib is able to extract commonalities over massive datasets in a very efficient way, it is mostly a coarse-grained and pipeline-based learning approach. This makes it less appropriate for online learning and for structuring fine-grained learning units of entities which behave very differently, as it is needed for CPSs. Although, by leveraging Spark's host languages, learning tasks can be implemented on a higher level of abstraction, implementing learning tasks with MLlib remains challenging. In this dissertation we propose an approach to model machine learning together with the domain data itself, instead of managing learned and collected data in different models.

Velox [121] is an online model management, maintenance, and serving platform for the Spark analytics stack, which aims at serving and managing models at scale. A model in the context of Velox refers to learning models,*i.e.,* models built from training datasets in order to enable predictions (not to models in the sense of MDE). While commodity cluster compute frameworks like Hadoop and Spark enabled to facilitate the creation of statistical models that can be used to make predictions in a variety of applications, *e.g.,* personalised recommendations or targeted advertising, they mostly neglect the problems of how models are trained, deployed, served, and managed. A common approach is to store the computed models into a data store that has no knowledge about the semantics of the model [121]. This let it to other application layers to interpret and serve the models and to manage the machine learning life cycles, *e.g.,* to integrate feedback into the models. Velox aims at filling this gap of model serving and management. Therefore, it performs two main tasks. First, Velox exposes models as a service via a generic API which provides predictions for a range of query types. Secondly, it manages the model life cycle, *i.e.,* it keeps the model up-to-date by implementing offline and online maintenance strategies. Instead of providing an additional modelling layer for model management and maintenance, the approach presented in this thesis suggests to integrate model management and maintenance directly into data and domain modelling. More specifically, by expressing the relationships and dependencies of learned information together with and on the same level as collected data—combined with online learning techniques—the model is always up-to-date. Moreover, this model is used to generate a domain-specific API (through MDE techniques), which enable applications to leverage the information and semantics captured in the model.

Essentially, Spark is like Hadoop a pipeline-based analytics framework. Although, it allows to efficiently process mini-batches in a stream-based way with Spark Streaming, the core of Spark is pipeline-based. While this is appropriate for many application domains, for others, like cyber-physical systems and IoT, which often require immediate reactions, such pipeline-based approach is problematic. This is continued in

machine learning libraries, like MLlib, which are built on top of Spark's core and are therefore also mainly designed for a pipeline-based processing model. This, essentially, makes Spark, like Hadoop, less appropriate for near real-time analytics. While GraphX offers rich, graph-based data structures in Spark, it does not provide support for time-evolving graphs nor for representing many different alternatives. Both, Spark and Hadoop enable distributed computing and are able to process data close to the source. However, although both analytics stacks are scalable, they require a considerable amount of computation power and are not well suited for smaller devices. While Velox and MLlib offers higher-level abstractions for learning, it does not allow to model data and learning at the same level. Whereas data analytics in the context of cyber-physical systems requires to analyse data in live, the amount of data is usually less compared to business intelligence or social media analytics, where Spark and Hadoop are optimised for. Although, the amount of data to be analysed in the context of this dissertation is "big" it is not in the dimensions of petabytes, as it can be the case for Spark. In summary, while there are similarities of our approach compared to Spark and Hadoop, the specific requirements of live analytics of CPS data and, therefore, the resulting techniques are quite different.

3.3 Stream processing frameworks

This section discusses real-time stream data processing systems. In contrary to Hadoop and Spark, these have been designed from the beginning as real-time data analytics frameworks.

Storm [308] is a real-time fault-tolerant and distributed stream data processing system developed at Twitter and released as an open source Apache project. It is one of the early open source and popular stream processing systems. Storm is mainly written in Clojure. It has been designed for use cases like real-time analytics, online machine learning, continuous computation, and ETL. Storm aims to become for real-time analytics what Hadoop is for batch analytics. The basic data processing architecture of Storm uses *streams of tuples* flowing through *topologies* [308]. In the context of Storm, a topology is essentially a graph where nodes are computations and edges represent the data flow. Two types of nodes are distinguished, *spouts* and *bolts*. Spouts are tuple sources for the topology and bolts are processing the tuples. Spouts are typically pulling data from queues like Kafka [209]. Topologies in Storm can have cycles. Storm itself runs distributed clusters, *e.g.,* Mesos. In Storm terminology a *Nimbus* is a master node. Clients send their topologies to such master nodes. The master node then handles the distribution and coordinates the execution of the topology. The execution is done on worker nodes, where each can run an arbitrary number of worker processes. At any time a worker process is mapped to a single topology. On the other hand, a single machine can have several worker processes. A worker process runs a JVM, which in turn can run one or more executors. An executor consists of one or several tasks. The actual work for bolds or spouts is done in tasks. Through YARN, Storm can be integrated into the Hadoop stack. Storm supports two data guarantee semantics, at-least-once as well as at-most-once processing. Data can be persisted in various key-value stores. **Trident** [66] is an extension and alternative interface to Storm. It

provides exactly-once processing, transaction-like persistence (with various key-value stores), and a set of analytical operations on top of streams. Like Storm, Trident was developed at Twitter.

Heron [213] is the successor of Storm. Like Storm, it has been developed at Twitter. Heron aims at addressing some of Storm's limitations (scalability, debug-ability, manageability, and efficient sharing of cluster resources) for processing the continuously increasing amount of data at Twitter. Heron is API-compatible with Storm. One of the issues addressed in Heron is the fairly complex design of Storm's workers by replacing it by a simpler model. A second change in Heron concerns the Nimbus, which is functionally overloaded in Storm and can therefore lead to a significant overhead [213]. Instead, Heron uses existing open source schedulers, *e.g.,* YARN and Aurora [329] and adds a new component, a *topology master*. The topology master manages a topology throughout its existence. Heron also introduces stream managers, which handle the routing of tuples efficiently. Unlike Storm's workers, Heron uses so-called "Heron Instances" where each one is a JVM process, which runs only a single task of a spout or bold. These—and other smaller changes and improvements—help to address some of the limitations of Storm. Heron, like Storm, supports both, at-least-once and at-most-once processing. Data can be persisted in various key-value stores.

Flink [4] is another open source framework for distributed big data analytics. It is an Apache project. Like Storm and Heron, Flink has been designed for real-time analytics. Flink supports exactly-once data processing. The core of Flink is a distributed streaming data flow engine, which provides data distribution, communication, and fault tolerance for distributed computations over data streams. Streams in Flink consist of key-value tuples. Flink enables to write programs in Java or Scala which are automatically compiled into Flink data flow programs. These can then be executed in a cluster or cloud environment. Flink itself doesn't provide a data storage system, instead it uses existing distributed storage systems like HDFS or HBase. Flink takes data input from message queues like Kafka. It includes several modules and APIs to create applications with:

- **DataStream API**: The core stream processing API of Flink. DataStream programs are implementing transformations (*e.g.,* filtering, updating, aggregating) on data streams. Message queues or socket streams are used as data sources. Results are delivered using *sinks*, which for example write the results to files, sockets, or external systems.
- **DataSet API**: An API for batch analytics. DataSet programs are implementing transformations (*e.g.,* filtering, mapping, joining, grouping) on data sets. Data sources are, for example, files or collections. Like for data streams, results are returned using sinks.
- **Table API**: A SQL-like querying language, embedded in Java and Scala. Operates on a relational table abstraction, which can either be created from external data sources, or existing DataSet and DataStream abstractions. Flink's Table API offers relational operators like selection, aggregation and table joins.

In addition to this, Flink comes with a number of libraries:

- **CEP**: A library to detect complex event patterns in data streams. Events can then be constructed from matching sequences.
- **Machine Learning library**: FlinkML is Flink's machine learning library. It supports a growing number of algorithms, such as support vector machines, multiple linear regression, k-Nearest neighbours join, alternating least squares. It is planned to extend FlinkML to streaming, *i.e.,* to enable online learning, but at the time this dissertation was written FlinkML is limited to batch learning.
- **Gelly**: Gelly is a graph API for Flink. It provides utilities to simplify graph analysis applications in Flink. Graphs can be transformed and modified with high-level functions, similar to how Flink supports batch processing. The graph API can only be used for batch processing.

Flink provides APIs for processing distributed batch data and distributed streams of data. It is an alternative to Spark and Hadoop but also to Storm and Heron. While Flink provides many interesting features, machine learning and graph processing is currently limited to batch processing. The streaming API is limited to process n-tuples of data. More complex data structures, like graphs, are not supported by the streaming API. Furthermore, it doesn't provide support for time evolving data nor exploring many different alternatives in parallel.

S4—Simple Scalable Streaming System [250]—is another distributed, partially fault-tolerant platform for processing streams of data. It has been originally developed at Yahoo! and has become, in the meantime, an Apache project. The architecture of S4 is close to the Actors model [71] and inspired by the map-reduce approach. It has been designed as a general purpose distributed stream computing platform. Computations in S4 are performed by so-called *Processing Elements* (PEs) and messages between PEs are transmitted in form of *data events*. PEs do not share any state, event emission and consumption are the only way of interaction. A stream in S4 is defined as a sequence of elements (events) of form (K, A) where K and A are the tuple-valued keys and attributes. The S4 platform consumes such streams, computes intermediate values, and optionally emits streams. A PE instance consists of four components: 1) its functionality implemented in a class, 2) the types of events that it can consume, 3) the keyed attribute in those events, 4) the value of the keyed attributes. *Processing Nodes* (PNs) serve as logical hosts to PEs. The responsibility of a PN is to listen to events, execute operations on incoming events, dispatch events, and to emit output. S4's communication layer uses ZooKeeper [16] to coordinate between nodes in a cluster. No explicit data guarantees are made and data is not persisted to disk.

Other than Flink—which provides a full stack of analytics libraries—S4 focuses on distributed stream processing. Like Flink, S4's stream processing API is limited to data tuples, *i.e.,* it doesn't support more complex data structures (like graphs). Moreover, it doesn't support analysing temporal data nor does it support exploring hypothetical actions. Last but not least, S4 doesn't provide any machine learning library.

MillWheel [72], developed at Google, is a framework for building distributed low-latency data processing applications. In MillWheel, users specify directed computation graphs and the application code which runs on a node. Arbitrary topologies

can be defined for the computation graph. The system handles persistent state of the continuous flow of data records (via BigTable [111]) and manages fault-tolerance guarantees. Data records are continuously delivered along the edges of the graph. Its programming model provides a notion of time. This makes it possible to write time-based aggregations. MillWheel places great emphasis on fault-tolerance. It guarantees that every data record in the system is delivered to its consumers and handles the processing of records in an idempotent manner. MillWheel uses checkpoints to persist its progress at fine granularity. Viewed from a high level MillWheel is a graph of user-defined transformations (called *computations*) on input data that produces output data. Computations can be parallelised across several machines. Input and output streams are defined as 3-tuples of (key, value, timestamp). The key is a *metadata* field with semantic meaning, the value is an arbitrary byte string, and timestamps are defined as values (usually the time an event occurred). One of the main motivations behind the development of MillWheel was that many other stream processing frameworks do not implement an exactly-once processing idiom (the Trident extension of Storm provides this meanwhile, as well as Apache Flink) or don't provide sufficient fault-tolerant persistent state support for Google's use cases.

Samza [59] is another distributed stream processing framework. It as an Apache project and provides the at-least-once data processing guarantee. Streams in Samza consist of key-value tuples. Samza uses Kafka as messaging service and YARN as a resource manager to provide fault-tolerance. Besides fault-tolerance, durability (messages are processed in the order they are written and no message is lost), processor isolation, and scalability, Samza provides managed states. This means that Samza manages snapshotting and restoration of a stream processor's state. In case a stream processor is restarted, its state is restored from a snapshot. Samza streams are ordered, partitioned, "replayable", and fault tolerant. A stream consists of immutable messages of a similar type. Data can be persisted with RocksDB [57]. The computing unit performing a logical transformation is called *job* in Samza. For scalability reasons, streams and jobs are chopped into smaller units: *partitions* and *tasks*. A stream consists of one or several partitions. Each partition is a totally ordered sequence of messages. Tasks are used to scale jobs. They are the unit of parallelism of the job—just as the partition is for the stream. A task consumes data from one partition for each of the job's input stream. The number of input partitions determine the number of tasks. A data flow graph can be built by composing multiple jobs. The edges are then streams containing the actual data and the nodes are here for transformations.

Stream Processing Core (SPC) [76] is a distributed stream processing middleware developed by IBM. SPC has been developed to support stream-mining applications using a subscription like model for specifying stream connections. Moreover, it provides both relational and user-defined operators. SPC is built to support applications which are composed from user-developed *processing elements*. These processing elements take data streams from sources as input, filter, and mine them. Continuous inquires, which are expressed as processing flow graphs are evaluated over streams of data. A subscription-like model is used to specify for flow-graphs which streams should be processed. It allows stream mining across raw data streams and data streams generated by other applications. SPC is designed to address the following points:

Table 3.1: Summary and comparison of important stream processing frameworks

	distributed processing	comp. model	source available	guarantees	storage	data model
Storm	✓	stream	✓	at-least-once at-most-once	✓ (k/v stores)	k/v tuple
Trident	✓	stream	✓	exactly-once at-most-once	✓ (k/v stores)	k/v tuple
Heron	✓	stream	✓	at-least-once at-most-once	✓ (k/v stores)	k/v tuple
Flink	✓	stream batch	✓	exactly-once	✓ *e.g.,* HBase,...	k/v tuple graph (only batch)
S4	✓	stream	✓	no explicit guarantees	✗	k/v tuple
MillWheel	✓	stream	✗	exactly-once	✓ BigTable	3-tuple (k,v, timestamp)
Samza	✓	stream	✓	at-least-once	✓ RocksDB	k/v tuple
SPC	✓	stream	✗	no explicit guarantees	no details provided	complex object

- data to be processed is streamed by default
- applications are not restricted to relational operators
- applications are inherently distributed
- support for discovery of new data streams
- processing elements of different applications need to share data to annotate discovered data in streams
- the volume of incoming data is very large, thus the system needs to be scalable

In SPC, high-level user requests are translated into processing flow graphs, referred to as *jobs* or *applications*. Jobs consist of one or several processing elements, which are the fundamental building blocks of an application written with SPC. Processing elements can communicate via *ports*, and can read, process, and write streams of data. They read streams from *input ports* and write streams to *output ports*. Each port defines a *format*, which defines the structure of the elements of a stream. Stream elements are referred to as *stream data objects*. Stream data ojects are typed using a global type system. SPC is designed to run on large-scale clusters. No explicit guarantees for data processing are made.

Table 3.1 provides a summary and comparison of the discussed stream processing frameworks. The table shows for each if it supports distributed processing, the computation model (stream or batch), if the source code is available, data guarantees, if it supports persistent storage, and the used data model.

In contrary to Hadoop and Spark, stream processing frameworks are specifically developed to address the challenge of near real-time data analytics and are therefore especially interesting for this thesis. For example, stream processing presents a good basis for enabling to reason about distributed and frequently changing data (*cf.* Chapter 6). While frameworks like Flink influenced the design and implementation of our approach, they lack major features needed for analysing data of complex cyber-physical systems. First of all, the data model underlying these approaches is rather simple (mostly key-value pairs), which makes it difficult to model the often complex data and relationships of CPSs. Moreover, this makes it difficult to apply an appropriate handling of time for complex structured data and/or the exploration of different hypothetical actions. Secondly, complex analytics and machine learning algorithms often rely on complex data representations, like graphs, to work efficiently. This is hard to model and represent with stream processing frameworks alone, *i.e.,* it is not natively supported. As a consequence, it is also difficult to integrate machine learning techniques into the live analysis. Overall, existing stream processing frameworks are very suitable for real-time analytics of systems like social networks, where information like the number of likes, friends, etc. need to be counted and aggregated. However, for live analytics of complex CPS data, they are insufficient.

3.4 Graph processing frameworks

Recently, much work in the area of data analytics focuses on large-scale graph representation, storage, and processing for analytics. Graphs are suitable to represent complex data and relationships [47], [288], [232] and are therefore especially interesting for this thesis in order to represent the context of complex CPSs. These approaches have significantly influenced the design and implementation of our work.

Pregel [232] from Google is a well-known system for large-scale graph processing. In Pregel, programs are expressed as sequences of iterations, which are called *supersteps*. A vertex can receive messages (sent in the previous iteration), send messages (to other vertices), modify its state and the state of its outgoing edges, and mutate the topology of the graph. The authors of [232] refer to this as a "*vertex-centric approach*". Pregel supports distribution but hides details—like fault-tolerance—behind an abstract API. The Pregel framework calls for each vertex a user-defined function (conceptually in parallel) during the execution of the superstep. A function implements a specific behaviour for a single vertex and a single superstep. It can read messages (sent by other vertices in the previous superstep), send messages to other vertices (which will receive the message in the next superstep), and modify the state of the vertex and its outgoing edges. Somewhat similar to map-reduce, this allows to focus on local actions, processing them independently, while the composition is handled by the system. The goal of this synchronous design is to make it easier to reason about program semantics when implementing algorithms and, at the same time, to ensure that Pregel programs are free of deadlocks. Assuming enough parallel machines, the performance of Pregel programs should be nonetheless comparable with asynchronous systems [232]. A Pregel computation takes an initialised graph as input, performs a sequence of supersteps separated by global synchronisation points, and yields an output. The vertices within a

superstep compute in parallel. Pregel's computation model is inspired by Valiant's bulk synchronous parallel (BSP) model [309]. Edges in Pregel are not first-class citizens, meaning they have no associated computations. Algorithms terminate in the Pregel computation model if all vertices are simultaneously inactive and no messages are in transit. All vertices start as active and set themselves inactive if they have finished their associate computations. A vertex is reactivated if it receives a message. Otherwise, inactive vertices are no longer called in subsequent supersteps. The set of values explicitly output by the vertices are the result of a Pregel program. In short, Pregel implements a pure message passing model, without the possibility of remote reads or any other kind of shared memory. For performance reasons, in a distributed scenario, messages are delivered asynchronously in batches. It keeps vertices and edges on the machine that performs the computation and uses network transfers only for messages. Fault-tolerance is implemented by checkpointing. Pregel uses a master/workers model. Pregel does not provide graph persistence but expects the graph to be completely in memory while processing.

Giraph [5] is an Apache project and inspired by Pregel. Like Pregel, Giraph implements a BSP model. Compared to Pregel, it adds several additional features, such as master computation, sharded aggregators, edge-oriented input, and out-of-core computation. The graph model is similar to Pregel, consisting of vertices and edges, where each can store a value. Messages can be sent between vertices. The computation model is also similar to Pregel, consisting of BSP-like sequences of iterations (supersteps). Giraph allows to implement graph processing jobs based on Hadoop's map-reduce implementation. Giraph is sometimes considered as the open source implementation of Google's Pregel. It allows graph structure changes. Facebook [113] made a number of usability, performance, and scalability improvements to Giraph in order to use it on very large-scale graphs (up to one trillion edges). Han and Daudjee [170] propose with **GiraphUC** a barrier-less asynchronous parallel (BAP) computation model, which aims to reduce both message staleness and global synchronisation. Therefore, they introduce local barriers (per worker) and logical supersteps. Logical supersteps are not globally coordinated, which allows different workers to execute a different number of logical supersteps. Global supersteps group a number of logical supersteps, which are separated by global barriers. They implement their BAP approach into Giraph, while preserving its BSP developer interface. Like Giraph itself, GiraphUC allows graph topology changes. With **Giraph++** [306] IBM proposes what they call a "think like a graph" programming model in opposition to the "think like a vertex" model, implemented for example in Pregel. This graph-centric model opens the partition structure of the graph up to users in order to be able to bypass heavy message passing and scheduling mechanisms within a partition. Giraph++ uses an asynchronous parallel (AP) computation model. They implemented this model on top of Giraph. This graph-centric model enables a lower level access, which is often needed to implement algorithm-specific optimisations. Giraph++ supports asynchronous computations and allows graph topology mutations. While Giraph itself, like Pregel, does not provide a persistence model, Apache **Gora** [6] fills this gap. Gora provides a data model and persistence for big data. It supports different data stores, *e.g.,* different column stores, key-value stores, document stores, and even relational database systems. Therefore, it requires to define a data model (using a JSON-like schema) and a XML mapping file, defining how the graph data is mapped to a persistent data store.

GraphLab [228] is a graph-based, distributed computation framework written in C++. It has originally been developed at Carnegie Mellon University and is available under the Apache license. The Turi company has been created to commercialise and continue the development of GraphLab [67]. Recently, Turi has been bought by Apple Inc. GraphLab was initially created for machine learning tasks but is now used for a broader range of tasks. While Pregel and Giraph are bulk synchronous message passing abstractions, GraphLab is an asynchronous distributed shared memory abstraction. Vertex programs can directly access information on the current vertex, adjacent edges, and also adjacent vertices (regardless of the edge direction). The computation model of Pregel and Giraph runs all vertex programs simultaneously in a sequence of supersteps. In GraphLab, vertex programs have shared access to a distributed graph, where data is stored on every vertex and every edge. GraphLab uses a so-called gather, apply, and scatter (GAS) model. The gather phase calls a *gather()* function in the vertex class on each edge on the vertex's adjacent edges, which returns a value for each gather. An apply phase, which calls an *apply()* function on the vertex class, passing the sum of the gather values as an argument. The scatter phase calls a *scatter()* function on the vertex class, again for each edge on the vertex's adjacent edges. In contrary to Pregel and Giraph, where a change made to a vertex or edge data is only visible in the next superstep, in GraphLab changes are immediately visible to adjacent vertices. The GraphLab computation model allows vertex programs to schedule neighbouring vertex programs to be executed in the future. GraphLab prevents neighbouring program instances from running simultaneously in order to ensure serialisability. GraphLab was designed for sparse data with local dependencies, iterative algorithms, and potentially asynchronous executions. Moreover, GraphLab provides a set of consistency models, allowing users to specify the consistency requirements without the need to develop their own locking strategies. With **SNAPLE**, Kermarrec *et al.,* [198] present an interesting approach for link prediction for GAS graph computation models, implemented on top of GraphLab. In [227] the authors extend the GraphLab abstraction to a distributed setting while preserving strong data consistency guarantees. GraphLab supports synchronous and asynchronous computations and allows graph structure changes with the exception of deletions.

PowerGraph [161] is another large-scale graph-parallel abstraction, like Pregel and GraphLab, but was specifically designed for graphs having highly skewed power-law degree distributions, as it is the case for many real-world graphs. PowerGraph can be seen in many ways as a successor of GraphLab. In their work, Gonzalez *et al.,* [161] present a new approach to distributed graph placement and representation for such power-law graphs. More specifically, they suggest to partition graphs by cutting vertices instead of cutting edges. Moreover, update functions for a vertex in PowerGraph are parallelised across multiple machines instead of running on a single one. Instead of factoring the computation over vertices, like many other approaches, they suggest to factor computation over edges. They argue that this often leads to a greater parallelism and reduces network communication and storage costs. PowerGraph uses a GAS computation model. In its gather phase information about adjacent vertices and edges is collected by a sum over the neighbourhood of the vertex. The sum function is user-defined and must be commutative and associative. The result of the sum operation is then used in the apply phase to update the central vertex. The scatter phase, finally, uses this value to update the data on adjacent edges. PowerGraph aims to

split high-degree vertices and to therefore allow greater parallelism in natural graphs. It supports both, synchronous as well as asynchronous computations. PowerGraph allows graph mutations with exception of deletions.

GraphChi [214] is a disk-based graph framework for running large-scale computations efficiently on a single personal computer, *i.e.,* in a not distributed way. Unlike many other graph processing frameworks, GraphChi does not require the graph to be completely in memory for computations but allows to process graphs from secondary storage. GraphChi introduces a parallel sliding window (PSW) method to be able to execute advanced data mining, graph mining, and machine learning algorithms on large graphs using a standard personal computer. The PSW method uses an asynchronous computation model and makes updated values immediately visible to subsequent computations. PSW processes graphs in three steps: 1) loading a subgraph from disk, 2) updating vertices and edges, and 3) writing updates to disk. GraphChi splits the vertices of a graph into disjoint intervals. Each interval is associated to a shard, which stores the edges (in the order of their source) that have a destination in the interval. Then, GraphChi processes a graph in execution intervals, *i.e.,* it processes vertices one interval at a time. After a subgraph for an interval has been loaded from disk into RAM, update functions for each vertex can be executed in parallel. To prevent race conditions (update functions accessing edges concurrently), GraphChi enforces external determinism, which ensures that each execution of PSW produces the same result. This is realised by updating vertices that have edges with both endpoints in the same interval in a sequential order. GraphChi allows graphs to evolve, *i.e.,* it allows changes in the graph structure. Two programming models are supported by GraphChi, the "standard model" and an alternative "in-memory vertices model". The standard programming model assumes that there is not enough RAM to store the values of vertices. Therefore, vertex values are broadcasted via edges. This model can be inefficient if vertex values are large, since they need to be replicated to all edges. For such cases, GraphChi provides an alternative programming model. This model requires that all vertex values can be stored in RAM. As a consequence, update functions can read neighbour values directly, without the need to broadcast vertex values to incident edges.

GRACE [317] is another graph-parallel programming platform. It aims at combining the convenient and relatively simple programming model of BSP with the performance of asynchronous execution. GRACE implements the AP computation model. It separates application logic from execution policies. GRACE provides a synchronous iterative graph programming model (*e.g.,* to debug and test applications) and a parallel execution engine for both synchronous and asynchronous execution. For accessing state, GRACE uses a message passing-based API, which is similar to the one of Pregel. An interesting aspect of GRACE is its customisable runtime. Users can implement their own execution policies. Unlike many other graph-parallel abstractions and programming platforms, GRACE has been developed for single machines (not distributed) and uses a shared memory model for data access. GRACE does not allow to mutate the graph topology.

Signal/Collect [297] is a programming model for synchronous and asynchronous (distributed) graph algorithms. Algorithms in signal/collect are written from the perspective of vertices and edges. The programming model implies that edges "signal" and

vertices "collect". Signalling means that an edge computes a message based on the state of its source vertex. The message is then sent to the target vertex of the edge. Collecting means that a vertex uses the received message to update its state. The signal/collect programming model allows to execute these operations in parallel all over the graph. Thus, it applies an asynchronous programming model. The computation stops when all messages have been collected and all vertex states have been computed. The authors of Signal/Collect introduce a threshold value, which determines if a node should collect its signals or wether it should send signals. Signal/Collect uses a shared memory approach for implementing the signalling. It also allows mutation of the graph structure while processing the graph. In the original paper, Signal/Collect didn't support distributed computations, however, the Signal/Collect implementation has been extended to support distributed computations in the meantime.

Trinity[1] [288] from Microsoft is a general purpose graph engine over a distributed memory cloud. It leverages graph access patterns in online and offline computations to optimize memory and communication. Trinity aims at supporting efficient online query processing and offline analytics on large graphs on commodity machines. The Trinity architecture organises the memory of multiple machines into a globally addressable, distributed memory (called a memory cloud) to support large-scale graphs. A Trinity system consists of three main components that communicate through a network: slaves, proxies, and clients. Slaves are responsible to store graph data and to perform computations (including sending and receiving messages) on that data. Trinity proxies are middle tiers between slaves and clients and only manage messages but don't "own" any data. Finally, clients are user interface tiers between the Trinity system and users. The Trinity memory cloud is basically a distributed key-value store. In addition, Trinity offers a message passing framework. Trinity enables users to define graph schemas, communication protocols, and computations with the "Trinity Specification Language". The data model of Trinity supports graphs (composed from nodes and edges) on top of an in-memory key-value store. This is somehow similar to how graphs are stored in the approach presented in this thesis, although we use key-value stores for persisting graph data on disk. The key is a system-wide unique identifier and the value is the node. In Trinity, when the value is associated with a schema, the value is called *cell* and the key *cell id*. While trinity ensures atomicity of operations on a single cell (key-value pair), it does not provide ACID transactions.

GraphCEP [235] is an approach to combine graph processing and parallel CEP for real-time data analytics. GraphCEP is built around a split–process–merge architecture. Splitters are responsible for appropriately splitting the incoming event streams in a way that operator instances can detect occurring patterns. Operator instances can run in parallel and use an interface to the distributed graph processing system. This allows to further parallelise the processing of graph structured data in a single operator instance. Mergers, reorder the concurrently detected events from operator instances into a deterministic order. The GraphCEP framework combines stream partitioning (splitters) with distributed graph processing (operator instances). GraphCEP uses parallel CEP for stream processing and a GAS API for distributed graph processing. The contribution made by GraphCEP is this combination of parallel CEP and graph processing. It does not implement a new distributed graph processing engine but is

[1] Trinity is the name of the research project, the public release has been renamed to Graph Engine

designed to use already existing GAS graph processing frameworks.

Table 3.2 summarises and compares the above discussed graph processing frameworks. The table shows for each graph processing framework if it requires the graph to be fully in memory (no storage) while processing, what computation model it applies and if it allows to mutate the graph structure while processing, if the source code is available, if it supports distributed computing, and if it supports stream processing of graphs.

Besides these, there exists a wide variety of additional graph processing frameworks, which are conceptually similar to the ones discussed above. They mostly provide optimisations for specific use cases or are alternative implementations. For example, Graph Processing System (GPS) [279] is another (optimised) open source implementation of Pregel, provided by Stanford. Apache Hama [8] is an implementation of the BSP programming model and includes in addition packages for query processing, graph computing, and machine learning. GraM [322] is another efficient and scalable graph engine from Microsoft Research and the University of Peking for a large class of widely used graph algorithms, which focuses—like the work presented in [263]—especially on multicore processing. GBASE [192] is another scalable and general graph management system from Carnegie Mellon University and IBM T.J. Watson, built on top of map-reduce/Hadoop. Qin *et al.,* [266], Bu *et al.,* [103] (HaLoop), and Kang [193] (PEGASUS) present similar approaches, using optimised map-reduce for graph processing. In the same direction, Ekanayake *et al.,* [137] propose with Twister a runtime for iterative map-reduce. Cassovary [166] is an open source processing library from Twitter for non-distributed large-scale graphs. Presto [312] is a distributed system for machine learning and graph processing that extends R and addresses many of its limitation, like leveraging multi-cores and dynamic partitioning of data to mitigate load imbalance. Parallel Boost Graph Library (BGL) [162] and MultiThreaded Graph Library (MTGL) [92] are generic libraries for distributed graph computation. PrIter [332], Piccolo [262], Galois [31], Naiad [248], DryadLINQ [325] are systems for general-purpose distributed data-parallel computing. These are suitable but not limited for graph processing. X-Stream [276] (single shared-memory machine) and its successor Chaos [275] (multiple machines) are systems for processing both in-memory and out-of-core graphs.

While most graph processing frameworks require the graph to be completely in-memory while processing [113], others, like Roy *et al.,* [275], suggest to process graphs from secondary storage. We allow in our approach to store graph data on secondary storage, since even with the availability of huge clusters at a certain point the limit of in-memory solutions is reached. Given that our proposed approach is built to evolve extensively in time and many worlds, the need for secondary storage is even more underlined, since many different versions of nodes can co-exist, making graphs even bigger. In addition, as argued in Chapter 1, the available computation power of CPSs in the context of this thesis is often limited, *i.e.,* data is often not processed in huge data centres but on commodity machines. This makes it necessary to process graphs from secondary storage and not only from main memory. Most graph processing frameworks are not built for stream processing and offer only limited support for it. GraphCEP is one of the few exceptions. Instead, most graph processing frameworks first load graph data from a storage, perform the graph processing, and then store the results

Table 3.2: Summary and comparison of important graph processing frameworks

	storage	computation model	source available	distr. comp.	stream support
Pregel	✗	BSP sync. exec. message passing graph mutation	✗	✓	✗
Giraph	✗(natively) ✓(w. Apache Gora)	BSP sync. exec. message passing graph mutation	✓	✓	✗
GiraphUC	✗	BAP async. exec. message passing graph mutation	✓	✓	✗
Giraph++	✗	AP async. exec. message passing graph mutation	✓	✓	✗
GraphLab	✗	GAS sync. exec. and async. exec. distr. shared memory no deletion mutation	✓	✓	✗
PowerGraph	✗	GAS sync. exec. and async. exec. distr. shared memory no deletion mutation	✓	✓	✗
GraphChi	✓	PSW async. exec. disk-based graph mutation	✓	✗	✓
GRACE	✗	AP supports sync. and async. exec. shared memory no graph mutation	✗	✗	✗
Signal/Collect	✗	signal/collect sync. exec. and async. exec. shared mem. f. signaling graph mutation	✓	✗ (recently added)	✗
Trinity	✗	no specific comp. model random graph access distr. memory cloud message passing	✓	✓	✗
GraphCEP	✗	parallel CEP for stream GAS for graph processing	✓	✓	✓

back to a persistent storage. In contrary, our approach requires strong support for frequently changing data. The computation model of our approach is more inspired by the distributed shared-memory model of GraphLab rather than the BSP model of Pregel. Whereas none of the above mentioned graph processing frameworks support time evolving graphs, some graph abstractions support graphs evolving over time. These are discussed in Chapter 3.6. Moreover, current graph abstractions don't support to evolve in many different worlds, *i.e.,* they lack concepts to support analysing many different hypothetical actions (*cf.* Chapter 3.7). Although, the proposed data model of our approach is essentially also a graph structure, it is semantically richer due to the fact that data is modelled in terms of domain concepts and relations between these domain concepts, *i.e.,* they have a strongly typed schema, rather than being simple nodes and edges. This essentially allows to model domain knowledge, data, and learning in the same model and with the same concepts (*cf.* Chapter 7).

3.5 Graph databases

Closely related to graph processing frameworks are graph databases.

Neo4j [47] is one of the first and most well-known graph databases today. It offers native graph storage, processing, querying, and is fully ACID compliant. Neo4j is written in Java and can be accessed from applications written in other languages using Neo4j's query language Cypher [23]. The data model of Neo4j consists of nodes and relationships (edges). Both, nodes and relationships can contain properties. In addition to properties and relationships, nodes can be labelled. Nodes have unique conceptual identities and are typically used to represent the entities of a domain. Every relationship must have a start node and an end node. Relationships are defined between node instances, not node classes. Labels are named graph constructs, which are used to group several nodes into sets. This allows queries to operate on smaller sets instead of the whole graph, making queries more efficient. Neo4j supports replication but until now it does not offer support for distribution. The community edition of Neo4j is open source under a GPL license. An enterprise edition is developed by Neo Technology Inc. under a commercial as well as a free AGPL license.

DEX [234], [233] (recently rebranded to Sparksee) is a high performance graph database management system based on bitmaps and additional secondary structures. DEX uses bitmaps as primary structures to store and manipulate the graph data, since they allow to represent information in a compact way and can be operated efficiently with logic operations. The logical data model of DEX defines a labeled and attributed multigraph as $G = \{L, N, E, A\}$, where L is a collection of labels, N the collection of nodes, E the collection of edges (directed or undirected), and A the collection of attributes. Labeled graphs in DEX provide a label for nodes and edges, defining their types. In DEX two types of graphs are distinguished, the *DbGraph* is the persistent graph that contains the whole database, and *RGraphs* that are used to temporarily query results. The design of DEX follows four goals: 1) it must be possible to split the graph into smaller structures for improved caching and memory usage, 2) object identifiers for nodes and edges should be used to speed-up graph operations, 3) spe-

cific structures must be used to accelerate the navigation and traversal of edges, and 4) attributes should be indexed to allow queries over nodes and edges based on value filters [234]. DEX uses bitmaps to define which objects (nodes or edges) are selected or related to other objects. As auxiliary structures, DEX uses maps with key values associated to bitmaps or data values. These two structures are combined to build links: binary associations between unique identifiers and data values. This allows for an identifier to return the value, and the other way around for a value, to return the associated identifiers. A graph is then built out of links, maps, and bitmaps. The original DEX papers [234], [233] contain no information about transaction support but the current version [61] is fully ACID conform. DEX has been initially developed by the Data Management group at the Polytechnic University of Catalonia (DAMA-UPC) and later on by a created spin-off called Sparsity Technologies. It comes with a dual license, a free one for evaluation and academic purposes and one for commercial usage. The source code is not available as open source.

HyperGraphDB [187] is a graph database which was specifically designed for artificial intelligence and Semantic Web projects. It is based on generalised hypergraphs (*e.g.,* as proposed in [99], [159]) where hyperedges can contain other hyperedges. Hypergraphs are extensions of standard graphs that allow edges to point to more than two nodes. In HyperGraphDB edges can also point to other edges. It is transactional and embeddable. As stated by Iordanov [187], the representational power of higher-order n-ary relationships was the main motivation behind the development of HyperGraphDB. The basic structural unit in the HyperGraphDB data model is called *atom*. Each atom has a tuple of atoms associated. This tuple is called *target set* and its size is referred to as the atom's *arity*. Atoms of arity 0 are nodes, while atoms of arity > 0 are links. The set of links pointing to an atom a is called the *incidence set* of atom a. Each atom in HyperGraphDB has a value, which is strongly typed. HyperGraphDB uses a two-layered architecture: the hypergraph storage layer and a model layer. As storage layer, HyperGraphDB suggests—like we do in our approach—to use key-value stores. The only requirement imposed by HyperGraphDB is that indices support multiple ordered values per single key. The model layer contains the hypergraph atom abstraction, the type system, caching, indexing, and querying. HyperGraphDB supports data distribution, using an agent-based peer-to-peer framework. Activities are asynchronous and incoming messages are dispatched using a scheduler and processed in a thread pool. HyperGraphDB is open source under the LGPL license.

Titan [65] is a distributed graph database. It supports ACID transactions, eventual consistency, and is designed to be used with different data backends, *e.g.,* Apache Cassandra, Apache HBase, and Oracle BerkeleyDB. Titian is open source and supports different processing frameworks, among others, Apache Spark, Apache Giraph, and Apache Hadoop. It supports TinkerPop Gremlin [15] queries. Titan is open source und the Apache 2 license.

OrientDB [52] is another distributed database. A distinctive characteristic of OrientDB is its multi-model, besides from being a graph database, it also supports document and key-value data models. OrientDB supports schema-less, schema-full, and schema-mixed modes and allows to query data with a slightly extended SQL (OrientDB SQL) variant and with TinkerPop Gremlin. It supports ACID transactions, sharding, and provides encryption models for data security. OrientDB implements several in-

Table 3.3: Summary and comparison of important graph databases

	distributed	transactional	query language (ql)	source available
Neo4j	✗	fully ACID	Cypher (declarative)	✓
DEX	✗	fully ACID (current version)	API for graph traversal no explicit ql	✗
HyperGraphDB	✓ (depends on underlying k/v store)	transactional (depends on underlying k/v store)	API for graph operations relational-style queries	✓
Titan	✓	fully ACID and eventual consistency	TinkerPop Gremlin API for graph traversal	✓
OrientDB	✓	fully ACID and eventual consistency	TinkerPop Gremlin OrientDB SQL (slightly extended SQL) API for graph traversal	✓
InfiniteGraph	✓	fully ACID and eventual consistency	TinkerPop Gremlin API for graph traversal	✗

dexing strategies based on B-trees and extendible hashing. This allows fast traversal ($O(1)$ complexity) of one-to-many relationships and fast add/remove link operations. OrientDB comes in two versions, a free community edition licensed under Apache 2 and a commercial enterprise edition with professional support. The community edition is available as open source.

InfiniteGraph [36] is a distributed graph database, which is, in the meantime, integrated into the thingspan [37] analytics stack. The InfiniteGraph graph model is a labeled, directed multigraph. Edges in InfiniteGraph are first-class entities with their own identity. InfiniteGraph provides ACID transactions and supports a schema-full model. InfiniteGraph has been developed by Objectivity Inc. and is not available as open source.

Table 3.3 summarises and compares the discussed graph databases. The table shows for the discussed graph databases if they support distribution, their transaction model, the used query language, and if the source code is available.

Like it is the case for graph processing frameworks, there exists also a wide variety of additional graph databases, which are conceptually similar to the ones discussed but which provide optimisations for specific use cases or are alternative implementations. ArrangoDB [17] is among the most well-known ones and supports multiple data models: graph, key-value, and document. Another one is FlockDB [30] which has been developed by Twitter in order to store social graphs. AllegroGraph [2] is a triple store designed to store RDF triples. Stardog [63], GraphDB [35], Dgraph [25], InfoGrid [40], blazegraph [21], GraphBase [33], and VelocityDB [68] are other examples of mostly commercial, generic graph databases.

The boundary between graph databases and graph processing frameworks is not sharp: on the one hand, some graph processing frameworks offer persistence and, on the

other hand, some graph databases only provide weak consistency models, like BASE. Nonetheless, most graph databases offer stronger consistency and transaction models than graph processing frameworks, whereas the latter have a stronger focus on graph traversing and distributed processing. The contribution of this dissertation is somehow in the middle of these two categories. Despite the main focus is a multi-dimensional graph data model for near real-time analytics, we also put a strong emphasis on efficient storage concepts for this graph model. None of the mentioned graph databases allow to natively represent time nor many different hypothetical worlds. Although, Neo4j does not support time natively, there are some discussions and patterns on how to best model time dependent data [55], [32]. These are discussed in more detail in Chapter 3.6.

3.6 Analysing data in motion

The need to deal with temporal data appears in many domains and has been discussed by different communities over time.

3.6.1 Temporal databases

Considering versioning (or time) as a crosscutting concern of data modelling has been discussed for a long time, especially in **(relational) database communities**. In [115] Clifford *et al.,* provide a formal semantic for historical databases and an intentional logic. Rose and Segev [273] incorporate temporal structures in the data model itself, rather than at the application level, by extending the entity-relationship data model into a temporal, object-oriented one. They also propose a temporal query language for the model. Ariav [273] also introduces a temporally-oriented data model (as a restricted superset of the relational model) and provides a SQL-like query language for storing and retrieving temporal data. He adds a temporal aspect to the tabular notion of data and provides a framework and a SQL-like query language for storing and retrieving data, taking their temporal context into account. The works of Mahmood *et al.,* [231] and Segev and Shoshani [284] go into a similar direction. The latter also investigate the semantics of temporal data and corresponding operators independently from a specific data model in [283]. Some of these temporal features are integrated into the SQL:2011 or ISO/IEC 9075:2011 [212] standard. This standard provides language extensions for temporal data definition and manipulation. For example, it allows to define a time period, which basically uses two standard table columns to define the start and end of a named time period. Major database vendors, like IBM, Oracle, and Microsoft implemented at least some of the temporal features based on this standard into their products. Regardless of noteworthy work in this area, temporal relational databases found only little consideration in practice. This mostly leaves the management of time to the application level, which often leads to tedious and complicated ad-hoc implementations. Besides being difficult to maintain these solutions are usually highly specialised and not reusable. In a newer work [111], Google embeds versioning at the core of their *BigTable* implementation by allowing each cell in a BigTable to contain multiple versions of the same data (at different timestamps).

Although **Neo4j** doesn't provide native support for managing time-evolving data, there are some best practices and patterns about how to model temporal data. In [55] methods how to model a time-varying social graph to mine for proximity of individuals, where nodes are individuals and edges represent proximity/contact relations of individuals, are discussed. First, the term frame is introduced, which is the basic temporal unit. A frame has a time interval defined by a start time and an end time. A time-varying graph is modelled in the following way:

- nodes of the social graph are nodes in Neo4j
- edges of the social graph are nodes in Neo4j
- intervals of time are nodes in Neo4j

Graph changes can be tracked by traversing from one frame to the next. Indexing of the timestamped sequence of frame nodes enables an efficient random access to the frame timelines. In [32], GraphAware, a Neo4j TimeTree library is presented. GraphAware is a library for managing time in Neo4j as a tree of time instants. It enables to represent events as nodes and link them to Neo4j nodes representing instants of time. This allows to capture the time of an event's occurrence. For example, to model that an email was sent on a specific day, in GraphAware a node `Email` would be created and linked to a node `Day` using a `SENT_ON` relation. The library provides an API to create basic, time-related queries. While these best practices and patterns allow to model time-evolving graphs, these solutions are rather complicated, even for small examples. Moreover, time is modelled like domain concepts and therefore scattered over the whole graph. This brings additional complexity for creating, updating, and querying domain nodes, due to the explicit management of time. Also, such solutions are usually application specific and cannot easily be reused in other applications (with exception of the library-based approach).

3.6.2 Temporal RDF and OWL

The necessity to store and reason about versioned data has also been discussed in the area of the **Semantic Web** and its languages, like RDF [217] and OWL [315]. For example, Motik [247] presents a logic-based approach for representing versions in RDF and OWL. He also proposes to extend SPARQL to temporal RDF graphs and presents a query evaluation algorithm.

3.6.3 Model versioning

Recently, the need to efficiently version models has been explored in the domain of **model-driven engineering**. However, model versioning has been mainly considered so far as an infrastructural issue in the sense that models are artefacts (*i.e.,* meta models) that can evolve and must be managed in a similar manner to textual artefacts

like source code, rather than from a runtime model perspective. Moreover, model versioning alone does not address the problem of discretisation, *i.e.,* it does not provide a temporal semantic, which would allow model elements to evolve independently and at different paces while preserving a consistent way to navigate the model in space and time. Kerstin Altmanninger *et al.,* [73] analyse the challenges coming along with model-merging and derive a categorisation of typical changes and resulting conflicts. Building on this, they provide a set of test cases which they apply on state of the art versioning systems. Koegel and Helming [204] take a similar direction with their *EMFStore* model repository. Their work focuses on how to commit and update changes and how to perform a merge on a model. Brosch *et al.,* [102] also consider model versioning as a way to enable efficient team-based development of models. They provide an introduction to the foundations of model versioning, the underlying technologies for processing models and their evolution, as well as the state of the art. Taentzer *et al.,* [300] present an approach that, in contrast to text-based versioning systems, takes model structures and their changes over time into account. In their approach, they consider models as graphs and focus on two different kinds of conflict detection, operation-based conflicts between different graph modifications and the check for state-based conflicts' on merged graph modifications. These works consider versioning at a model level rather than at a model element level. Moreover, these approaches focus on versioning of meta models whereas our work focuses on versioning of runtime/execution models. Our approach enables not only to version a complete model, but considers versioning and history as native mechanisms for any model element. Moreover, versioning in the modelling domain is usually considered from a design/architecture/infrastructural point of view, and models are versioned as source code files would be. In contrast to this, our versioning approach regards the evolution of model elements from an application point of view (*e.g.,* BigTable or temporal databases). It allows to keep track of the evolution of domain model elements—their history—and use this history efficiently on an application level.

Most of the above mentioned work address storing and querying of versioned data but largely ignores the handling of versioning at an application level. However, many reasoning processes require to explore simultaneously the current state and past history to detect situations like a system state flapping. Our approach proposes to consider model versioning and history as native mechanisms for modelling foundations. We not only efficiently store historical data (what is done in other works before), but we propose a way to seamlessly use and navigate in historized models. Also, we do not extend a specific data model (*e.g.,* the relational data model or object-oriented one) but use model-driven engineering techniques to integrate versioning as a crosscutting property of any model element. We aim at providing a natural and seamless navigation into the history of model element versions.

3.6.4 Time series databases

With the emergence of big data, the Internet of Things, and cyber-physical systems temporal aspects of data, in form of **time series databases**, lately gained again in importance [224]. Time series databases are optimised for handling timestamped data values. Usually, a time series is a successive sequence taken at equally spaced

points in time. Time series databases are heavily used for data mining and forecasting [196], [140], [169], [110]. One of the newest time series databases which received much attention lately is *influxdb* [39]. They position themselves as an IoT and sensor database for real-time analytics. While many time series databases provide interesting features, like SQL-like query languages, their data model is essentially flat and does not support complex relationships between data. Mostly, only a flat value (like an integer or double) is stored together with a timestamp. This also counts for Atlas [18], which was developed by Netflix to manage dimensional time series data for near real-time operational insights, and *OpenTSDB* [51]. *RRDtool* [58] is another famous data logging and graphing system for time series. All of this work has in common that it provides high performance storage and management specialised for time series data. However, these solutions provide very little support for richer data models, like graphs, if any. Usually only one flat value together with a timestamp is stored, therefore complex data models with relationships between data are difficult to build with time series'. Also, most time series' require equally spaced points in time and do not allow data to evolve at different paces. While this is sometimes the case for regular sensor measurements, it does not fit the numerous and heterogenous events occurring in complex CPSs and IoT. Time series databases make it also difficult to correlate and analyse events composed from different independently timed values, *i.e.,* values not stored at the same time intervals.

3.6.5 Temporal graph processing

The need to analyse time-dependent events has also been discussed in **graph processing** communities. This is especially interesting in the context of this work, since our proposed model-driven live analytics approach essentially also relies on a complex graph model (of interacting model elements, *i.e.,* objects).

Chronos [171] is a storage and execution engine for iterative graph computation on temporal graphs. It is designed for running in-memory, both on multi-core machines and in distributed settings. Chronos defines a temporal graph as a sequence of graph snapshots at specific points in time. Data locality is a central design goal behind Chronos. A stack of graph snapshots can be seen from two dimensions, 1) the time dimension across snapshots and 2) the graph-structure dimension among neighbours within one snapshot. Chronos favours time(-dimension) locality over structure(-dimension) locality. In addition, Chronos schedules graph computation to leverage the time-locality in the temporal graph layout, *i.e.,* the computation model of Chronos uses batch operations for each vertex (or each edge) across multiple snapshots. Chronos adopts locality aware batch scheduling (LABS). Instead of assigning different snapshots to different cores, Chornos assigns different graph partitions (across multiple snapshots) to different cores. To store temporal graph data on disk, Chronos defines snapshot groups. A snapshot group is valid for a certain time interval and consists of a full snapshot for the beginning of the interval and all deltas until the end of the interval. This allows to reconstruct temporal graphs within an interval while keeping a compact format to store the temporal graph data. Besides describing this disk layout, no further information about concrete storage technologies are provided in [171]. The source code of Chronos is not available. Chronos addresses the need to analyse

temporal data by optimising the data locality and scheduling of the graph computation. Nonetheless, the temporal data model of Chronos itself is rather standard and a temporal graph is represented as a sequence of full graph snapshots. This comes with some limitations: First of all, full graph snapshots are expensive in terms of memory requirements (both on disk and in-memory). Secondly, for every small change in the graph it would be necessary to snapshot the graph in order to keep track of the full change history. Thirdly, the continuous semantic of time is losed by the discretisation in snapshots. Thus, navigating in the time and space dimensions of the graph is problematic, which complicates analytic algorithms.

GraphTau [188] is a time-evolving graph processing framework built on top of Apache Spark. It represents time-evolving graphs as a series of consistent graph snapshots. Dynamic graph computations are performed as a series of deterministic batch computations on discrete time intervals. A graph snapshot in GraphTau is a regular graph, based on two Spark RDDs, one for vertices and one for edges. Design goals of GraphTau are a "consistent and fault tolerant snapshot generation", a "tight coordination between snapshot generation and computation stages", and " operations across multiple graph snapshots" [188]. GraphTau allows algorithms to run continuously as new data becomes available. Two computation models are offered, 1) the *pause-shift-resume* model and the online *rectification model* The pause-shift-resume model starts running an algorithm with the availability of the first snapshot, pauses if a new snapshot becomes available, switches to the new snapshot, and finally resumes computation on the new graph. On the other side, the rectification model allows applying differential computation on a wide variety of graph algorithms. GraphTau tries to unify data streaming and graph processing by building on Spark's RDD data structure and its streaming mechanisms. For storing graph snapshots, GraphTau relies on graph databases like Titan and Neo4j. The source code of GraphTau is not available. Similarly to Chronos, GraphTau models temporal graphs as a sequence of snapshots—with the before mentioned consequences.

With **ImmortalGraph** [239] Miao *et al.,* presents an extension of Chronos. Like it is the case for Chronos, data locality is a central design goal of ImmortalGraph. ImmortalGraph takes both, persistent storage and memory into account and regards data locality for time and graph-structure. In addition, they propose a locality-aware scheduling for iterative computations on temporal graphs. Like Chronos, ImmortalGraph uses *snapshot groups* for their graph layout. A snapshot group consists of a graph state representing a time range $[t_1, t_2]$. It contains a snapshot of the complete graph at time t_1 and graph updates until t_2. Graph states between t_1 and t_2 can then be computed by applying the updates on the graph snapshot at t_1. A temporal graph then consists of a series of snapshot groups for successive time ranges. Within a snapshot group, updates can be stored in structure-locality or time-locality order. Like Chronos, ImmortalGraph uses LABS. Similar to most other approaches, ImmortalGraph represents a temporal graph as a sequence of graph snapshots. This leads to potentially a huge number of snapshots (in case of frequently changing data), which are difficult to analyse efficiently. In addition, in order to find a specific version of the time-evolving graph, ImmortalGraph needs to compute the version based on a snapshot and deltas, which can be inefficient for highly time-dependent analytics.

Instead, with **Temporal graphs**, Kostakos [208] discusses a graph representation

that encodes temporal data into graphs. Kostakos assumes for his approach that events are instantaneous and have no temporal duration, *i.e.,* an event starts and finishes at the same point in time. In some respects similar to what we propose, he suggests to create one node instance per semantic node per point in time. Then, he uses unweighted directed edges to link node instances that are linked to each other. Kostakos links consecutive pairs of node instances of a node instance set with additional directed edges, where the weight represents the temporal distance between the pair. In addition, he defines a number of temporal graph metrics, such as temporal proximity, geodesic proximity, and temporal availability, on which he puts a large focus on. While the presented data model is interesting, it is rather abstract. Important topics, like storage, distribution, graph processing and traversing, details on creation of new nodes and modifications of existing nodes, are not presented in detail—neither is a concrete implementation provided.

Khurana and Deshpande [202] propose with **Historical Graph Store (HGS)** another interesting approach for storing and analysing historical graph data at large scale. HGS consists of two major components, the Temporal Graph Index (TGI) and the Temporal Graph Analysis Framework (TAF). TGI is the index component to efficiently store the graph history by partitioning and encoding differences over time (called deltas). These deltas are designed to efficiently retrieve temporal graph primitives such as neighbourhood versions, node histories, and graph snapshots. TGI uses distributed key-value stores (like Apache Cassandra) to store the partitioned deltas. TGI is an extension of DeltaGraph [200], a previous work of the authors. It contains various forms of differences in the graph, such as atomic events and changes in subgraphs over intervals of time. The temporal index of TGI is basically a set of different changes or deltas. Deltas are stored in three different forms in TGI. First, atomic changes are stored in chronological order in event list partitions. This enables direct access to changes of a part or whole of the graph at specific timepoints. Secondly, the state of nodes at different timepoints is stored in form of derived snapshot partition deltas. This enables direct access to the state of a neighbourhood or the entire graph at specific timepoints. Thirdly, TGI stores a meta index of pointers to the list of chronological changes for each node. TAF is a framework to specify a wide range of temporal graph analysis tasks. The execution engine of TAF is implemented on top of Apache Spark. The data model of HGS defines a temporal graph store as a sequence of snapshots. Similar to what we propose, HGS is a node-centric approach, *i.e.,* the graph is seen as a set of evolving vertices over time and edges are considered as attributes of the nodes. While TGI is a powerful indexing strategy for time-evolving graphs, the fact that TAF is based on Spark comes with the respective restrictions (*cf.* Chapter 3.2.3) for live analytics. Furthermore, the organisation of representing time-evolving graphs in snapshots makes it less suitable for analysing frequently changing data, since snapshots from different timepoints must be computed. The source code of HGS is not available as open source.

Kineograph [112] from Cheng *et al.,* is a distributed system that takes incoming data streams to construct a continuously changing graph. It also supports graph-mining algorithms to extract insights from frequently changing graph structures through an incremental graph computing engine. To support graph mining algorithms that assume a static graph structure, Kineograph supports an efficient snapshotting strategy,

called *epoch commit protocol*. This decouples graph mining from graph updates. Kineograph first imports raw data through a set of *ingest nodes*. Ingest nodes analyse the incoming data records and create transactions of graph updates. Each transaction is assigned with a sequence number and then distributed to *graph nodes*. Graph nodes form a distributed in-memory key-value store. The storage engine maintains with each vertex an adjacency list as metadata and independently stores application data. First, graph nodes store updates from ingest nodes, then each ingest node reports the graph update in a global progress table. A snapshotter periodically instructs graph nodes to take a snapshot based on the sequence numbers in the progress table. The sequence numbers are used as a global logical clock defining the end of an epoch. Graph nodes then can commit stored local graph updates in an epoch following a pre-determined order. Each epoch commit produces a snapshot. Graph updates trigger incremental graph computations on the new snapshot to update associated values of interest. Kineograph's partitioning is based on the hashing of vertex ids, without any locality considerations. In order to enable various graph mining algorithms that might require different communication models, Kineograph supports the *push* [229] and the *pull* [262] models. The source code of Kineograph is not publicly available.

G* [215] is a framework for large-scale distributed processing of dynamic graphs. G* allows to store graph data on a persistent storage. Like many other approaches, time-evolving graphs in G* are represented as a sequence of graph snapshots (regular clones of the graph at different timestamps). However, in order to optimise storage, G* uses an index, called compact graph index (CGI), which allows to take advantage of commonalities among the graph snapshots. This index contains tuples of (vertex ID, disk location) pairs. CGI stores only one (vertex ID, disk location) tuple for every vertex version in a collection for the combination of graphs that contains that version. G* focuses on data locality by assigning a subset of vertices and their outgoing edges from multiple graphs to a server. This allows to access all edges of a vertex without contacting other servers. Queries can be executed in parallel using distributed operators to process graph data in parallel. The source code of G* is not publicly available and no explicit support for stream processing, *i.e.,* frequent small changes, is provided.

Table 3.4 summarises the discussed temporal graph processing frameworks. It shows for each discussed temporal graph processing framework the used storage concept, if it supports distributed processing, if the source code is available, how the temporal graph is represented, and if stream processing is supported.

Most of these approaches have in common that they represent time-evolving graphs, in some form or another, as a sequence of snapshots. Different approaches use different trade-offs between required storage space and performance to access a graph snapshot at a specific point in time. Using full snapshots, for example, comes with high storage costs (disk and in-memory) but allows fast access to graph data at any timepoint. Others, like HGS, provide more complex indexing strategies, such as using regular snapshots and, in addition, store deltas for a snapshot for a certain time interval. The state of a graph at any timepoint in this interval is then recomputed by using the snapshot and applying the deltas. We found that snapshotting (both, full and partly snapshotting) comes with comparable high memory requirements (in disk and

Table 3.4: Summary and comparison of temporal graph processing frameworks

	storage	distributed processing	source available	temp. graph represent.	stream support
Chronos	no details provided	✓	✗	seq. of snapshots	✗
ImmortalGraph (ext. of Chronos)	no details provided	✓	✗	seq. of snapshots	✗
GraphTau	uses graph DBs, *e.g.,* Titan, Neo4j	✓	✗	seq. of snapshots	✓
Temporal graphs (Kostakos)	no details provided	no details provided	✗	no details provided	✗
HGS	uses distributed k/v stores *e.g.,* Cassandra	✓	✗	seq. of snapshots	✗
Kineograph	yes, but no details provided	✓	✗	seq. of snapshots	✓
G*	yes, but no details provided	✓	✗	seq. of snapshots	✗

in memory). In addition, the graph data model of most of these approaches requires to keep a full graph snapshot (or at least comparable large parts) for a certain point in time in-memory. All of this makes it difficult to analyse frequently changing graph data in live. This is even more the case if approaches are built around Spark-like or tightly iterative computation models, like it is the case for GraphTau. Additionally, considering frequently changing data, most snapshotting approaches require for every change to snapshot the graph in order to keep track of the full change history. This can lead to a vast amount of snapshots, even for delta-based ones. All of the above mentioned approaches loose the continuous semantic of time [289] by discretising continuously evolving graph data. This makes querying and traversing graph data more complex, since the discretised timepoints must be always taken into consideration, *i.e.,* the state of a graph is not defined between the two closest snapshots. Thus, navigating in the time and space dimensions of the graph is problematic, which complicates analytic algorithms. Last but not least, none of these approaches support representing many different hypothetical graph states and none of these approaches can be easily extended to support this.

3.7 Exploring hypothetical actions

The importance of exploring different hypothetical actions has been discussed in different domains.

In a small scale, Microsoft integrated what-if analysis in their popular spreadsheet tool Excel [41]. It allows to "try" different values (scenarios) for formulas. For instance, it allows to specify a result that a formula should produce and then to evaluate what sets of values will produce this result.

The idea of what-if analysis with hypothetical queries has been discussed in database communities. Balmin *et al.,* [85] proposed an approach for hypothetical queries in OLAP environments. They enable data analysts to formulate possible business scenarios, which then can be consequently explored by querying. Unlike other approaches, they use a "no-actual-update" policy, *i.e.,* the possible business scenarios are never actually materialised but only kept in main memory. In a similar approach, Griffin and Hull [163] focus on queries with form Q *when* $\{U\}$ where Q is a relational algebra query. This paper develops a framework for evaluating hypothetical queries using a "lazy" approach, or using a hybrid of eager and lazy approaches. They present an equational theory and family of rewriting rules. In [78] and [77], Arenas *et al.,* developed an approach for hypothetical temporal queries of form *"Has it always been the case that the database has satisfied a given condition C"*. Despite there is no explicit time in these queries, they call them "temporal" due to a similarity with dynamic integrity constraints.

Although these approaches have a similar goal than our approach, they differ in many major points. First, they mainly aim at data analysts who perform selective queries on a modest number of possible business scenarios to investigate impacts of decisions. In contrary, we aim at intelligent systems and complex data analytics, which need to explore a very large number of parallel actions (*e.g.,* as for genetic algorithms or the presented smart grid case study), which can be highly nested. Moreover, these systems usually face significantly higher demands regarding performance. In addition, most of the above mentioned approaches do not support (or only in a limited manner) the co-evolution of worlds, which is an essential feature of our proposed multi-dimensional graph data model. Another major difference is that our data model is a fully temporal graph, supporting both the exploration of different hypothetical worlds and the temporal evolution of data and temporal queries. Our proposed multi-dimensional graph data model can be used in arbitrary middleware layers for analytics and is independent of the concrete underlying database, whereas most of the work on hypothetical queries has been done on relational databases.

A very recent work in this direction is Noms [48], a decentralised database based on ideas from Git, developed from former Google employees. As in Git, data in Noms is represented as a directed acyclic graph. Every node in the directed acyclic graph (DAG) has a hash, which is derived from the values of the node and from the values of the nodes reachable from that node (transitively). A Noms database is essentially a Merkle DAG [44]. Nodes with different hashes represent different logical values and nodes with same hashes represent identical logical values. Similar to our approach, Noms is implemented on top of key-value stores, such as LevelDB. Data in Noms is always immutable, once it is stored, it is never changed, but represented as a sequence of state changes. Noms stores the entire change history and provides a Git-like workflow, including operations to fork, merge, and efficiently synchronise changes. This makes Noms appropriate for data version control, applications which need to store the data history, and also as an archival database. Noms is in an early development state and currently misses some major features like a query language. While the fork mechanism of Noms can be used to create different what-if scenarios, this isn't the purpose of Noms. Instead, Noms can be best compared with a Git for large-scale, structured data. In particular, it is not very well suited for many small changes over time, due

to the fact that deduplication is managed on a chunk level of 4 KB. If we take the application domain of this thesis, CPSs, or IoT, such small changes, *e.g.,* regularly collected sensor values, are the norm. Moreover, different elements change at very different paces. For such cases, Noms would create complete copies of the dataset. Also, the indexing strategy used in Noms makes it more appropriate for traceability rather than live analytics of frequently changing data.

3.8 Reasoning over distributed data in motion

Reasoning over complex, distributed data which changes very frequently is challenging and has been discussed by different communities. While stream processing frameworks, as discussed, allow to quickly analyse data—even in a distributed scenario—the underlying data model of stream processing frameworks is rather simple and insufficient for modelling data of complex CPSs.

In modelling domains, several authors identified the need of infinite, unbounded models and some sort of model streaming. In [119] Combemale *et al.,* propose a formal definition of an infinite model as an extension of the MOF formalism together with a formal framework to reason on queries over these infinite models. Their work aims at supporting the design and verification of operations that manipulate infinite models. Particularly, they propose formal extensions of the MOF *upperbound* attribute of the *Property* element to define infinite, unbounded collections and iterate over them. For similar reasons, we also define models as infinite streams of model chunks. However, our approach goes beyond this and allows the distribution of model chunks over computational nodes and the definition of repeatable asynchronous operations to lazily (re)load these chunks from remote computational nodes. This is somewhat similar to what is proposed for stream processing [81] in programming languages or database management systems.

In [122] Cuadrado *et al.,* discuss the motivation, scenarios, challenges, and initial solutions for streaming model transformations. They motivate the need for this new kind of transformation with the fact that a source model might not be completely available at the beginning of the transformation, but might be generated step by step. They present an approach and provide a prototype implementation, built on top of the Electic transformation tool. Interestingly, they highlight possible benefits of distributed model transformations by replicating the same transformations in several execution nodes. In a similar direction goes the work of Dávid *et al.,* [127]. They suggest to use incremental model query techniques together with complex event processing to stream model transformations. Their approach foresees to populate event streams from elementary model changes with an incremental query engine and to use a complex event processing engine to trigger transformation rules. This is applied for gesture recognition for data coming from a KINECT sensor. On the contrary to [122] their approach uses derived information regarding the model in the form of change events, which decouples the execution from the actual model. Ráth [269] presents an approach for change-driven model transformations, directly triggered by complex model changes carried out by arbitrary transactions on the model. They identify challenges

for change-driven transformations and define a language for specifying change-driven transformations as an extension of graph patterns and graph transformation rules. This language is a generalisation of previous results on live model transformations, offering trigger events for arbitrarily complex model changes, and dedicated reactions for specific kinds of changes. Our and these approaches have in common that we identify a need for continuous or infinite models. Unlike these approaches we do not stream events or model transformations and use complex event processing engines to detect complex events, but view runtime models itself as continuous streams. We use observers on top of these streams of model chunks to efficiently distribute frequently changing runtime models of CPSs. Beyond model queries, our approach defines a generic distributed modelling layer for runtime usages. However, incremental queries are also a clear illustration of the kind of applications which should be built with such unbounded models.

Several authors worked on the issues of large-scale models [206], model persistence, and the fact that they might grow too big to fit completely into main memory. Benelallam *et al.,* [89] present with *Neo4EMF* a scalable persistence layer for EMF based on Neo4j. Neo4EMF offers on-demand loading of model elements and by using Neo4j as a backend, it enables to use advanced features like online backups, horizontal scalability, and advanced monitoring. Neo4EMF is now called *NeoEMF/Graph* [160]. *MongoEMF* [46] is similar to Neo4EMF but is using mongoDB [45] as a persistence backend. In this context, Connected data objects (CDO) [22] is a distributed shared model on top of EMF, which has been designed as a development-time model repository and runtime persistence framework with pluggable storage backend support. Pagán *et al.,* [259] propose Morsa, an approach for scalable access to large models through on demand loading. They suggest to use NoSQL databases for model persistence and provide a prototype that integrates transparently with EMF. In their evaluation they showed that they have significantly better results than the EMF XMI file-based persistence and CDO. In a similar direction goes the work of Koegel and Helming [204], Gomez *et al.,* [160], or Hartmann *et al.,* [174]. However, none of this work addresses distribution or asynchronicity. To address the scalability of queries, Száárnyas *et al.,* [299] present an adaptation of incremental graph search techniques, like EMF-IncQuery. They propose an architecture for distributed and incremental queries.

Fouquet *et al.,* [148] discuss the challenge how to propagate reconfiguration policies of component-based systems to ensure consistency of architecture configuration models over dynamic and distributed systems. They propose a peer-to-peer distributed dissemination algorithm for the models@run.time context based on gossip algorithms and vector clock techniques that are able to propagate the reconfiguration policies to preserve consistency of architecture configuration models among all computation nodes of the system. While their goal is essentially to propagate changes made in the model of one computation node to the model of other computation nodes, their approach differs significantly from ours. First, they focus mainly on architectural configuration models [246], which are typically of manageable size and can be exchanged in one piece. On the contrary, we focus on big runtime models supporting millions of elements over several thousand distributed instances, making it basically impossible to exchange the complete model in a reasonable time. Secondly, our approach promotes observable streams and asynchronous operations enabling a reactive programming style, which

makes it unnecessary to compare exchanged models in order to find changes in the models propagated from one computation node to others. Last but not least, instead of using a gossip and vector clock-based dissemination strategy to ensure model consistency, we rely on protocols like web sockets or WebRTC together with lazy loading to stream our models between distributed nodes.

As mentioned earlier, graph data structures can be used to represent the context of complex systems, like CPSs. Many graph processing frameworks and graph databases (*cf.* Chapters 3.4, 3.5, and 3.6.5), like Pregel, Giraph, and GraphLab support distributed computing. Some of them, *e.g.,* Titan, OrientDB, and Giraph (through Apache Gora) in addition support persistence. However, while these approaches enable distributed computing, *e.g.,* through BSP, GAS, or map-reduce computation models, they have not been designed for frequently changing data. Instead, their goal is essentially to process computation intensive algorithms and distribute the computation for performance reasons. Our approach, on the other hand, aims at quickly reacting to frequently changing data. Stream processing frameworks, on the other hand, allow to react to frequent changes, however, the simple underlying data models (mostly key-value tuples), are usually insufficient to model the complex data of CPSs.

3.9 Combining domain knowledge and machine learning

Over the last few years, machine learning has become an important technique for many application domains. However, implementing machine learning algorithms and integrating them into applications remains challenging. Different approaches have been taken to address this.

TensorFlow [69] is an interface for expressing machine learning algorithms and an execution engine to execute these on a wide range of devices from phones to large clusters. A TensorFlow computation is represented as a directed graph. Nodes in the graph represent mathematical operations, called *ops*, while the edges represent multidimensional data arrays, called *tensors*. An op takes zero or more tensors, performs computations, and produces zero or more tensors. Two phases are distinguished in TensorFlow. A construction phase where the graph is assembled and an execution phase which uses a session to execute ops in the graph. TensorFlow is used within Google for a wide variety of projects, both for research and for use in Google's products. Similar to our approach, TensorFlow allows to model machine learning at a higher level of abstraction. However, unlike in our approach machine learning is expressed in its own model aside from the domain model and not connected to it. TensorFlow is adapted for image and video recognition, whereas our approach is adapted for learning from frequently changing domain data.

GraphLab [229] goes in a similar direction than TensorFlow. In Section 3.4, GraphLab has been discussed from the perspective of graph processing frameworks. Now, in this Section, GraphLab is discussed from the perspective of a machine learning framework. Low *et al.,* propose an approach for designing and implementing efficient and prov-

ably correct parallel machine learning algorithms. They suggest to use a data graph abstraction to encode the computational structure as well as the data dependencies of the problem. Vertices in this model correspond to functions which receive information on inbound edges and output results to outbound edges. Data is exchanged along edges between vertices. GraphLab aims at finding a balance between low-level and high-level abstractions. In contrary to low-level abstractions GraphLab manages synchronisation, data races, and deadlocks and maintains data consistency. On the other side, unlike high-level abstractions GraphLab allows to express complex computational dependencies using the data graph abstraction. Low *et al.,* [227] present a distributed implementation of the GraphLab abstraction. Like TensorFlow, GraphLab is an interface for expressing machine learning algorithms and an execution engine. While there are similarities, like the idea that machine learning algorithms should be expressed with a higher-lever abstraction, our approach focuses on weaving machine learning algorithms into domain modelling. This allows to use results from learning algorithms in the same manner than other domain data.

In [95] Bishop proposes a model-based approach for machine learning. This approach comes closest to ours. He introduces a modelling language for specifying machine learning problems and the corresponding machine learning code is then generated automatically from this model. As a motivation Bishop states the possibility to create highly tailored models for specific scenarios, as well as for rapid prototyping and comparison of a range of alternative models. With *Infer.NET* he presents a framework for running Bayesian inference in graphical models. Infer.NET is used in several practical applications. Similar to Bishop we propose to express machine learning problems in terms of a modelling language and automate the mapping of a domain problem to the specific representation needed by a concrete machine learning algorithm. While Bishop suggests to specify machine learning problems in separate models with a dedicated modelling language, our approach extends domain modelling languages with the capability to specify machine learning problems together with domain models using the same modelling language. This allows to decompose learning into many small learning units which can be seamlessly used together with domain data.

Stone and Veloso [295] present a survey of multiagent systems from a machine learning perspective. The survey is intended to serve as an introduction to the field of multiagent systems as well as an organisational framework. They discuss robotic soccer as an example application of complex multiagent systems. This survey is interesting in the context of our work since it focuses, like our work, on machine learning for systems that consist of multiple independent entities that interact in a domain.

Domingos *et al.,* [134] propose an approach for incremental learning methods based on Hoeffding bounds. They suggest to build decision trees on top of this concept and show that these can be learned in constant memory and time per example, while being very close to the trees of conventional batch learners. With Massive Online Analysis (MOA) Bifet *et al.,* [94] present an implementation and a plugin for WEKA [168] based on Hoeffding trees. Our contribution is a methodology to weave micro machine learning into data modelling to support applications which need online analysis of massive data streams.

Hido *et al.,* [182] present a computational framework for online and distributed ma-

chine learning. There key concept is to share only models rather than data between distributed servers. They propose an analytics platform, called Jubatus, which aims at achieving high throughput for online training and prediction. Jubatus focus on real-time big data analytics for rapid decisions and actions. It supports a large number of machine learning algorithms, *e.g.,* classification, regression, and nearest neighbour. Jubatus only shares local models, which are smaller than datasets. These models are gradually merged. Jubatus, like our approach, allows independent and incremental computations. However, Jubatus doesn't aim at combining domain modelling and machine learning, neither does it allow to decompose a complex learning task into small independent units, which cab be composed.

These approaches allow to model machine learning and the corresponding machine learning code is then generated automatically, but they don't address the challenge, how machine learning and domain modelling can be combined. While intelligent systems like CPSs have to face many unpredictable events, they also face many predictable situations for which behavioural models can be already defined at design time of the system. In fact, learned information can directly depend on domain knowledge and vice versa. Therefore, modelling domain knowledge and machine learning independently, in different models, with different modelling languages and techniques, can be inefficient.

3.10 Synthesis

While a lot of interesting work has been done in the domain of data analytics, existing solutions do not meet the requirements of analysing complex and frequently changing data of CPSs. Analytics platforms like OLAP, Hadoop, and also Spark have been mainly developed for pipeline-based (distributed) batch analytics. On the other hand, stream processing frameworks like Storm, Flink, and MillWheel do allow fast and distributed processing of frequently changing data, however their underlying data model is rather simple (mostly key-value tuples), which makes it difficult to represent complex structured data, as needed in the context of CPSs. In addition, this simple data model makes it also difficult to combine machine learning and domain modelling in a single model. Graph processing frameworks like Pregel, Giraph, and GraphLab aim at tacking the challenge to model complex data and to efficiently analyse it in a distributed manner. However, they mostly consider only static graph data, *i.e.,* they load a graph into memory, process the data, and subsequently store the result. In other words, graph processing frameworks are predominately not suitable to analyse frequently changing data. Graph databases like Neo4j, DEX, and Titan fall into a similar category. Therefore, some graph processing frameworks, *e.g.,* Chronos, Historical Graph Store, and Kineograph have been developed to tackle the challenges of time-evolving graphs. The vast majority of these use some form of graph snapshotting and represent a temporal graph as a sequence of such snapshots. Such discretisation leads to loosing the continuous semantic of temporal data [289]. Also, very frequent and small changes—like it is the case for cyber-physical systems and IoT—lead to a large amount of (redundant) snapshot data. This, in turn, makes correlating data from different snapshots resource consuming, conflicting with the near real-time re-

quirements such systems usually face. Moreover, with the exception of OLAP, none of these solutions allow to explore many hypothetical actions. Existing solutions for hypothetical queries in OLAP environments fail short to explore a large number of hypothetical actions, complex actions, or nested actions. In summary, none of the existing solutions allows to: 1) analyse frequently changing data (data in motion), 2) explore many different hypothetical actions, 3) reason over distributed data in motion, and 4) combine domain knowledge and machine learning at the same time.

Part II

Analysing data in motion and what-if analysis

4

A continuous temporal data model to efficiently analyse data in motion

This chapter introduces a novel temporal data model and storage system, together with a time-relative navigation semantic to tackle the challenge of efficiently analysing data in motion. This is essential since reasoning processes typically need to analyse and compare the current context with its history. Yet, existing approaches fail to provide sustainable mechanisms to efficiently support the notion of time.

This chapter is based on the work that has been presented in the following papers:

- Thomas Hartmann, François Fouquet, Grégory Nain, Brice Morin, Jacques Klein, Olivier Barais, and Yves Le Traon. A native versioning concept to support historized models at runtime. In *Model-Driven Engineering Languages and Systems - 17th International Conference, MODELS 2014, Valencia, Spain, September 28 - October 3, 2014. Proceedings*, pages 252–268, 2014
- Thomas Hartmann, François Fouquet, Grégory Nain, Brice Morin, Jacques Klein, and Yves Le Traon. Model-based time-distorted contexts for efficient temporal reasoning. In *The 26th International Conference on Software Engineering and Knowledge Engineering, SEKE 2014, Vancouver, BC, Canada, July 1-3, 2014.*, pages 746–747, 2014 (best paper award)

Contents

4.1 Introduction

Cyber-physical systems need to continuously analyse their context in order to adapt themselves to varying conditions. Therefore, building appropriate context representations, able to reflect the current context, *i.e.,* the internal state and environment, is of key importance. This task is not trivial [261] and different approaches and languages are currently used to build such context representations, *e.g.,* ontologies [226] or DSLs [181], for different purposes. In the domain of model-driven engineering, the paradigm of models@run.time [246], [88] has proven its suitability to represent and reason about the context of cyber-physical systems. In this thesis we build on—and extend—the concepts of models@run.time to create context models. As CPSs evolve in a dynamic context, reasoning processes need to analyse and compare the current context with its history.

Let us consider the smart grid example. Due to changes in the production/consumption chain over time, or to the sporadic availability of natural resources (heavy rain or wind), the properties of the smart grid must be continuously monitored and adapted to regulate the electric load in order to positively impact costs and/or eco-friendliness. For instance, predicting the electric load for a particular region requires a good understanding of the past electricity production and consumption in this region, as well as other data coming from the current context (such as current and forecast weather). Since the electrical grid cannot maintain an overload for more than a few seconds or minutes [108], it is important that protection mechanisms work in this time range (near real-time). Therefore, reasoning processes need to be able to efficiently analyse temporal, *i.e.,* continuously evolving, data. Such data is also called *dynamic or data in motion* (as opposed to traditional data at rest) [186].

However, models@run.time and model-driven engineering in general lack native mechanisms to efficiently support the notion of time. In fact, models, as abstractions of the context of a real system, are usually only able to reflect a snapshot of a real system at one specific timestamp. Therefore, they are not able to represent continuously evolving contexts. It is a common approach for such systems to regularly sample and store the context of a system at a very high rate in order to provide reasoning algorithms with historical data. Figure 4.1 shows a context—represented as an object graph—sampled at three different timestamps, t_i, t_{i+1}, and t_{i+2}. As discussed in Chapter 2, models@run.time can be thought of as object graphs, where every object (*i.e.,* node in the graph) corresponds to one model element of the runtime model. Respectively, every edge in the graph corresponds to one relationship. Model elements of the runtime model conform to their respective meta classes. It is important to note that in this context the terms model element, context element, and object are all refering to the runtime model of the context and are therefore used interchangeably. Each graph in the figure represents the context at a given point in time, where all context variables, independently from their actual values, belong to the same time. This is represented in the figure by placing the graphs in horizontal planes in time. The evolution of the system in time is then represented as a stack of context snapshots, where every snapshot represents the object graph (context) at one point in time.

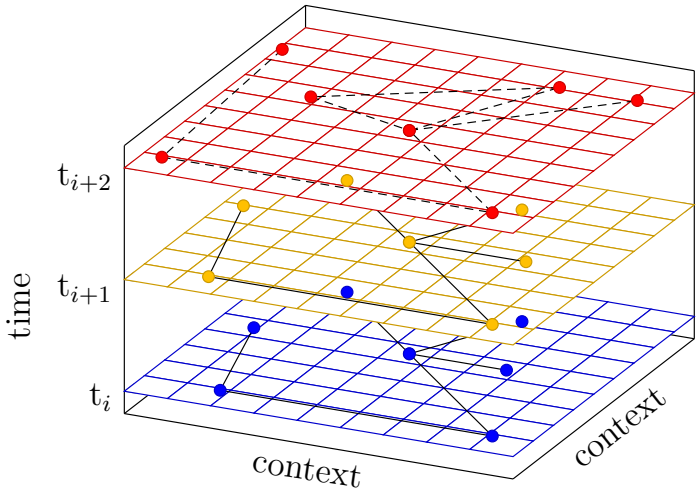


Figure 4.1: Linear sampled context

This systematic, regular context sampling, however, yields to a vast amount of data and redundancy, which is very difficult to analyse and process efficiently. Moreover, it is usually not sufficient to consider and reason just with data from one timestamp, *e.g.,* t_i or t_{i+1}. Instead, for many reasoning processes, *e.g.,* to investigate a potential causality between two phenomena, it is necessary to simultaneously consider and correlate data from different timestamps (*e.g.,* t_i and t_{i+1}). Reasoning processes, therefore, need to mine a huge amount of data, extract a relevant view (containing elements from different snapshots), and analyse this view. This overall process requires some heavy resources and/or is time-consuming, conflicting with the near real-time response time requirements such systems usually need to meet. Going back to the smart grid example: In order to predict the electric load for a region, a linear regression of the average electric load values of the meters in this region, over a certain period of time, can be computed. Therefore, reasoning processes would need to mine all context snapshots in this time period, extract the relevant meters and electric load values, and then compute the load value.

Because of this limitation, models fail in providing suitable and sustainable abstractions to deal with domains relying on temporal reasoning. An efficient temporal data model would *1)* anchor time as an integral part of a data model and its elements, *2)* provide a semantic to efficiently navigate in space (*i.e.,* in the object graph representing the context) and in time, *3)* offer a storage mechanism for efficiently storing temporal data.

It is important to keep in mind, as discussed in Chapter 2 of this thesis, that time series are not abel to represent complex context models. Time series are only able to represent "flat" timestamped values but not complex interconnected models, where every model element and every relationship between model elements can evolve independently in time. In addition, time series usually assume that values evolve following fixed time intervals, instead of on-demand. Moreover, time series databases mostly offer only limited capabilities to analyse this data.

In this chapter we propose a temporal data model as an extension of models@run.time. More specifically, this data model provides a native notion of time to enable efficient

reasoning over temporal data. A novel navigation semantic and concept allows to efficiently navigate the temporal model and allows to consider it as continuous, instead of as a stack of snapshots. Finally, an innovative storage concept allows to only store changed elements instead of full snapshots.

The remainder of this chapter is as follows. Section 4.2 describes the concepts of our approach and shows how we suggest to inject time as a first-class property into data models. Then, Section 4.3 defines a temporal validity for context elements. Based on this, Section 4.4 presents a novel temporal navigation semantic. It defines operations to navigate in the time dimension of a context element and a time-relative navigation concept for traversing the relationships within a temporal context model. Section 4.5 presents an efficient storage system for temporal data. Implementation details and a concrete API example is provided in Section 4.6. We evaluate our approach in Section 4.7 on a concrete smart grid reasoning engine for electric load prediction. The conclusion of the chapter is presented in Section 4.8.

4.2 Time as a first-class property

In a first step towards a temporal data model, we suggest to consider time as a first-class property, crosscutting any context element, *i.e.,* every object of the graph. This allows each element to evolve independently in time. To do so, we associate every element of the context model with a timestamp. In this way, the semantically same object can exist at different timestamps with different values, *i.e.,* in different *versions*. While every object in the graph can be uniquely defined by a single identifier, to uniquely define an object and its version, we need an additional identifier. For this purpose, we define—in addition to an object identifier—a timestamp. It is important to note that this timestamp reflects the domain time rather than the system time. In the context of this work, timestamps are used to distinguish different versions of an object in terms of domain evolution over time. For now, these timestamps are not used in order to create a partial ordering of events in distributed systems, like for example proposed by Lamport [216]. A runtime model can then be composed of objects from different timestamps. Physics, and especially the study of laser [307], discusses a *time distortion* [183] property, specifying that the current time is different depending on the point of observation. Applied to our case, this means that context elements can have different values, depending on the origin of the navigation context, *i.e.,* depending on the timestamp of the inquiring actor. For this reason, we refer to such contexts as *time-distorted contexts*, or simply *temporal contexts*. Figure 4.2 shows such a time-distorted context. The context is again represented as an object graph. Here, the context variables—again independently from their actual values—belong to different timestamps. Such a context can no longer be represented as a graph lying entirely in one horizontal plane (in time). Instead, graphs representing temporal contexts lie in a curved plane. These can be considered as specialised *views*, dedicated for a specific reasoning task, composing navigable contexts to reach elements from different points in time. In contrast to the usage of the term view in database communities we do not aggregate data but offer a way to efficiently traverse specific time windows of a

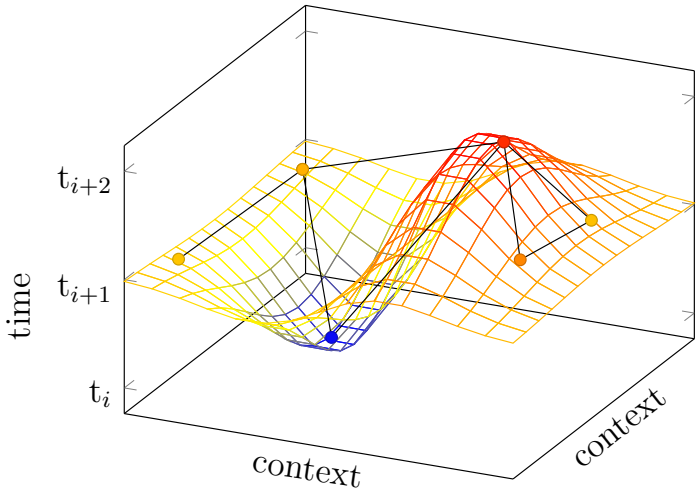


Figure 4.2: Time-distorted/temporal context

context. It is important to note that navigating from one node in the object graph to another one is no longer uniquely defined. Instead, different values could be resolved depending on the time.

We claim that such temporal context representations, which weave time directly into the data model itself, can efficiently empower temporal analytic processes and can outperform traditional full sampling approaches. The contribution of this chapter is to consider temporal information as a first-class property, crosscutting any context element, allowing to organise context representations as temporal views dedicated for reasoning processes, rather than a mere stack of snapshots. We argue that this approach can enable many analytic processes to react in near real-time.

Our hypothesis is that temporal knowledge is part of a domain itself and that defining and navigating temporal data directly within domain contexts is far more efficient and convenient than regularly sampling a context and independently querying each object with the appropriate time. Therefore, we provide a concept to navigate into the time dimension of contexts. Most importantly, we enable a context representation to seamlessly combine elements from different points in time, forming a time-distorted context, which is especially useful for time related reasoning.

4.3 Continuous validity of model elements

Instead of relying on context snapshots, we define a context model as a continuous temporal structure. Nevertheless, each context element (object) of this structure can evolve independently. Since every change occurs at a specific timestamp, *e.g.,* data measured from sensors at regular intervals, every data model is defined only at discrete timepoints. Considering the fact that every context element can evolve independently and at different paces, analytic processes need to find the "correct" time point in order to extract the correct version of an element.

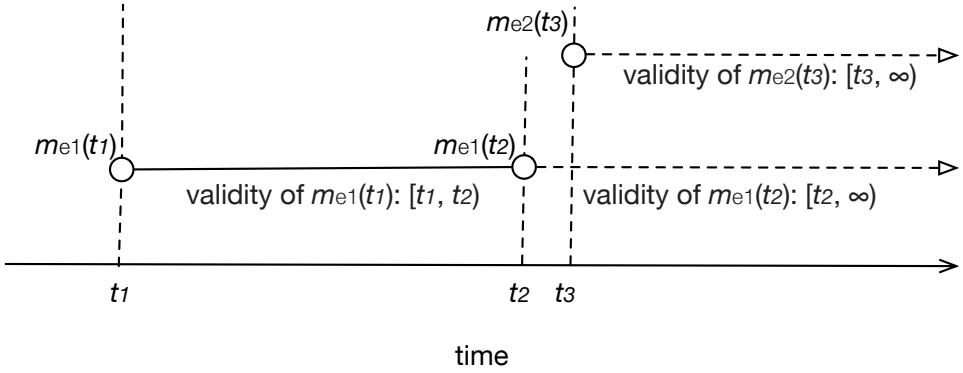


Figure 4.3: Continuous validity of model elements

Let us take a concrete example. Smart meters regularly measure and report customers' consumption values. Different smart meters might send their values at different time points. Reasoning processes analysing this data would have to find for every smart meter the time point of the latest measured value, relative to the reasoning time, to be able to extract the correct values. This is not just the case for "flat" values, like consumption data, but also for relationships between context elements.

In order to define a context as a continuous structure, we specify a continuous validity for context elements. Therefore, we first define an implicit *validity* for elements of the context model. As mentioned before, each context element is associated with a timestamp. This timestamp reflects the domain time at which this value become valid. In other words, a timestamp defines a *version* $v_{m_e}(t)$ of a (context) model element m_e at a time t. A version contains the values of all attributes and relationships of a model element at one timepoint. If a model element now evolves, an additional version of the same element is created and associated to a new timestamp: the domain time at which the new version is valid. Timestamps can be compared and thus form a chronological sequence. Therefore, although timestamps are discrete values, they logically define intervals in which a model element can be considered as *valid* with regards to a timestamp.

We define a continuous temporal semantic where a model element is valid from the moment it is captured until a new version is captured. New versions are only created if necessary, *i.e.,* if a context element actually changes. Figure 4.3 shows two context elements, m_{e1} with two versions and m_{e2} with one version, and their corresponding validity periods. As represented in the figure, version $m_{e1}(t_1)$ is valid in interval $[t_1, t_2[$. Since there is no version of model element m_{e1}, which is captured later than t_2, the validity of version $m_{e1}(t_2)$ is the open interval $[t_2, +\infty[$. Accordingly, version $m_{e2}(t_3)$ is valid in $[t_3, +\infty[$.

Although context elements are still sampled at discrete timestamps, a continuous validity allows to represent a context as a continuous structure. This structure provides the foundation for our continuous temporal data model. In fact, it is what allows context elements to evolve independently and at different paces, making the full sampling of a context model unnecessary.

4.4 Navigating in time

A flexible navigation in time requires to *1)* select a specific version of a model element and *2)* a time-relative navigation concept for traversing the temporal data model.

4.4.1 Selecting model element versions

Based on the idea that it is necessary for intelligent systems to consider not only the current context but also historical data to correlate or investigate potential causalities between two phenomena, we provide means to enable an efficient navigation into time. Therefore, we define three basic operations, which can be called on each model element:

- The *shift* operation is the main possibility to navigate a model element through time. It takes a timestamp as parameter, looks for the context element version of itself, which is valid at the required timestamp, loads the corresponding version of the element from storage (can be main memory) and returns the loaded version. Due to the definition of a continuous validity of model elements, the shift operation always returns a valid version, *i.e.,* the last version relative to the timestamp for which the shift operation was called.
- The *previous* operation is a shortcut to retrieve the direct predecessor (in terms of time) of the current context element.
- The *next* method is similar to the *previous* operation but retrieves the direct successor of the current context element.

These operations allow to shift context elements, independently from each other, through time.

4.4.2 Time-relative navigation

Traversing an object graph composed of elements from different timestamps is complex, since a relationship r from an element m_{e1} to an element m_{e2} is no longer uniquely defined. Thus, the navigation between context elements cannot rely on relations (in space) only. Instead—depending on the timestamps t_1 and t_2 of m_{e1} and m_{e2}, and depending on the set of versions of m_{e1} and m_{e2}—a relationship r from m_{e1} to m_{e2} can link different versions of m_{e2}. This means, which version of m_{e2} is related to m_{e1} by r depends on the timestamp t of m_{e1}. Processing this time-relative navigation manually is complicated and error-prone. Therefore, we provide a concept to automatically resolve relationships, taking the time aspect into account, while traversing the context model. This time-relative resolution is completely transparent and hidden behind methods to navigate in the graph. Hereby, each context element is resolved relative to the time where the navigation is performed from.

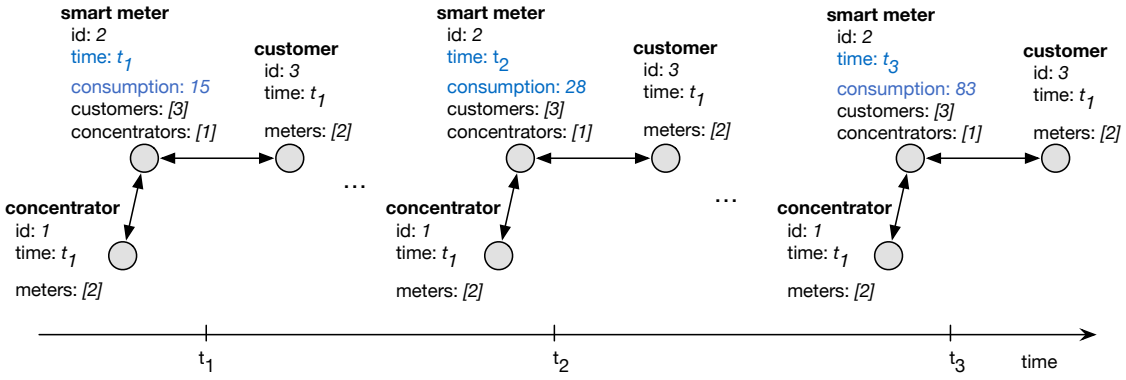


Figure 4.4: Time-evolving context model

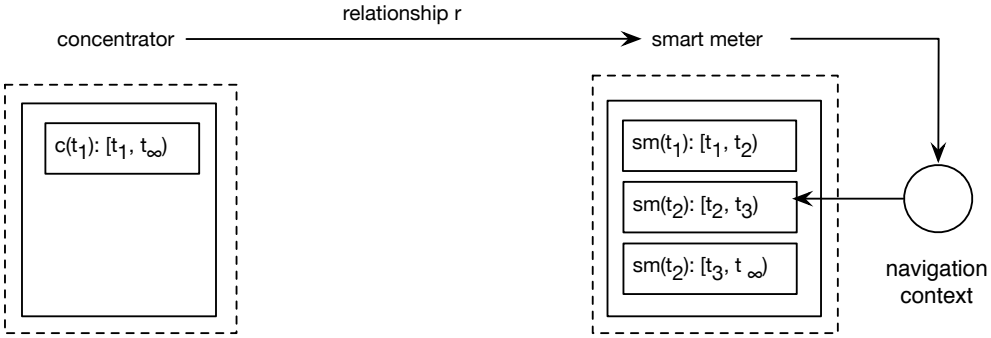


Figure 4.5: Time-relative navigation using a navigation context

Figure 4.4 shows a concrete example of a time evolving context model. The context is again represented as an object graph. As can be seen in the figure, the object graph consists of three different objects (context elements). The *smart meter* element evolves at t_1, t_2, and t_3, all other objects remain unchanged. This leads to one version of the *concentrator* and *customer* object and to three versions of the *smart meter* object. The different versions of the *smart meter* object differ in the *consumption* values and the timestamps. The relationship from the *concentrator* to the *smart meter* object is, therefore, not uniquely defined. Instead, traversing a temporal data model is thus only defined by two dimensions: space and time.

To cope with this problem, we define what we call a *navigation context*. Selecting or navigating context elements is always relative to a navigation context. The navigation context is either set globally before a context model is traversed, or is implicitly set when one of the basic temporal operations is used. If we consider again the example presented in Figure 4.4 and assume that the navigation context is set to a time t_i, where $t_i \geq t_2 \wedge < t_3$, navigating the context model from the *concentrator* to the *smart meter* object would yield the latest valid version of the *smart meter* relative to the navigation context, which is the *smart meter* object with, id: 3, time: t_2, and consumption: 28. This time-relative navigation in context models is depicted in Figure 4.5.

Considering model elements in the context of a specific time interval creates a navigable time dimension for context models. This time-relative data resolution is one of the novel concepts of this contribution. Unlike in previous approaches (*e.g.,* rela-

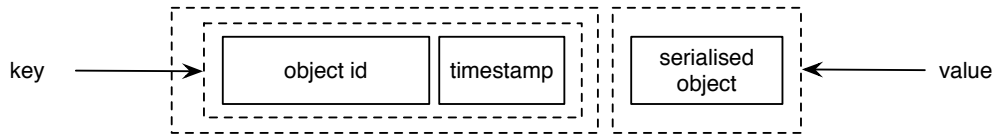


Figure 4.6: Key/value structure for time-relative storage

tionships in MOF [238] or predicates in RDF [217]), the navigation function is not constant but yields different results depending on the navigation context (*i.e.,* the current observation timepoint).

4.5 Storing temporal data

In this section we describe an approach to efficiently store temporal data. Therefore, we rely on two properties: *1)* each context element (object) must be uniquely identifiable, and *2)* it must be possible to get a serialised representation of all attributes and relationships of a context element, with no relativity to a time. To ensure the first property, we define an unique identifier for every context element, *e.g.,* an ongoing number. Since the same semantic object can exist in several versions, *i.e.,* in different states, we need an efficient way to store and load these different versions. Therefore, we define a surrogate key, which consists of an object identifier and a timestamp, to uniquely identify an object version. For the second property, we serialise context elements into *string* representations. Similar to the concept of *traces* [98], [203], each model element can be transformed into a *string* representation and vice versa. The *string* representation of a model element includes all its attribute and relationship information. We use a compressed JSON-like format for the serialisation. Listing 1 shows the *smart meter* object from Figure 4.4 in version $[t_2, t_3)$ in its serialised form. This makes it possible to organise temporal data in a key/value manner. As key, we

Listing 1 Serialised version of a *smart meter* model element

```
{
  "metaclass": "lu.uni.snt.reason.SmartMeter"
  "id": 2,
  "time": "t2",
  "consumption": 28
  "customers": "[3]",
  "concentrators": "[1]"
}
```

use the identifier of an object together with a timestamp. Respectively, as value we use the serialised form of the version of this object. This is depicted in Figure 4.6. This organisation allows us to use technologies eligible for big data to efficiently store and load context data. The data can be stored using different back ends,*e.g.,* key/value stores, or simply in memory (as a cache).

Since data captured in context models usually evolve at a very high pace (milliseconds or seconds), and our approach foresees to not only store the current version of model elements but also historical ones, context models can quickly become very large. In such cases, context models may no longer fit completely into memory, or at least it is no longer practical to do so. Therefore, based on our storage concept, we implement a *lazy loading* mechanism to enable efficient loading of big context models. Elements of the runtime model are dynamically (on-demand) loaded into main memory when the object graph is traversed. Therefore, our implementation allows to manage context models of arbitrary sizes efficiently and it hides the complexity of resolving—and navigating—temporal data.

4.6 Implementation details and API

This section describes how we integrate our temporal modelling approach into the open source modelling framework KMF and show an API example of how temporal data can be manipulated within this implementation. In order to implement a temporal data model in KMF, we first adapt the handling of unique identifiers. Therefore, we extend KMF's generator to generate for every created object an unique object identifier, simply called *id*. As id's we use an ongoing number of *long*s. The id is a technical identifier and independent of any business identifier. Additionally, we extend the KMF generator to automatically generate a timestamp attribute for all context elements and an associated public *getter* method to read the timestamp value. Next, we generate a *key* method, which returns a concatenation of id and timestamp in the from of *[id,timestamp]*. This key uniquely identifies a context element version and is used for storing and loading context elements in key/value stores. For every context element we generate a generic method to serialise and unserialise the object into a *string* representation and vice versa. We provide an expandable datastore interface and several implementations for different key/value stores. Currently we provide implementations for Google's LevelDB [42], Facebook's RocksDB [57], Redis [54], and mongoDB [45]. Finally, the navigation context is implemented as a special object given to the factory of the generated API. The navigation context object determines which version should be resolved while the model is traversed.

Modelling approaches usually use meta model definitions (*i.e.,* concept descriptions) to generate domain specific APIs. The following example illustrates our API. In addition to a classical modelling API, our implementation provides functions for creating, deleting, storing, loading, and shifting versions of context elements. Applications can use this API to create new context elements, specify their timestamps, store them, change their attributes and relationships, and store new versions (with different timestamps). In addition, the API can be used to specify the *context time* on which elements should be resolved while traversing the model. One can imagine the definition of the context time as the curve shown in Figure 4.2. Listing 2 shows Java code that uses a *Context ctx* (an abstraction to manipulate model elements) to perform these actions. The API provides a seamless way to create, manipulate, and navigate in the temporal data model.

Listing 2 Usage of the temporal data model API

```
//creating and manipulating model elements
ctx.setTime("2014/3/1"); //sets the context time

SmartMeter m1 = ctx.createSmartMeter("m1"); (//creates a new smart meter
//add a new consumption object
m1.setConsumption(ctx.createConsumption(125.6));

SmartMeter m2 = ctx.load("m2"); //retrieves smart meter m2 at the context time
m2.setConsumption(ctx.createConsumption(77.12)); //sets new value
m2.setRegisteredBy(m1); //sets the registerBy relation

ctx.setTime("2014/3/2"); (//sets the context time
SmartMeter m1_2 = m1.shift("2014/3/2"); //shifts smart meter m1 in time
m1_2.setConsumption(ctx.createConsumption(193.7));


// definition of the context time
ctx.setTime("2014/3/3"); //sets the context time
SmartMeter r_m1 = ctx.load("m1"); //loads the smart meter at the context time
assert(r_m1.getConsumption()==193.7); //latest value relative to context time

SmartMeter r_m2 = r_m1.getRegisteredEntities().get(0);
assert(r_m2.getConsumption()==77.12);
assert(r_m2.getTime()=="2014/3/3"); //retrieves time where it is resolved to
```

4.7 Evaluation

In this section, we evaluate if the proposed temporal data model is able to efficiently analyse data in motion. Therefore, we apply it on an industrial case study and evaluate its impact. The case study is taken from our cooperation with Creos Luxembourg S.A. and has initially led to the research behind this approach. In a nutshell, in this case study we evaluate the performance of a reasoning engine that needs to analyse temporal smart grid data. Therefore, it has to aggregate and navigate temporal data and, if necessary, take corrective actions. This case study is based on the smart grid model presented in Figure 1.4, which is periodically filled with live data from smart meters and sensors. Based on the electric consumption, smart meters can derive the electric load in a region. The idea for this reasoning engine is to predict, if the load in a certain region will likely exceed or surpass a critical value. Therefore, a linear regression of the values of the meters in this region, over a certain period of time, has to be computed.

This case study has been implemented twice, once with a traditional sampling strategy, and once using our temporal data model, which we implemented into the KMF framework (*cf.* Section 4.6). The full sampling approach and our approach both use Google's LevelDB as a storage backend and both are executed using JDK 8. All experiments are conducted on a MacBook Pro with an Intel Core i7 CPU, 16GB RAM, and a SSD. Each experiment has been executed 100 times and the presented results are average values.

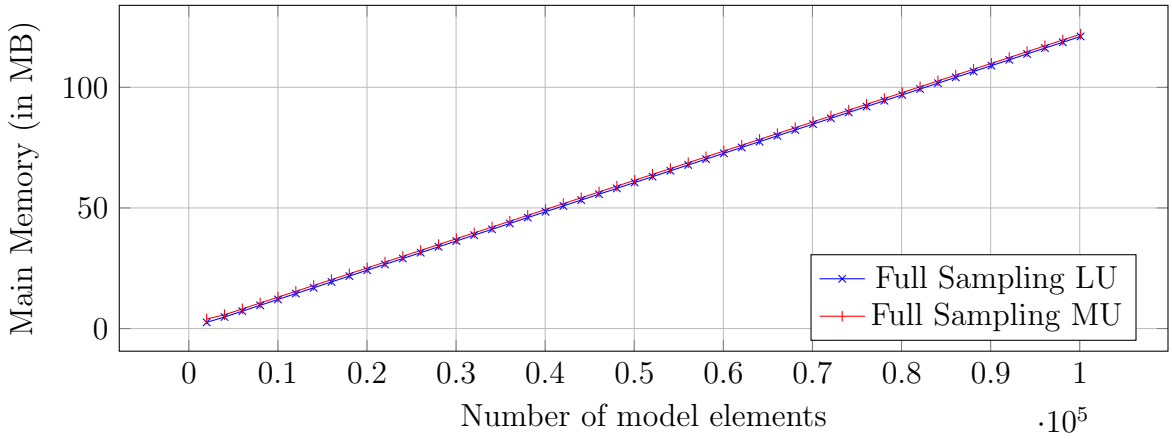


Figure 4.7: Memory usage for model update operations using the full sampling strategy

The following validation is based on three key performance indicators (KPIs): *1)* time and memory requirements to update the context model, *2)* performance to navigate the context model in time, and *3)* space requirements for persisting the temporal data. For each KPI, we compare our approach with the classic sampling strategy, taking a snapshot of the entire model for each modification (or periodically). The measured memory value for KPI-1 is main memory (RAM), for KPI-3 it is disk space. The measured time is the time required to complete the reasoning process (depending on the KPI). Main memory is measured in terms of used heap memory, queried using Java's runtime API.

4.7.1 KPI-1: Model updates

First, we evaluate time and memory requirements to update the proposed temporal data model and compare this to a full sampling approach. We analyse modifications of two magnitudes: *1)* a large update (LU) that consists in creating a new concentrator and a smart meter subtree (1,000 units) and *2)* a minor update (MU) that consists in updating the consumption value of a specific smart meter, which is already present in the context model. For this experiment, we keep the size of each update constant but vary the size of the context model and the history. We grow the context model from 0 to 100,000 elements, which approximately corresponds to the dimension of the actual size of our Luxembourg smart grid model. The results of KPI-1, in terms of memory usage, are depicted in Figure 4.7, for using the full sampling approach and in Figure 4.8, for using the temporal data model. Outcomes of KPI-1, with respect to the required time for updating the context models are shown in Figure 4.9, for full sampling and in Figure 4.10, for the temporal data model.

Let us first consider **main memory**. The full sampling strategy depends on the size of the model, as reflected by the linear progression of the required main memory size, to perform the updates. In contrary, our approach results in two flat curves for LU

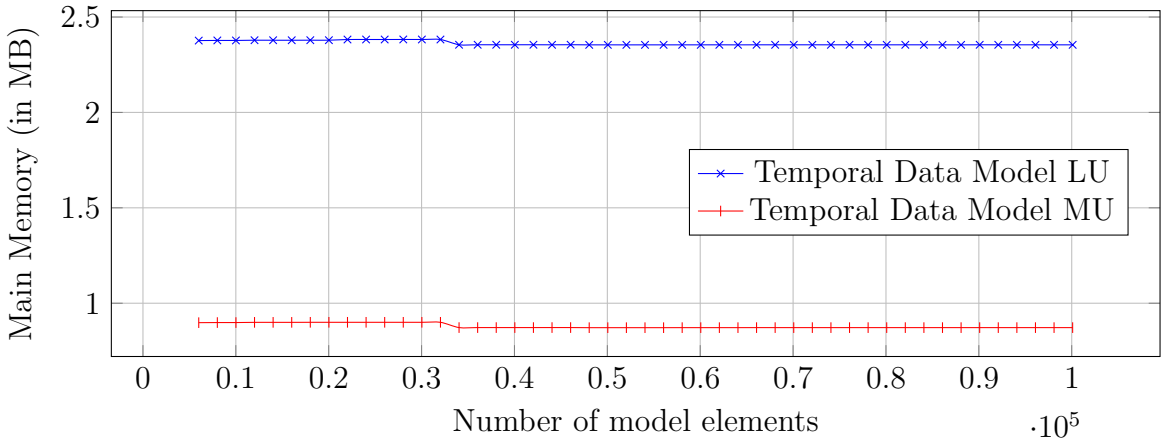


Figure 4.8: Memory usage for model update operations using the temporal data model

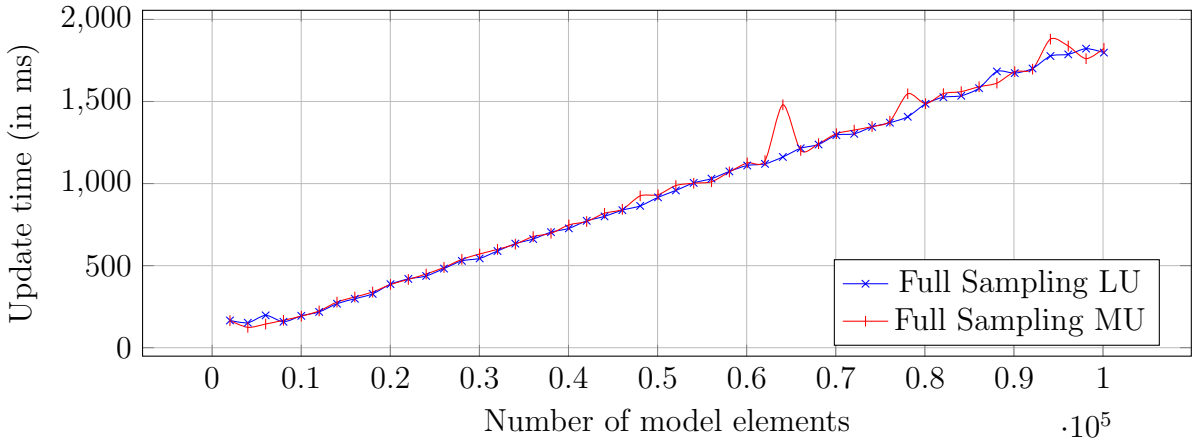


Figure 4.9: Update time for model manipulations using the full sampling strategy

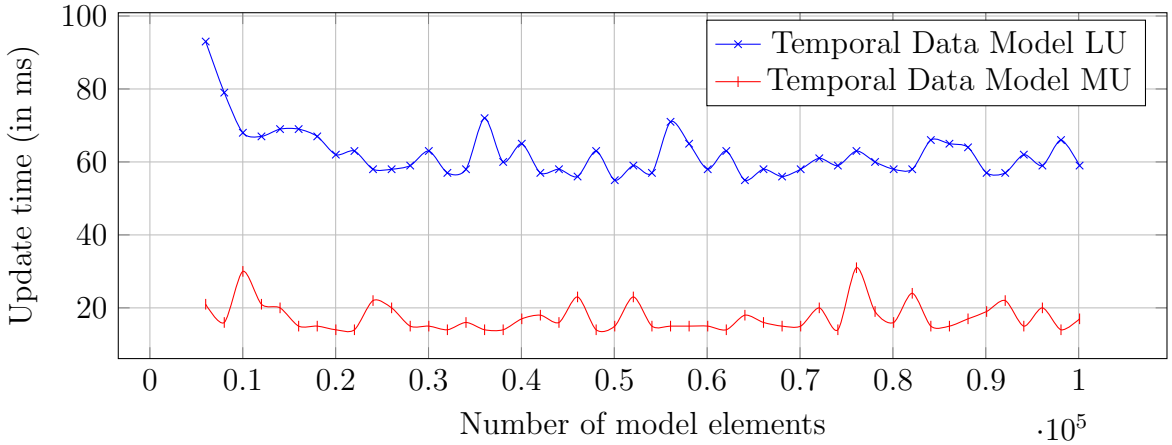


Figure 4.10: Update time for model manipulations using the temporal data model

and MU updates, showing that the required memory only depends on the size of the update, not on the size of the model. This is confirmed by the fact that LU requires more memory than MU, but both are constant—less than 2.5 MB, compared to up to 100 MB of the full sampling strategy. This is due to our lazy loading approach, *i.e.,* only the elements which need to be updated are loaded into main memory.

Next, we look at the **time** required to update context models. The time to insert new elements using the full sampling approach depends on the size of the model, but is nearly constant with the proposed temporal data model. This behaviour is similar to what we observed for the required main memory. The fact that the updated time is less for MU compared to LU confirms that our approach reduces the time needed to modify elements. Looking at the results of the experiments, KPI-1 demonstrates that even in the worst case scenario, where all elements evolve at the same pace, our approach offers a major improvement for model update operations (factor of 33 for time and between 50 to 100 for memory).

Finally, we analyse the capability of our temporal data model to handle **batch insertions**. Therefore, we additionally performed a batch insert using once the full sampling and once our approach. The batch insert consists of 10,000 historical values for each smart meter, resulting in a model of 1 million elements. As a result, we obtain **267** seconds to insert with the full sampling strategy and **16** seconds for our approach. This means that even in the worst case, we still have an improvement of a factor of **17** for the insertion time.

4.7.2 KPI-2: Navigating the context model in time

For the following experiment, we consider an already existing smart grid model containing, a history of consumption values. We evaluate the required time to execute a complex computation over the historical consumption data. We run several prediction algorithms over the model, which correlate historical data in order to predict the future state of the grid and, for example, throw an alert in case of a potential overload risk.

We define two prediction categories, each for two different scales, resulting in 4 different reasoning processes: *1)* small deep prediction (SDP), *2)* small wide prediction (SWP), *3)* large deep prediction (LDP), and *4)* large wide prediction (LWP). Wide prediction means that the algorithm uses a correlation of data from neighbouring smart meters in order to predict the future consumption. This means that the algorithm needs to explore, *i.e.,* navigate, the model in wide. The underlying idea is that the electric consumption within a region (a number of geographically close smart meters) remains comparable over time for similar contexts (weather conditions, time of the year, etc.). The deep prediction strategy uses the history of customers to predict their consumption habits. In this case, the algorithm needs to navigate the model in deep, *i.e.,* it needs to navigate the history of a model element. For both approaches we perform a linear regression to predict the future consumption using two scales: large (100 meters) and small (10 meters).

Table 4.1: Reasoning time to predict the electric consumption (in milliseconds)

Type	SDP	SWP	LDP	LWP
Full	1,147.75 ms	1,131.13 ms	192,271.19 ms	188,985.69 ms
Lazy	2.06 ms	0.85 ms	189.03 ms	160.03 ms
Factor	557	1,330	1,017	1,180

The results are presented in Table 4.1. The gain factor of using the temporal data model, compared to full sampling, is defined as $Factor = (Full\ Sampling\ time\ /\ Native\ Versioning\ time)$. As can be seen in Table 4.1, the gain factor lies between **557** and **1,330**. Using the proposed temporal data model instead of a full sampling approach, reduces the processing time from minutes to seconds. This experiment showed that the usage of a temporal data model can significantly reduce the time to analyse historical data. This can enable reasoning processes to react in near real-time.

4.7.3 KPI-3: Storing temporal data

In this section, we evaluate the overhead, introduced by our approach, for storing temporal data. As in the experiments before, we compare the results with a full sampling strategy. Our goal is to determine, how much of a model must be changed in one step, so that storing temporal models is more costly in case of disc space, compared to a full sampling approach. Intuitively, the more changes are done, the higher the overhead will be compared to full sampling. In other words, we want to investigate, after which percentage of modifications becomes our solution less efficient in terms of storage space, compared to the full sampling approach. It is important to note that the navigation gains remain still valid.

For this evaluation, we load an existing model (containing 100 smart meters), update the consumption value of several meters, serialise it again and store it. By varying the percentage of smart meters updated per version (period), we can compare the size of the storage space, which is required for the comparison with our approach and the full sampling approach. To ensure a fair comparison we use for both cases a compact JSON serialisation format. Results are depicted in Figure 4.11.

Regardless of the amount of modifications, the full sampling approach requires 39.1 KB to store one version (snapshot) of the model. This is a serious overhead for small modifications. In contrary, the temporal data model requires a variable amount of storage space, *i.e.,* the required storage space depends on the amount of modifications. It varies from 393 bytes for 1% of changes to 39.8 KB for 100% of changes (the complete model changes). A linear augmentation of model changes leads to a linear augmentation of needed storage space. This confirms that our storage strategy for model elements has no unexpected side effect.

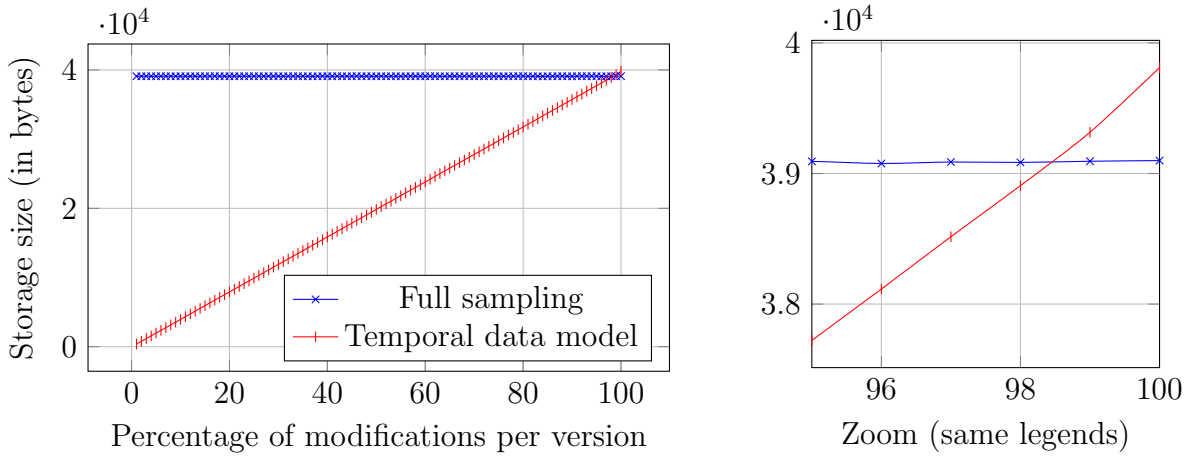


Figure 4.11: Required storage space to save temporal data

To put the observed results into perspective, our proposed temporal data model reduces the required storage space by **99.5**% for 1% of changes. On the other hand, it increases the required storage space by 1.02% for 100% of modifications. This means that up to **98.5%** of modifications of a model, our approach needs less memory than a full sampling approach. Also, the overhead of 1.02% for a change of the full model has to be set into relation to the features enabled by this overhead (navigation, insertion time gains, comparison time gains).

Besides the presented runtime usage improvements, this validation shows that the temporal data model offers nearly constant time and memory behaviour, which allows to face massive amounts of historical data and large-scale context models. This validation demonstrates that the proposed temporal data model is able to efficiently analyse data in motion.

4.8 Conclusion

Modelling approaches, such as models@run.time, provide semantically rich reflection layers, which enable cyber-physical systems to reason about their context. As these systems evolve over time, reasoning processes typically need to analyse and compare the current context with its history. The use of models to organise and store such dynamic data—also called data in motion—suffers from the lack of sustainable mechanisms to efficiently handle historical data. Despite the fact that considering time as a crosscutting concern of data modelling has been discussed since quite some time, today's modelling approaches mostly still rely on a discrete representation of time. Therefore, a common approach consists in a temporal discretisation, which regularly samples the context (snapshots) at specific timestamps to keep track of the history. Analysing these data would then require to mine a huge amount of snapshots, extract a relevant view, and finally analyse it. This would require lots of computational power and be time-consuming, conflicting with the near real-time response time requirements these systems usually face. In this chapter, we presented a novel temporal data model, which considers time as a first-class property crosscutting any model element,

allowing to organise context representations as temporal views dedicated for reasoning processes, rather than a mere stack of snapshots. By introducing a temporal validity, independently for each model element, we allowed each model element to evolve independently and at different paces, making the full sampling of a context model unnecessary. Finally, we added a time-relative navigation, which makes an efficient navigation between model elements, coming from different timestamps, possible. This allows us to assemble a temporal data model for reasoning purposes and seamlessly and efficiently navigate along the time dimension of data, without the need to manually mine the necessary data from different context models. The proposed temporal data model has been implemented and integrated into the open source modelling framework KMF and evaluated on a smart grid reasoning engine for electric load prediction. We showed that our approach supports temporal reasoning processes, outperforms a full context sampling by far, and can be compatible with near real-time requirements. To sum up, we demonstrated that the proposed temporal data model is able to efficiently analyse data in motion.

5

A multi-dimensional graph data model to support what-if analysis

Over the last few years, the cross-fertilisation of big data and cyber-physical systems*, respectively, the* Internet of Things *has boosted data analytics from a descriptive era, mostly confined to the explanation of past events, to the emergence of new predictive techniques. Nevertheless, existing predictive techniques still fail to envision alternative futures, which inevitably diverge when exploring the impact of* what-if *decisions. What-if analysis calls for the design of scalable data models that can cope with the complexity and the diversity of representing and exploring many different alternatives. This chapter introduces a multi-dimensional graph data model, called many-world graph, which combines multi-dimensional graphs and temporal data to organise a massive amount of unstructured and continuously changing data. The proposed data model is an extension of the temporal data model presented in the previous chapter.*

This chapter is based on the work that has been presented in the following paper:

- under submission at ACM/USENIX EuroSys 2017: Thomas Hartmann, Assaad Moawad, Francois Fouquet, Gregory Nain, Romain Rouvoy, Yves Le Traon, and Jacques Klein. PIXEL: A Graph Storage to Support Large Scale What-If Analysis

Contents

5.1 Introduction

In their "2013 Hype Cycle of Emerging Technologies" report Gartner considers prescriptive analytics as one of the "innovation triggers" of the next five to ten years [1]. For instance, the emerging domains of *cyber-physical systems* and the *Internet of Things* are expected to increasingly control bigger and bigger parts of our critical infrastructures, like electric grids, (semi-)autonomously [194]. This requires advanced data analytics to turn the huge amount of collected data into valuable insights to identify suitable decisions [277]. However, technologies for prescriptive analytics are yet in their infancies. It heavily relies on the exploration of what might happen if this or that action would be applied, which is referred to as *what-if analysis* [167]. What-if analysis therefore plays a crucial part of decision-making.

Every action induces some side-effects, which potentially lead to an alternative state from where a set of other actions can be applied and so forth. When considering complex systems, such as CPSs or IoT, hundreds or thousands of alternative actions must be explored simultaneously. As in the many-world interpretation [139], every action can be interpreted as a divergent point leading to an alternative, independent world. This means that every data variable can have alternative values in different worlds. What-if analysis therefore tries to establish the sequence of actions that leads to the desired values of all variables, *i.e.,* the desired world.

In addition, actions and values have a temporal dimension. As discussed in detail in Chapter 4, it is usually not enough to consider just the current state of a system, but it is often necessary to also consider and reason about historical data, using for instance approaches like sliding window analytics [93]. Therefore, given a specific world, variables can have different values for different points in time. This can lead to different histories for the values of variables in different worlds. In addition to that, every world and their variables can evolve independently at different paces. This leads to a huge combinatorial complexity of world and timepoint alternatives. Therefore, what-if analysis requires to define a data model that can represent at the same time:

- **Temporal—*i.e.,* evolving—data**: Most of nowadays data is temporal in nature: from social networks, financial transactions, medical records to self-driving cars. What-if analysis typically not only needs to process current, but also historical data (*cf.* Chapter 4).
- **Several alternative worlds**: To independently explore different actions, it is necessary to "fork" or "snapshot" the underlying data, so that every action can be simulated on its own dataset.

The fast-growing area of graph analytics (for example GraphX [323]) suggests to organise the massive amounts of unstructured, constantly changing data which such analytics have to deal with, as graphs. Graphs and associated computation models have been demonstrated to be especially suitable to depict complex data and their relationships [232], [240]. An increasing number of work discuss challenges of temporal aspects of graph data [82], [107], [201], however an efficient exploration of many

independently evolving worlds remains an open issue. As discussed in Chapter 2, models@run.time can be thought of as object graphs, where every node corresponds to one model element of the runtime model. Respectively, every edge in the graph maps to a relationship of the runtime model. In this chapter, we follow the terminology of graph analytics and speak about nodes and edges rather than model elements and relationships.

To address the combinatorial complexity of world and timepoint alternatives, we propose in this chapter a novel graph data model, called many-world graph (MWG), where values of each node are resolved on-demand, based on the viewpoint (defined by a world and a timepoint) where we read from. As in the famous example of Schrödinger's cat [282], where the cat is "dead" and "alive" at the same time (in different worlds) and the actual state is just revealed at the moment the cat is observed, in our approach nodes and edges can have many different values at the same time (depending on the world and time), which are just revealed at the moment the graph is observed. Like in this example, every node can have alternative values depending on the current viewpoint (time and world). Let us now suppose we could influence the state of the cat. The goal would then be, if we want to save the cat, to select the sequence of actions that leads to the world where the graph represents the state where the cat is considered as alive. Based on this concept, our MWG implements an efficient on-demand fork concept for nodes and edges and at the same time supports temporal nodes and edges. We show that this allows to efficiently explore a large number of independent actions—in time and many worlds—even on a massive amount of data (hundreds of millions of nodes, timepoints, and hundreds of thousands of worlds). We believe that this model can prepare the ground for efficient what-if analysis.

We integrated this data model into the Kevoree Modeling Framework[1], to evaluate its capabilities and limits. First, we evaluate its performance when used as a base graph storage. We compare our approach with a state of the art graph storage. Secondly, we focus on evaluating the temporal aspects of our approach. Besides raw performance testing for different scenarios, we compare our approach to a state of the art time series database. Thirdly, we evaluate the performance of inserting and reading from many different worlds for different scenarios. Finally, we validate our approach with a scenario from the smart grid case study. For all cases, we discuss results, limits, best and worst cases.

The remainder of this chapter is organised as follows. First, Section 5.2 motivates the research behind this contribution, based on the smart grid case study. Sections 5.3 and 5.4 introduces the main concepts of MWG and their implementation in KMF. We thoroughly evaluate our approach in Section 5.5. The chapter concludes in Section 5.6.

[1]The source code of our many-world graph implementation is available under https://github.com/kevoree-modeling/mwDB

5.2 Motivating example

Existing solutions supporting prescriptive analytics, and more specifically *what-if* analysis, are poorly addressing the challenges of scalability. To illustrate this issue, we consider the smart grid case study, which builds on *prescriptive analytics* to take advanced and (semi-)autonomous decisions to adjust the load and the topology of the smart grid dynamically. Prescriptive analytics aims to explore several candidate actions to answer the question *"what should we do?"*, in respect to a given goal (*cf.* Chapter 2). Beyond statistical forecasting, *prescriptive analytics* requires to explore what happens if this or that action would be applied, *i.e.,* to navigate through different alternative scenarios. In this chapter, we focus more specifically on *what-if* analysis, which is an essential and challenging part of prescriptive analytics.

Intelligent load management is a major concern for electricity utility companies [142]. In particular, they are expected to avoid potential overload situations in electricity cables by appropriately balancing the load. The electric load in cables depends on the consumption of customers connected to a given cable, on the topology (*i.e.,* how cables are connected to each other), and on *power substations*. A topology can be changed by opening/closing so-called *fuses* in cabinets. This results in connecting/disconnecting the cables that connect households to different power substations, therefore impacting the electricity flow in the grid. Applying prescriptive analytics to this scenario would mean to simulate the electric load for different hypothetical topologies (what-if scenarios) with the goal to find an "optimal" one—*i.e.,* where the load in all cables is best balanced. Then, the necessary actions leading to this topology can be suggested as a result of the prescriptive analytic process. Smart grids are very large-scale systems, connecting hundreds of thousands or even hundreds of millions of nodes (customers). This makes the simulation of different what-if scenarios very challenging. Moreover, many different topologies are possible, which can easily lead to thousands of different scenarios.

To avoid potential overload situations, alternative topologies need to be explored a priori, *i.e.,* before the problem actually occurs. The computation of the electric load depends, aside from the topology, on the consumption data of customers. In the context of a smart grid, this data is measured by smart meters, which are installed at customers' homes and regularly report to utility companies (*e.g.,* every 15 minutes [179]). One can compute the electric load based on profiles of customers' consumption behaviour. These profiles are built using live machine learning algorithms, as the ones we introduced in [179]. However, the huge amount of consumption data quickly leads to millions of values per customer and efficiently analysing such large historical datasets is challenging. The temporal dimension of data often results in inefficient data querying and iteration operations to find the requested data. While this issue has been extensively discussed by the database community in the 80s and 90s [115], [283], this topic is gaining popularity again with the advent of *time series* databases for IoT and sensor data (*e.g.,* influxDB [39]). Time series can be seen as a special kind of temporal data, which is defined as a sequence of timestamped data points, and is used to store data like ocean tides, stock values, and weather data. It is important to note that in time series, data is "flat"—*i.e.,* , time series' only contain primitive values, like raw measurements. However, they are not able to model complex data structures and their relationships,

like for example the evolution of a smart grid topology. Therefore, time series analysis is not sufficient to support complex what-if analysis and prescriptive analytics. On the other side, graph-based storage solutions (*e.g.,* Neo4j [47]), despite some attempts to represent time dependent graphs [107], [55], [32], are poorly addressing the time dimension in their model. They are either failing to navigate through alternative versions of a given graph, or covering this issue by generating distinct clones of the graph (*cf.* Chapter 3).

These limitations, therefore, motivates our work to support such large-scale what-if analysis for prescriptive analytics. More specifically, we introduce the concept of *many-world graphs* as a scalable data model to explore alternative scenarios in the context of what-if analysis.

5.3 Many-world graphs

In this section we detail a multi-dimensional graph data model, called *many-world graph*. The goal of this data model is to efficiently support large-scale what-if analysis. First, we introduce the key concepts behind many-world graphs. Then, we formalise its semantics, starting by a simple graph model, which we extend in a first step with a temporal dimension and in a second step with a dimension to represent several different alternatives.

5.3.1 Key concepts

This chapter introduces the notion of *many-world graphs*, which are directed graphs whose structure and properties can evolve along time and parallel worlds. In particular, many-world graphs build on the following core concepts:

- **Timepoint**: a discrete event, usually encoded as a timestamp
- **World**: a parallel world (or universe), used as an identifier
- **Node**: a domain-specific concept, which exists across worlds, used as an identifier
- **State**: the value of a node for a given world and timepoint
- **Timeline**: a sequence of states for a given node and a given world

Depending on the considered timepoint (t) and world (w), different states can therefore be resolved from a given node (n), as illustrated in Figure 5.1. States are organised into chunks (c), which can be uniquely mapped from any viewpoint $\langle n, t, w \rangle$.

Therefore, we define a function read, which resolves for a node (n), a timepoint (t), and a world (w) a state chunk (c).

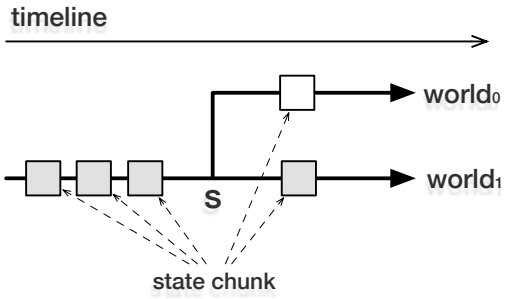


Figure 5.1: State chunks of a node in two worlds

Definition 1 *Function* $read(n, t, w) \mapsto c_t$

We associate each state chunk with a timepoint (c_t) and define a timeline as:

Definition 2 *A timeline* ($t_{n,w} = [c_0, \ldots, c_n]$) *is an ordered sequence of chunks belonging to a given node (n) from a given world (w).*

Alternative state chunks in different worlds, therefore, form alternative timelines. As a consequence, a resolution function (*read*) returns a chunk (c_t) for an input viewpoint as the "closest" state chunk in the timeline.

Therefore, when a many-world graph is navigated, state chunks of every node have to be resolved according to an input world and timepoint. The processing of many-world graphs made of millions of nodes cannot be done in memory, thus requiring to efficiently store and retrieve chunks from a persistent data store. For this purpose, we decompose state chunks into keys and values and we store these values in a key/value storage. The mapping of nodes to state chunks (including attributes and references to other nodes) and their storage is further detailed in Section 5.4.1.

While prescriptive analytics builds on what-if analysis for new worlds along time, there are two techniques that can be employed when forking worlds: *snapshotting* and a *shared past* (cf. Figure 5.2).

Snapshotting consists in copying all state chunks of all timepoints from a *parent world* p to the *child world* w, thus leaving both worlds to evolve completely independently, in the past and in the future. Although this approach is simple, it is obviously very inefficient in terms of time and storage to clone all state chunks of all historical records.

We therefore propose to adopt an alternative approach based on a *shared past*. Let us consider a scenario where a new world w is diverged from a parent p at a point s in time. Before timepoint s, both worlds share the same past and thus resolve the same state chunks. After the divergence timepoint s, world w and p *co-evolve*, which means that each can have their own timeline for $t \geq s$. Therefore, both worlds share the same past before the divergent point (for $t < s$), but each evolves independently after the divergent point for $t \geq s$.

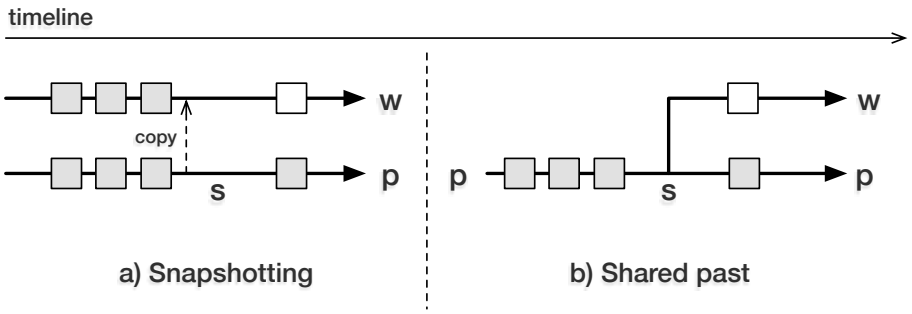


Figure 5.2: Types of many-worlds

5.3.2 Many-world graph semantics

With the *many-world graph* (MWG), we seek to efficiently structure and analyse data that can evolve independently in time and many worlds. In an abstract way, such a graph can be defined as $G = N \times T \times W$, where N is the set of nodes, T the set of timepoints, and W the set of worlds. This would be the equivalent of snapshotting. However, what-if analysis needs to explore many different actions, which usually does not affect all data in all worlds and all timepoints. Therefore, snapshotting would be an extremely inefficient approach to manage such a graph. To address this combinatorial problem of world and timepoint alternatives, we define our MWG in a way so that values of each node are resolved on-demand, based on a world and a timepoint. We avoid snapshotting with a novel concept of *on-demand forks of nodes*. In this section, we formalise the semantics of our MWG by starting with a base graph definition, which we first extend with temporal semantics (time-evolving graph) and then with the many-world semantics.

5.3.3 Base graph (BG)

A graph G is commonly defined as an ordered pair $G = (V, E)$ consisting of a set V of vertices (or nodes) and a set E of edges. In order to distinguish between nodes and their states, we define a different layout. First, we define a node as a conceptual identifier that is mapped to what we definet as a "state chunk". A state chunk contains the values of all attributes and edges which belong to a node. Attributes are typed with one of the following primitive types: `int`, `long`, `double`, `string`, `bool`, or `enumeration`.

Definition 3 *Formally, the state chunk of a node n is: $stateChunk_n = (A_n, R_n)$, where A_n is the set of tuples of names and values of the attributes of this node and R_n is the set of tuples of names and values of relationships from n to other nodes.*

From now on, we refer to edges as directed relationships or simply as relationships. Unlike other graph models (*e.g.,* the one from Neo4j [240]) our model does not support attributes for edges. Besides being simple, this also makes our graph data model similar to the object-oriented one, which today is the dominating data model of many modern

programming languages, like Java, C#, Scala, and Swift. It also enables a seamless integration into KMF.

We introduce the function $read(n)$ that resolves the state chunk of a node n. It returns the state chunk of the node, which contains the relationships or edges to other nodes. We can now define a *base graph (BG)* as:

Definition 4 $BG = \{read(n), \forall n \in N\}$

The main difference to common graph definitions is that our base graph is not statically defined, but is the result of the evaluation of the $read(n)$ function over all nodes n. This means that the graph is dynamically created. Implicitly, all state chunks of all nodes are dynamically resolved and the graph is created by linking the nodes accordingly to the relationships defined within the resolved state chunks.

In this way, only the destination nodes need to be listed in the set, since all the directed edges start from the same node n, thus making it redundant to list the source node. For example, if we have: $stateChunk_n = \{\{att1\}, \{m, p\}\}$, where $m, p \in N$, this means that the node n has one attribute and two relationships (one to node m and another one to node p). Two directed edges can be implicitly constructed: $n \rightarrow m$ and $n \rightarrow p$.

5.3.4 Temporal graph (TG)

In this section, we extend our BG with temporal semantics. Therefore, we extend the function $read(n)$ with a function $read(n, t)$, with $t \in T$. T is a totally ordered sequence of all possible timepoints: $\forall t_i, t_j \in T : t_i \leq t_j \vee t_j \leq t_i$. We also extend the state chunk to its temporal representation:

Definition 5 *$stateChunk_{n,t} = (A_{n,t}, R_{n,t})$, where $A_{n,t}$ and $R_{n,t}$ are the sets of resolved values of attributes and relationships, for the node n at time t.*

Then, we define the *temporal graph (TG)* as follows:

Definition 6 $TG(t) = \{read(n, t), \forall n \in N\}, \forall t \in T.$

Every node of the TG can evolve independently. As timepoints can be compared, they naturally form a chronological order. We define that every state chunk belonging to a node in a TG is associated to a timepoint and can therefore be organized according to this chronological order in a sequence $TP \subseteq T$. We call this ordered sequence of state chunks the *timeline* of a node. The timeline $tl(n)$ of a node n is defined as:

Definition 7 $tl_n = \{stateChunk_{n,t}, \forall t \in TP \subseteq T\}$

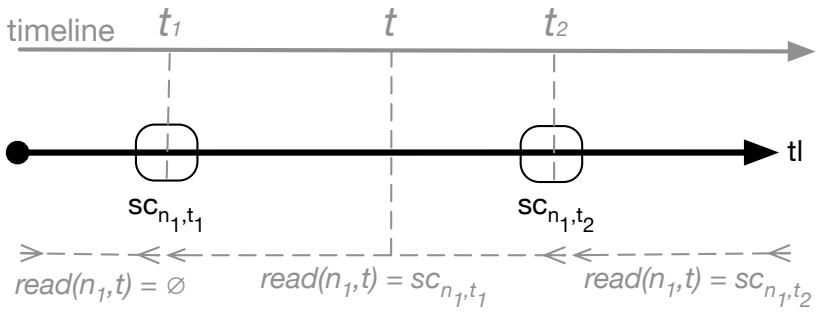


Figure 5.3: TG node timeline

The two basic operations `insert` and `read` are defined as:

Definition 8 *Operation* $insert(stateChunk_{n,t}, n, t)$:
$(StateChunk \times N \times T) \mapsto \{\emptyset\}$, *as the operation that inserts a state chunk in the timeline of a node* n, *such as:*
$tl_n := tl_n \cup \{stateChunk_{n,t}\}$

Definition 9 *Operation* $read(n, t)$:
$(N \times T) \mapsto StateChunk$, *as the operation that retrieves, from the timeline* tl_n, *and up until time* t, *the most recent version of the state chunk of* n, *which was inserted at timepoint* t_i.

$$read(n,t) = \begin{cases} stateChunk_{n,t_i} & if\ (stateChunk_{n,t_i} \in tl_n) \\ & \wedge (t_i \in TP) \wedge (t_i < t) \\ & \wedge (\forall t_j \in TP \rightarrow t_j < t_i) \\ \emptyset & otherwise \end{cases}$$

According to these definitions, although timestamps are discrete, they logically define intervals in which a state chunk can be considered as *valid* within its timeline (*cf.* Chapter 4). Considering the following operations:

$insert(sc_{n_1,t_1}, n_1, t_1)$ and $insert(sc_{n_1,t_2}, n_1, t_2)$, where we are inserting 2 state chunks sc_{n_1,t_1} and sc_{n_1,t_2} for the same node n_1 at two different timepoints with $t_1 < t_2$, we define that sc_{n_1,t_1} is valid in the open interval $[t_1, t_2[$, and sc_{n_1,t_2} is valid in $[t_2, +\infty[$ (as defined in Chapter 4). Therefore, an operation $read(n_1, t)$ resolves $\emptyset$ if $t < t_1$, sc_{n_1,t_1} when $t_1 \leq t < t_2$, and sc_{n_1,t_2} if $t \geq t_2$ for the same node n_1. The corresponding time validities are depicted in Figure 5.3.

Since state chunks with this semantic have temporal validities, relationships between nodes also have temporal validities. This leads to *temporal relationships* between TG nodes and forms a natural extension of relationships in the time dimension.

Once the time resolution returns the correct timepoint t_i, the temporal graph can be reduced to a base graph, therefore a TG for a particular t can be seen as a base graph:

Definition 10 *$TG(t) \equiv BG_{t_i}$, for $t = t_i$.*

5.3.5 Many-world graph (MWG)

To extend the TG with a many-world semantic, we refine the definition of the resolution function $read(n, t)$, by considering, in addition to time the different worlds, the function $read(n, t, w)$, with $t \in T$ and $w \in W$, where W is the set of all possible worlds, which resolves the state chunk of node n at timepoint t in world w. In analogy to Section 5.3.4, the state chunk definition is extended as follows:

Definition 11 *$stateChunk_{n,t,w} = (A_{n,t,w}, R_{n,t,w})$,*
where $A_{n,t,w}$ and $R_{n,t,w}$ are the sets of resolved values of attributes and relationships, for the node n at time t, in world w.

From this definition, a *many-world graph (MWG)* is formalised as:

Definition 12 *$MWG(t, w) = \{read(n, t, w), \forall n \in N\}, \forall (t, w) \in T \times W$, where W is a partially ordered set of all possible worlds.*

The partial order $<$ on the set W is defined by the **parent** ordering, with $(p < w) \equiv (p = parent(w))$. Intuitively, the set W is partially ordered by the generations of worlds, however worlds that are created from the same parent, or the world that are created from different parents, cannot be compared (in terms of order) to each other. Moreover, we define the first created world as the **root world**, with $parent(root) = \emptyset$. Then, all other worlds are created by diverging from the root world, or from any other existing world in the world map set WM of our many-world graph. The divergence operation is defined as follows:

Definition 13 *Operation $w = diverge(p)$:*
$World \mapsto World$, as the function that creates world w from the parent world p, with $p < w$ and $p \in WM \subseteq W$. After the divergence, we have: $WM := WM \cup \{w\}$

According to this definition, we call the world w as the **child** of world p and it is added to the world map of our many-world graph. For the many-world graph, we define the **local timeline of a world and a node** as $ltl_{n,w}$:

Definition 14 *$ltl_{n,w} = \{stateChunk_{n,t,w}, \forall t \in TP_{n,w}\}$*

With $TP_{n,w} \subseteq T$, which is the ordered subset of timepoints for node n and world w, *i.e.,* the timepoints where node n in world w has been changed. As $TP_{n,w}$ is ordered, there exists a timepoint $s_{n,w}$, which is the smallest timepoint in $TP_{n,w}$. $s_{n,w}$ defined as: $s_{n,w} \in TP_{n,w}, \forall t \in TP_{n,w}, s_{n,w} < t$. We call this timepoint a **divergent timepoint**—*i.e.*, where the world w starts to diverge from its parent p for node n. Following the shared-past concept between a world and its parent as described in Section 5.3, we define the global timeline of a world per node as:

Definition 15

$$tl(n,w) = \begin{cases} \emptyset \text{ if } w = \emptyset \\ ltl(n,w) \cup subset\{tl(n,p), t < s_{n,w}\}, p < w \end{cases}$$

The global timeline of a world, according to this definition, is the recursive aggregation of the local timeline of the world w, and the subset of the global timeline of its parent p, up until the divergent point $s_{n,w}$.

Finally, we extend the operations `insert` and `read` as:

Definition 16 *Operation* $insert(stateChunk_{n,t,w}, n, t, w)$: $(StateChunk \times N \times T \times W) \mapsto \{\emptyset\}$*, as the function that inserts a state chunk in the local timeline of node* n *and world* w*, such as:* $ltl_{n,w} := ltl_{n,w} \cup \{stateChunk_{n,t,w}\}$

Definition 17 *Operation* $read(n, t, w)$: $(N \times T \times W) \mapsto StateChunk$*, as the function that retrieves a state chunk from a world* w*, at time* t*. It is recursively defined as:*

$$read(n,t,w) = \begin{cases} read_{ltl_{n,w}}(n,t) & \text{if } (t \geq s) \wedge (ltl_{n,w} \neq \emptyset) \\ read(n,t,p) & \text{if } (t < s) \wedge (p < w, p \neq \emptyset) \\ \emptyset & Otherwise \end{cases}$$

The `insert` operation always operates on the local timeline $ltl_{n,w}$ of the requested node n and world w. For the read operation, if the requested time t is greater or equal to the divergent point in time $s_{n,w}$, of the requested world w and node n, the read is resolved on the local timeline $ltl_{n,w}$, as defined in Section 5.3.4. Otherwise, if the time is less than the divergence timepoint, we recursively resolve on parent p of w, until we reach the corresponding parent to read from.

Once the world resolution is completed, a many-world graph state chunk can be reduced to a temporal graph state chunk, which in turn can be reduced to a base graph state chunk once the timepoint is resolved. Similarly, over all nodes, a many-world graph can be reduced to a temporal graph, then to a base graph, once the read function dynamically resolves the world and time.

Figure 5.4 shows an example of a MWG with several worlds, where w_0 is the root world. In this figure, w_1 is diverged from w_0, w_2 from w_1, and w_3 from w_0. Thus, we have the following partial order: $w_0 < w_1 < w_2$ and $w_0 < w_3$. But no order between

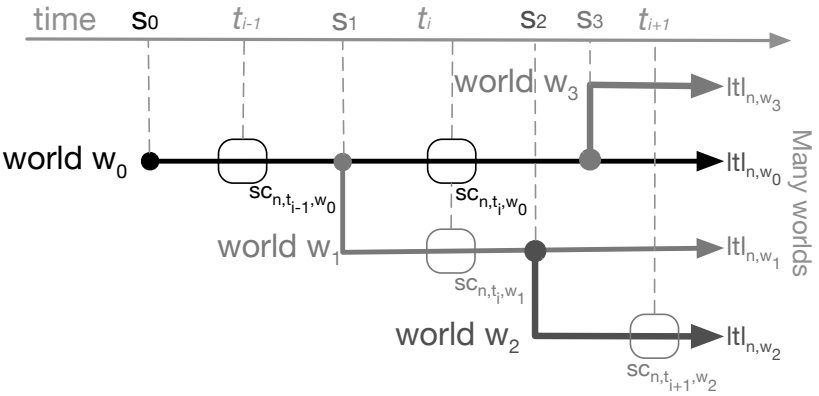


Figure 5.4: Many worlds example

w_3 and w_2 or between w_3 and w_1; s_i for i from 0 to 3, represent the divergent timepoint for world w_i respectively. An insert operation on the node n and in any of the worlds w_i, will always insert in the local timeline ltl_{n,w_i} of the world w_i. However, a read operation on the world w_2, for instance, according to the shared-past view, will resolve a state chunk from ltl_{n,w_2} if $t \geq s_2$, from ltl_{n,w_1} if $s_1 \leq t < s_2$, from ltl_{n,w_0} if $s_0 \leq t < s_1$, and $\emptyset$ if $t < s_0$.

It is important to note that this semantic goes beyond copy-on-write [153] strategies. In fact, a world is never copied, not even if data is modified. Instead, only modified nodes are copied and transparently loaded.

5.4 MWG implementation

Our MWG concept is supported by an implementation and integration into the Kevoree Modeling Framework, to provide the support for creating, reading, updating, forking, and deleting graphs and nodes along time. In particular, the following sections therefore dive into the implementation details to clarify the technical choices we made to outperform the state of the art.

5.4.1 Mapping graph nodes to state chunks

The MWG is a conceptual view of data to work with temporal data and to explore many different alternative worlds. Internally, we structure the data of a MWG as an unbounded set of what we call *state chunks*. Therefore, as discussed in Section 5.3, we map the conceptual nodes (and relationships) of a MWG to *state chunks*. State chunks are the internal data structures reflecting the MWG and at the same time also used for storing the MWG data. A state chunk contains, for every attribute of a node, the name and value of the attribute and, for every outgoing relationship, the name of the relationship and a list of identifiers of the referenced state chunks. Figure 5.5 depicts, in form of a concrete example, how nodes are mapped to state chunks in accordance

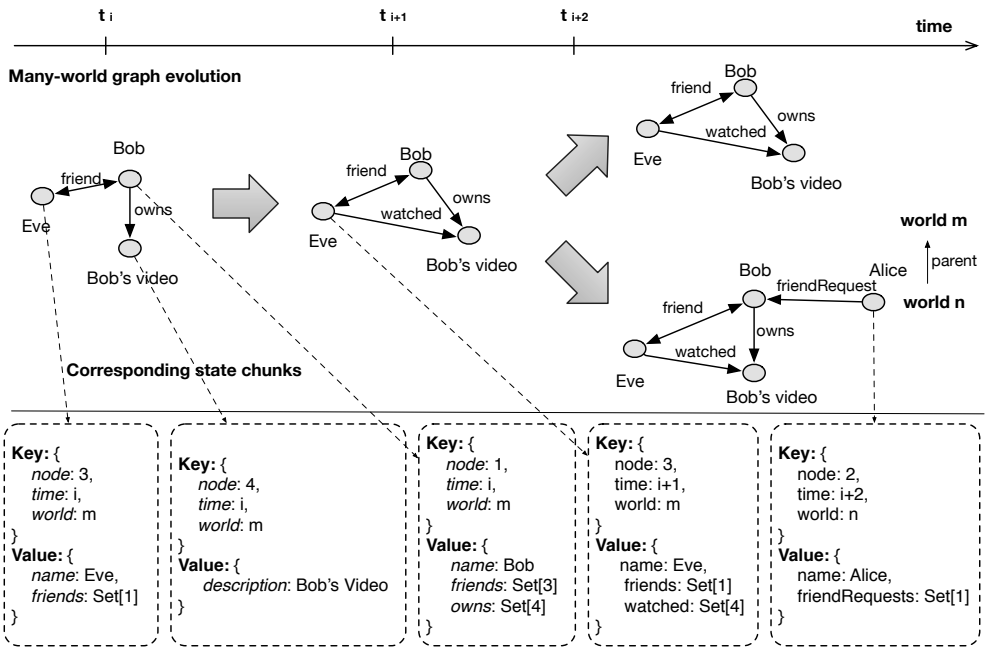


Figure 5.5: Mapping of nodes to storable state chunks

with the semantic definitions of Section 5.3.2.

As it can be seen for time t_i (the starting time of the MWG), we map the nodes and the relationships to three state chunks: one for *Eve*, one for *Bob*, and one for *Bob's video*. At time t_{i+1}, the MWG evolves in the form that a relationship *watched* from *Eve* to *Bob's video* is added. Since this evolution only affects *Eve*, we only create one additional state chunk for *Eve* for the corresponding time t_{i+1}. All other nodes are unchanged at time t_{i+1} and therefore are still valid. Then, at time t_{i+2}, world m of the MWG diverges into two worlds, world m and n. While world m remains unchanged, in world n *Bob* meets *Alice*, who sends a friend request to *Bob*. Only *Alice* changes (comes into the game) so that we only create one additional state chunk for *Alice* for time t_{i+2} and world n. Here, we see the advantage of the MWG and its semantic: while we are able to represent complex graph data, which evolves in time and in many worlds, we only need to store a fraction of this data. In this example, the graph contains semantically 13 different nodes and 16 relationships (counting each bidirectional relation as two) and evolves in two different worlds and three different timestamps, but we only have to create five state chunks to represent all of this. Whenever the MWG is traversed (or data queried in some form) the correct state chunks are retrieved with the right time and world. The resolution algorithm behind this is presented in details in Section 5.4.2.3.

State chunks are the units we use for storage—and as we will see in the next Chapter, also for distribution. They are stored on disk and loaded into main memory while the MWG is traversed or when nodes are explicitly retrieved. Loading state chunks can be qualified as lazy, because only attributes and sets of identifiers are loaded. This theoretically allows to process unbounded MWGs. For persistent storage of state chunks, we rely on common key/value stores by using the 3-tuple of $\{node; time; world\}$

as key and the state chunk as value. We serialise the chunk state into a Base64 encoded JSON `string`. Despite being simple, this format has the advantage that state chunks can be easily distributed over networks (*cf.* Chapter 6). Moreover, it reduces the minimal required interface to insert state chunks into and read from a persistent data store to put and get operations. This allows to use different storage backends depending on the requirements of an application: from in-memory key/value stores up to complex and distributed NoSQL databases.

5.4.2 Indexing and resolving state chunks

In this section, we detail the implementation of the index structures used in KMF and the state chunk resolution algorithm. We combine two structures for the indexes of the MWG: *time trees* and *many-world maps*.

5.4.2.1 Index time tree (ITT)

As discussed in Section 5.3.3, timepoints are chronologically ordered. This creates implicit intervals of "validity" (*cf.* Chapter 4) for nodes in time. Finding the right "position" in a timeline of a node must be very efficient. New nodes can be inserted at any time, *i.e.,* not just after the last one. Besides skip lists [265], ordered trees (*e.g.,* binary search trees) are suitable data structures to represent a temporal order, since they have efficient random insert and read complexities. If we consider n to be the total number of modifications of a node, the average insert/read complexity is $O(log(n))$ and $O(n)$ in the worst case (inserting new nodes at the end of a timeline). Given the fact that inserting new nodes at the end of a timeline and reading the latest version of nodes is the common case, we use red-black trees for the implementation of our time tree index structure. The self-balancing property of red-black trees avoids that the tree only grows in depth and improves the worst case of insert/read operations to $O(log(n))$. Furthermore, we used a custom Java implementation of red-black trees, using primitive arrays as a data backend to minimise garbage collection times, as garbage collection can be a severe bottleneck in graph data stores [311]. Every conceptual node of a MWG can evolve independently in time. For scalability reasons, we decided to use one red-black black tree, further called *Index Time Tree (ITT)*, per conceptual node to represent its timeline. Figure 5.6 depicts how the ITT looks like and evolves for the node *Eve* introduced in Figure 5.5.

As it can be seen, at time t_i, one conceptual version of node *Eve* exists and, therefore, the ITT has only one entry. At time t_{i+1}, *Eve* changes, a new conceptual version of this node is created and the ITT is updated accordingly. Then, at time t_{i+2}, there are additional changes on the MWG, which do not impact *Eve*: the ITT of *Eve* remains unchanged.

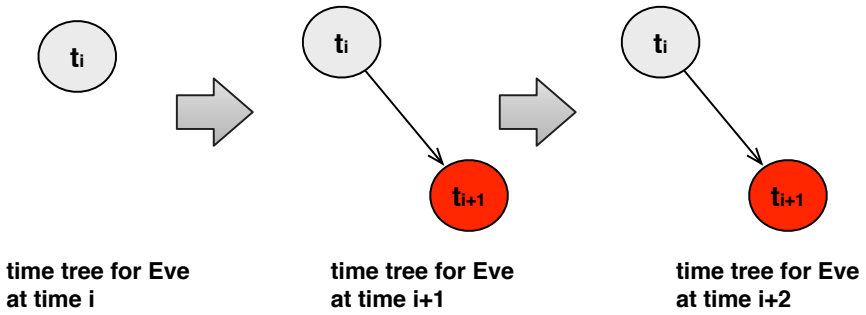


Figure 5.6: Example ITTs for node *Eve* of Figure 5.5

5.4.2.2 World index maps (WIM)

Since new worlds can diverge from existing worlds at any time and in any number, the hierarchy of worlds can arbitrarily grow both in depth and width. As it can be observed in Figure 5.4, the divergent point is therefore not enough to identify the parent relationship. In our many-world resolution, we use a global hash map, which stores for every world w the corresponding parent world p from which w is derived: $w \rightarrow p$. We refer to it as *Global World Index Map (GWIM)*. This allows us to insert the parent p of a world w, independently of the overall number of worlds, in average in constant time $O(1)$ and in the worst case in $O(l)$, where l is the total number of worlds. We also use a custom Java hash map implementation built with primitive arrays to minimise garbage collector effects.

In addition to the GWIM, we define one local index map, called *Local World Index Map (LWIM)*, per conceptual node to identify different versions of the same conceptual node in different worlds. In this map, we link every world in which a node exists with its "local" divergent time, meaning the time when this node was first modified (or created) in this world and therefore starts to diverge from its parent: $w \rightarrow t_{local\ divergence}$. When a conceptual node is first modified (or created) in a world, its state chunk is copied (or created) and the LWIM of the node is updated, *i.e.,* the world in which the node was modified, is inserted (and mapped to its local divergence time). Both, the GWIM as well as the LWIM must be recursively accessed for every read operation of a node (see the semantic definition in Section 5.3.5).

Other than the total number of worlds l, we define another notation: m, as the maximum number of hops necessary to reach the root world, *i.e.,* the depth ($m \leq l$). Figure 5.7 reports an example of two MWG with the same number of worlds $l = 6$, but in the first case we can always reach the root world in $m = 1$ hop, while in the second case, we might need $m = 4$ hops in the worst case (from world w_4 to w_0).

The recursive world resolution function has a minimum complexity of $O(1)$ in the best case, where all worlds are directly derived from the root world (shown in Figure 5.7-a). The worst case complexity is $O(m) \leq O(l)$, like for the **stair-shaped** case shown in Figure 5.7-b, where we might to have to go several hops down before actually resolving the world. In the next section, we show how these index structures are used to resolve state chunks.

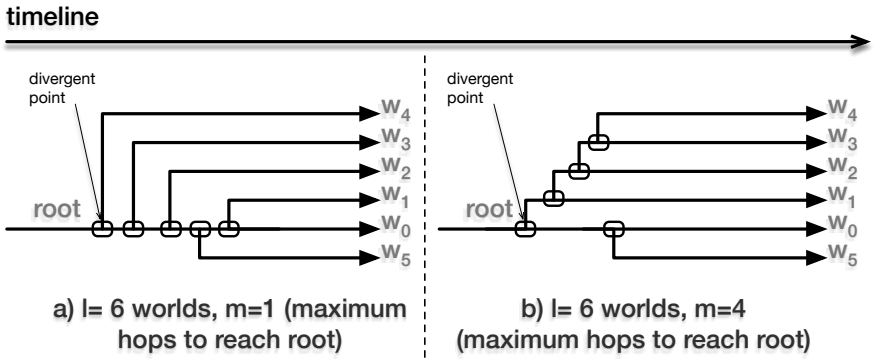


Figure 5.7: Example of different configurations of the same number of worlds l, but with a different m

5.4.2.3 Chunk resolution algorithm

To illustrate the resolution algorithm implemented in KMF, let us consider the example of Figure 5.5. Assuming we want to resolve node *Bob* at time t_{i+2} in world n. We first check the LWIM of *Bob* and see that there is no entry for world n, since *Bob* had never been modified in this world. Therefore, we resolve the parent of world n with the GWIM, which is world m. A glance in the LWIM of *Bob* reveals that world m diverged (or started to exist in this case) for *Bob* at time t_i. This indicates that world m is the correct place to lookup state chunks, since we are interested in *Bob* at time t_{i+2}, which is after time t_i where world m for *Bob* starts to be valid. World m is the "closest" where *Bob* has been actually modified. Otherwise, it would have been necessary to recursively resolve the parent world of m from the GWIM until we find the correct world. In a last step, we look at the ITT of *Bob* to find the "closest" entry to time t_{i+2}, which is time t_i. Finally, this index indicates KMF to resolve the state chunk for *Bob* (id 1) with the following parameters: $\{node\ 1; time\ i; world\ m\}$. This state chunk resolution is summarised in Listing 3.

Listing 3 State chunk resolution

```
1:  procedure RESOLVE(id, t, w)
2:      lwim ← getLWIM(id)
3:      s ← lwim.get(w)
4:      if t >= s then
5:          itt ← getITT(id)
6:          return itt.get(t, w)
7:      else
8:          p ← GWIM.getParent(w)
9:          return resolve(id, t, p)
10:     end if
11: end procedure
```

The full resolution algorithm has a complexity of $O(1)$ for insert, and a complexity of $O(1) + O(m) + O(n) \leq O(l) + O(n)$ for read operations, where l is the number of worlds, and n number of time points, and m maximum depth of worlds.

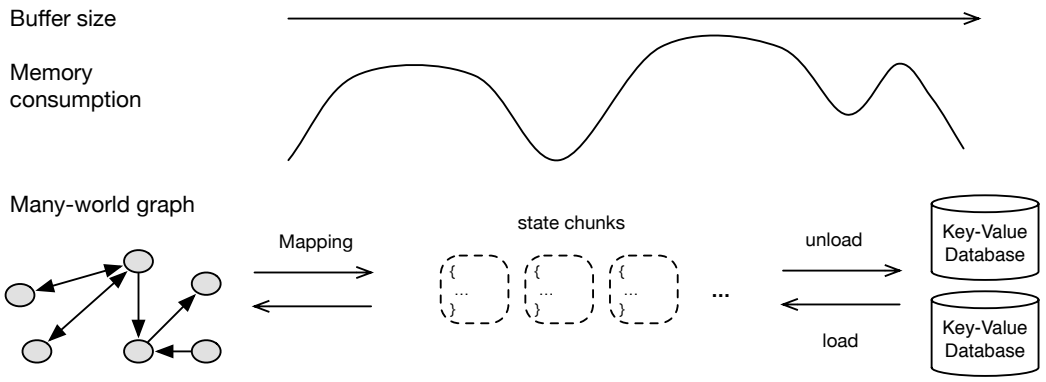


Figure 5.8: Graph memory management in KMF

5.4.3 Scaling the processing of graphs

Memory management and transactions or "units of work" are closely related. In our implementation of KMF, we first need to connect our framework to a database. This connection, further called *unit of work (UoW)*, marks the beginning of what can be seen in a broader sense as a long-living transaction. While working with this connection the state chunks representing the MWG are loaded on-demand into main memory. All modifications of the MWG are therefore performed in memory. When saving, the modified (and new) state chunks are written from memory into persistent key/value stores and the used main memory is cleared. This marks the end of this connection (unit of work or long-living transaction).

To non-intrusively work with graphs of unlimited sizes, we allow to optionally automate the handling of UoWs. Therefore, it is possible to define a fixed buffer size for UoWs. Whenever the actual memory consumption is close to exceed the defined buffer size (*e.g.*, reaches 80% of the specified size) the state chunks are automatically stored in the persistent key/value store and the memory is freed. This, together with the on-demand loading of state chunks into main memory, allows KMF to non-intrusively work with MWGs of theoretically unlimited sizes. The automated memory management mechanisms are depicted in Figure 5.8.

This comes with the drawback that it is hard to isolate or rollback changes made in a UoW since they could be already (partly) persisted. For this reason, we are working on the concept of *merging* a world into another one. This would allow KMF to use the concept of worlds as isolated transactions. A new UoW could then simply always start a new temporal world and in case the UoW should be actually saved, the temporal world could be merged into the world where it diverged from.

5.4.4 Querying and traversing graphs

For querying the MWG, we provide a language which syntax is similar to graph traversal languages, like Gremlin [272]. The novel part is that we can specify the world w

and time t that we want to explore before traversing the graph, *e.g.,* `mwg.of(w, t)`. Then, when we traverse the graph, state chunks will be always resolved (according to Algorithm 3) with this world and time. Besides this, our graph traversal language is rather standard so that we do not detail it here. It should be mentioned that nodes and attributes are strongly typed and, as mentioned in Section 5.3.3, these types map directly to the ones of modern object-oriented languages. This allows to provide an interface for these languages (we currently support Java and TypeScript), which enables to query the MWG directly from a program and return correspondingly typed objects. All graph traversal operations are performed asynchronously and a scheduler transparently dispatches each call to the available resources. For future work, we plan to integrate secondary indexes, which we do not support for the time being.

5.5 Experiments

Our MWG is designed to independently explore different actions on several nodes in time and many alternative worlds. The goal of this section is to thoroughly assess the theoretical complexity of the MWG implementation for time and world resolution under load and to identify its practical boundaries.

First, the experimental setup is detailed in Section 5.5.1. Then, we evaluate the performance of our base graph by comparing it with Neo4j, a state of the art graph database. In this first benchmark we evaluate only the base graph without time and many worlds.

After this, we benchmark the ITT in Section 5.5.3 by inserting/reading millions of timepoints for the same node and in the same world. As a reference, we compare our results to influxDB [39], a state of the art time series database. It is important to keep in mind that the MWG is able to handle full temporal graph data where every node and every relationship can independently evolve (*cf.* Chapter 4), whereas time series databases are designed for "flat" timestamped values. We chose to compare the performance of our approach to influxDB for three main reasons: First, as of today, it is one of the fastest time series databases, which optimise both read and insert operations with dedicated time indexes, similar to our implementation. Secondly, it does not include any third party dependencies, thus making a direct comparison more meaningful. And last but not least, they provide a performance benchmark, which we use as reference for the comparison.

In Section 5.5.4, we validate the complexity of insert and read operations of our world index map over $2,000$ nodes for 2 different worlds. We extend this experiment to a larger scale in Section 5.5.5, by varying the percentage of modified nodes and number of nested worlds. In Section 5.5.6 we simulate a what-if scenario from the domain of evolutionary algorithms, where a small percentage of nodes (mutation rate 3 %) changes between each generation. We show that our MWG is appropriate for such applications and can scale to hundreds of thousands of independent worlds. Finally, we evaluate our implemented solution on a concrete real-world case study from the smart grid domain in Section 5.5.7, to asses its practical capabilities and limits.

5.5.1 Experimental setup

For all experiments, we use the throughputs of **insert** and **read** operations as key performance indicators. We executed each experiment 100 times to assess the reproducibility of our results. Unless stated otherwise, all reported results are the average of the 100 executions. All experiments have been executed on the high performance computer (HPC)[2] of the University of Luxembourg (Gaia cluster). We used a Dell FC430 instance with 2 Intel Xeon E5-2680 v3 processors, running at 2.5 GHz clock speed and 128 GB of RAM. The experiments were executed with Java version 1.8.0_73. All experiments (except the comparison to influxDB) have been executed in-memory without persisting results. The rational behind this is that we want to evaluate our MWG implementation and not the performance of 3rd party key/value stores, which we use for persisting data. All experiments are available on GitHub[3].

5.5.2 Base graph benchmarks

The objective of this experiment is to evaluate the performance of our MWG implementation as a standard graph storage, neglecting time and many-worlds. Therefore, this section compares the performance of our implementation to state of the art graph databases. For this comparison, we use the graph database benchmark [34] provided by Beis *et al.,* [87]. This benchmark is based on the problem of community detection in online social networks. It uses the public datasets provided by *Stanford Large Network Dataset Collection* [62]. This dataset collection contains sets from "social network and ground-truth communities" [324], which are samples extracted from Enron, Amazon, YouTube, and LiveJournal.

The benchmark suite defines several metrics, among which:

- *MIW* to create the graph database and configure it for massive loading, then populate it with a particular dataset. The time for the creation of the whole graph is measured.
- *SIW*: to create the graph database and populate it with a particular dataset. Every object insertion (node or edge) is committed directly and the graph is constructed incrementally. The insertion speed is then measured.

We compare the performance of our MWG implementation to Neo4j, which was the best performing base graph in [87]. Table 5.1 reports on the results of MIW and SIW, for both, our MWG implementation and Neo4j, along the different datasets. For all benchmarks, MWG outperforms Neo4j by factors ranging from 1.3x to 20x, especially in the SIW benchmark, due to the advanced caching techniques implemented in MWG, which allows the MWG to retrieve a node much faster.

[2] https://hpc.uni.lu

[3] `https://github.com/kevoree-modeling/experiments`

Table 5.1: MIW and SIW benchmark speed in 1000 values/second for both MWG and Neo4J. Larger numbers mean better results (shown in bold).

			MIW		SIW	
dataset name	**nodes** (x1000)	**edges** (x1000)	**Neo4J** (x1000 val/sec)	**MWG** (x1000 val/sec)	**Neo4J** (x1000 val/sec)	**MWG** (x1000 val/sec)
Enron	36	367	54.3	**162.4**	2.1	**24.4**
Amazon	403	3,387	319.2	**433.4**	1.0	**17.5**
YouTube	1,134	2,987	121.0	**153.0**	0.5	**17.3**
LiveJournal	3,997	34,681	99.2	**314.1**	0.3	**10.2**

Table 5.2: Average insert and read time in thousands of values per second. The execution is for different timepoints for the same node and in the same world.

(n) in millions	**Insert speed (1000 val./s)**	**Read speed (1000 vsl./s)**	**Insert / log(n)**	**Read / log(n)**
1	589.17	605.30	42.6	43.8
2	565.05	564.11	38.9	38.8
4	554.40	544.23	36.4	35.8
8	537.22	528.18	33.8	33.2
16	520.98	516.26	33.2	31.1
32	515.05	485.73	29.8	28.1
64	489.55	458.32	27.2	25.5
128	423.53	400.49	22.7	21.5
256	391.56	378.50	20.2	19.5

5.5.3 Temporal graph benchmarks

The aim of this experiment is to validate the complexity of the ITT (cf. Section 5.4.2.1). We compare the performance of temporal data management of our approach with plain time series databases. Therefore, we consider only **one world** and **one node id** and benchmark the throughput of insert and read operations over a varying size of timepoints, from 1 million to 256 million. Table 5.2 reports the measured results under progressive load, to check the complexity according to the expected one.

As one can observe, read and insert performance follows an $O(log(n))$ scale as n increases from 1 million to 256 million. The performance deterioration beyond 32 million can be explained due to a 31 bit limitation in the hash function of the ITT. This comes from the fact that our ITT is implemented as a red-black tree backed by primitive Java arrays. These are limited to 31 bit indexes (1 bit is used for the sign). At these large numbers, collisions become very recurrent. For instance, for the 256 million case, there are around 8% of collisions. This compares to less than 0.02% of collisions for 1 million.

To address this problem, we plan for future work an off-heap memory management implementation (based on Java's unsafe operations), which would allow us to solve the limitation of 31 bit indexes for primitive arrays and to use hash functions with more than 31 bits.

For the comparison with a time series database, influxDB in this case, we use the influxDB benchmark [38]. It consists of creating $1,000$ nodes (time series), where $1,000$ historical values are inserted in each node on a standard MacBook Pro. The second test consists of creating $250,000$ nodes, where $1,000$ historical values are inserted in each, executed on an Amazon EC2 i2.xlarge instance.

The main difference with the experiment presented above is that the ITT of each node does not grow the same way in terms of complexity as an ITT of 250 million elements in a single node does. Just for the sake of comparison, we applied the same benchmarks using the same machine types. We use RocksDB [57] as our key/value backend. Despite the fact that our MWG is not limited to flat time series, but a full temporal graph, we are able to outperform influxDB by finishing the MacBook test in 388 seconds compared to their 428 seconds (10% faster), and by getting an average speed of $583,000$ values per second on the Amazon instance, compared to their $500,000$ values per second (16% faster). It is important to note that, when all elements are inserted in the same ITT, the speed drops to $391,560$ inserts per second in average (as shown in Table 5.2). This is due to the increased complexity of balancing the ITT of one node. The experiment therefore assess that our implementation is able to manage full temporal graphs as efficiently (on a comparable scale) as time series databases are able to manage flat sequences of timestamped values.

5.5.4 MWG benchmarks of a node

In this experiment, we show the effect on insert and read performance of creating many worlds from one node. Diverging only one world from the root world is not enough to measure a noticeable performance difference. Therefore, we created 100 nested parallel worlds from a root world w_0. We measure the insert performance for the worlds w_0 and w_{100}. Then we measure, for the root world, the read performance R_0 at a shared past timepoint $t_1 = 5000 < s$ and R_1 at timepoint $t_2 = 15000 > s$ (after the divergence). We repeat the experiments for the same timepoints t_1 and t_2, but from the perspective of world w_{100}, to get read performance R_2 and R_3. The results are depicted in Figure 5.9, as box plots over 100 executions. From these results, we can conclude the following points: The insert performance is similar for both worlds. The read performance for the root world is not affected by the divergence $R_0 = R_1$. The read performance of world w_{100} depends on the timepoint. It is faster to read after the divergence point than before it, *i.e.,* $R_3 > R_2$. This is due to the recursive resolution algorithm of our MWG implementation, as explained in Section 5.4.2.2.

In this experiment, we validated that the write and read performance on the many-world graph are not affected by the creation of several worlds. In particular, we also showed that the read speed is kept steady, after the divergence for the child worlds.

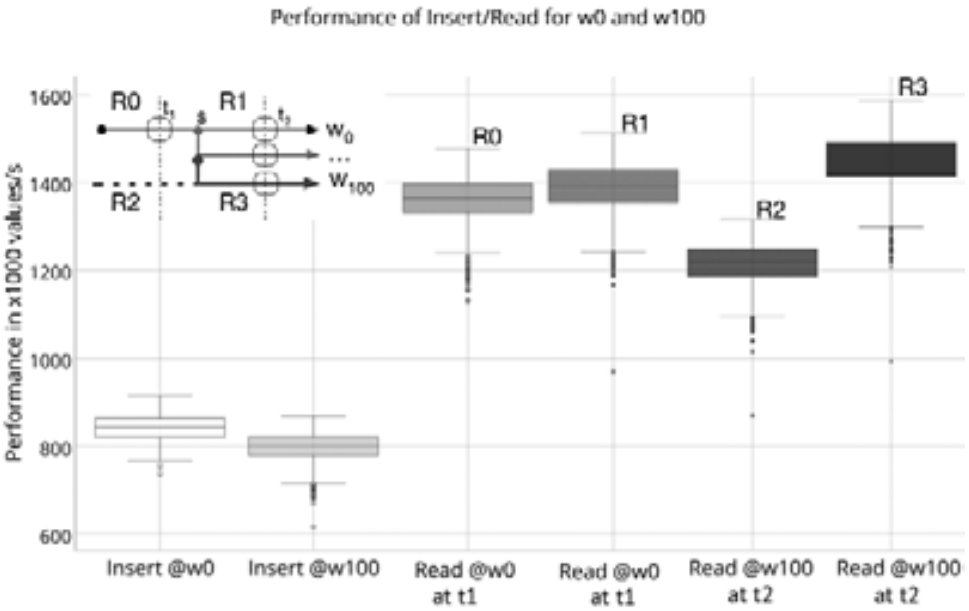


Figure 5.9: Insert and read performance before and after the divergent timepoint **s**

5.5.5 MWG benchmarks of a graph

To study the effect of recursive world resolution for the whole graph, we consider the **stair-shaped** scenario presented in Figure 5.7-b. In this benchmark, we create a graph of $n = 2000$ nodes, each having an initial fixed timeline of $10,000$ timepoints in the main world. Then, we select a fixed $x\%$ amount of these nodes to go through the process of creating the shape of stairs of m steps across m worlds. In each step, we modify one timepoint in the corresponding world of the corresponding node. For this experiment, we vary m from 1 to $5,000$ worlds by steps of 200 and x from 0 to 100% per steps of 10%. This generates 250 different experiments. We executed each experiment 100 times and averaged the read performance of the whole graph before the divergent time, from the perspective of the last world. Figure 5.10 shows the results in form of a heat map of the average read performance for the different combinations of number of worlds and percentage of nodes changed. The brightest area in the figure (lower left) represent the best performance (low number of worlds or low percentage of nodes changed in each world). The darkest area (upper right) represent up to 26% of performance drop (when facing an high percentage of changes and an high number of worlds).

This benchmark is the worst case for the MWG, since for m^{th} world, a read operation might potentially require m hops on the World Index Map (WIM), before actually resolving the correct state (*e.g.,* reading the first inserted node from the perspective of the last created world), as discussed in Section 5.4.2.2. The performance drop is linear in $O(m)$ and it is also linear according to the percentage of nodes changed from one world to another. For less than 20% of changes, the performance drop is hardly noticeable even at an high number of worlds (lower right). It is important to note that our solution only stores the modifications for the different worlds and rely on the resolution algorithm to infer the past from the previous worlds. Any snapshotting

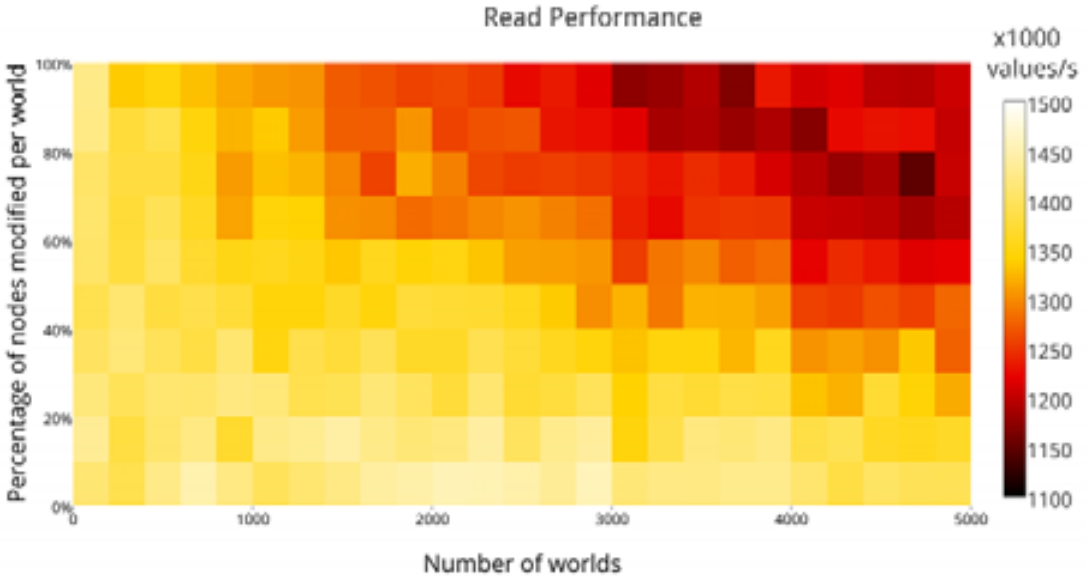


Figure 5.10: Read performance before the divergent timepoint, over several worlds and several percent of nodes modified

technique, cloning the whole graph of 2,000 nodes, each including 10,000 timepoints, for 5,000 times would be much costlier to process than our solution. To sum up, we show in this section that our index structure allows independent evolution of nodes at scale. The performance decreases linearly with the percent of nodes changed and according to the maximum number of reached worlds.

5.5.6 Deep what-if simulations

As the motivation of our work is to enable deep what-if simulations, we benchmark in this section the read performance over a use-case close to the ones we can find in this domain. We use a setup similar to the previous section: a graph of $n = 2,000$ nodes with initially 10,000 timepoints in the root world. The difference is that we fixed the percentage of changes between one world to another to $x = 3\%$ (similar to a nominal mutation rate in genetic algorithms of 0.1%–5% [292]). The second difference is that changes only randomly affect 3% of the nodes for each step. This is unlike the previous experiment, where the target was to reach a maximum depth of worlds for the same amount of $x\%$ of nodes. We executed this simulation in steps of 1,000 to 120,000 generations (120 experiments, each repeated 100 times). The number of generations is similar to the typical number of generations in genetic algorithms [292]. In each generation, we create a new world from the previous one and randomly modify 3% of the nodes. At the end of each experiment, we note the performance of reading the whole graph of 1,000 nodes. Figure 5.11 reports on the results our implementation of a MWG achieves. In particular, one can observe that the read performance drops linearly, 28% after 120,000 generations. This validates the linear complexity of the world resolution, as presented in Section 5.4.2.2 and the usefulness of our approach for what-if simulations, when a small percentage of nodes change, even in a huge amount

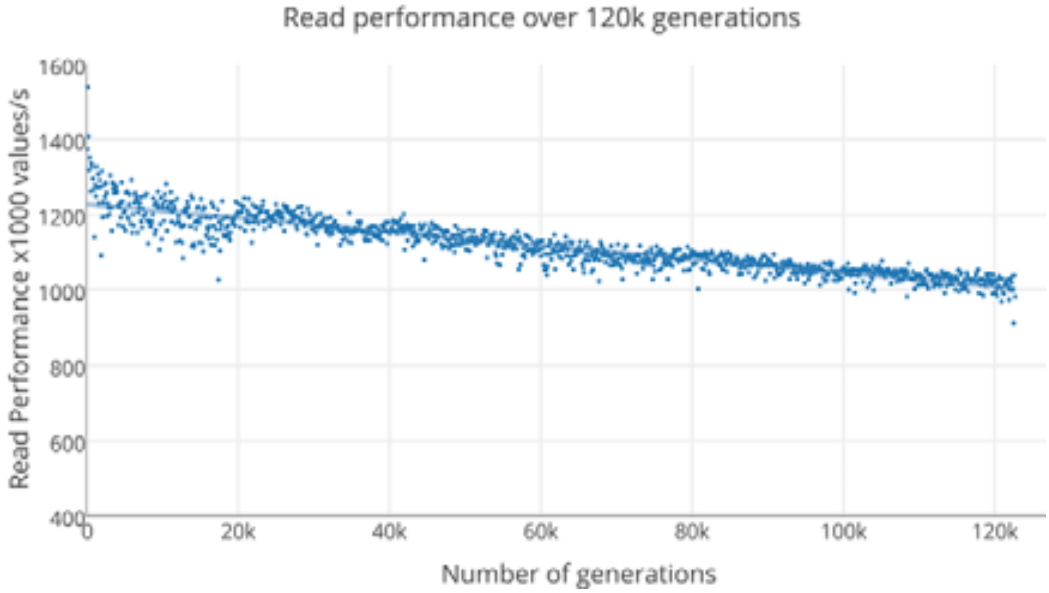


Figure 5.11: Average read performance over 120, 000 generations with 3 % mutations

of deep nested worlds.

5.5.7 Smart grid case study

In this experiment, we evaluate the our implementation on a real-world smart grid case study, which we introduced in Section 5.2. In particular, we leverage our MWG to optimise the electric load in a smart grid. Therefore, we build profiles for the consumption behaviour of customers. Based on the predicted consumption, we simulate different hypothetical what-if scenarios for different topologies, compute the expected electric load in cables, and derive the one with the most balanced load in all cables. This allows to anticipate which topology is likely to be the best for the upcoming days.

For this experiment, we use an in-memory configuration, without a backend storage, because we do not need to persist all the different alternatives. We use the publicly available smart meter data from households in London [60]. As the dataset from our partner Creos Luxembourg S.A. is confidential, we use this publicly available dataset for the sake of reproducibility. The grid topology used in our experiments is based on the characteristics of the Luxembourg smart grid deployment [173]. We consider 5, 000 households connected to the smart grid, considering 4, 000 consumption reports per customer. This leads to 20, 000, 000 values used to learn the profiles. As described in [173] around 100 customers are connected to one transformer substation. We simulate 50 power substations for our experiments and we suppose that every household can be connected to every power substation. This is a simplification of the problem, since which household can be connected to which power substation depends on the underlaying physical properties of the grid, which we neglect in the following experiment.

Figure 5.12 reports on the simulation results over 500, 000 worlds, where in each world

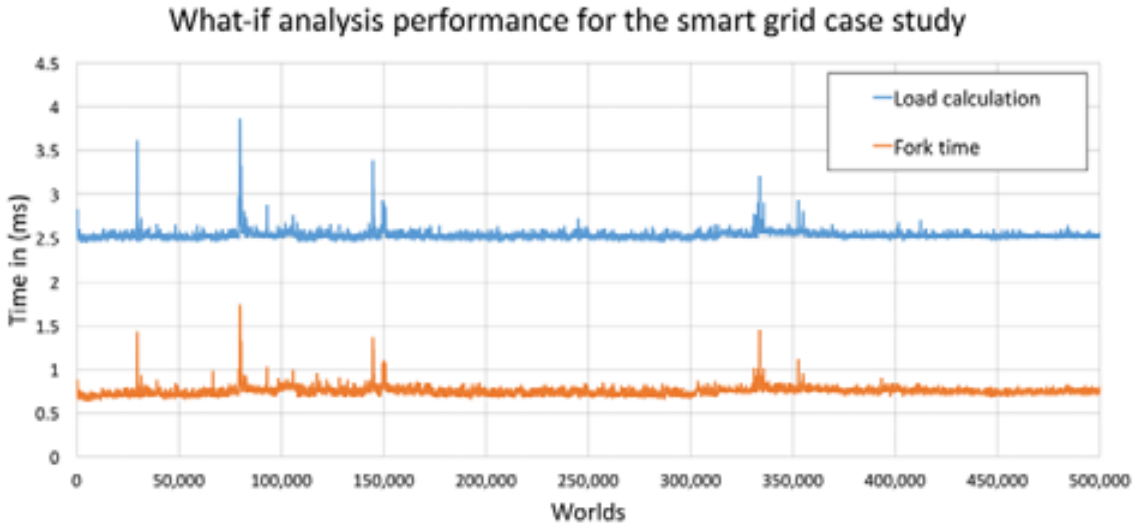


Figure 5.12: Performance of load calculation in a what-if scenario for the smart grid case study

we mutate 3% of the power substations connections to smart meters. We plot the time spent (in *ms*) on the load calculations and world creation (fork time) per world. As depicted in Figure 5.12, both curves are quite constant, with some peaks due to garbage collection. Based on Figure 5.12, we can conclude that our proposed MWG is scalable and can apply to large-scale systems, such as the smart grid.

5.5.8 Discussion and perspectives

Beyond the specific case of smart grids we described in this chapter, we believe that MWGs can find applications in a large diversity of application domains, including social networks [198], digital marketing, smart cities, healthcare, sales, and biology [167]. For example, in the case of smart cities, a MWG can store and learn the mobility models of citizens and then explore the impact of closing/opening roads on the traffic. Another domain of application for such what-if analysis is weather forecasting. As weather forecasts are built on complex models, anticipating the impacts of certain effects (*e.g.,* air pollution) requires to simulate what would happen in such cases, based on complex simulation models. Additionally, in the domain of software engineering, MWGs can be used to trace the evolution of mobile apps [180] and thus identify the sequence of refactoring actions to be performed in order to improve the software quality. MWGs can also be used to monitor the execution of deployed software and explore future states, thus predicting the impact of changing parameters or executing specific actions.

Aside of potential applications of this approach, our perspectives also include the extension of MWGs to consider different *laws of evolution* for the stored graphs, thus going beyond the application of machine learning [179]. We are also looking at the integration of our solution with existing graph processing systems, like Giraph [5]. Finally, beyond the support of what-If analysis, the coverage of alternative prescriptive analytics based on MWG is another research direction we are aiming for.

5.6 Conclusion

We proposed a novel graph data model, called *many-world graph*, which allows to efficiently explore a large number of independent actions—both in time and many worlds—even on a massive amount of data. We validated that our MWG implementation follows the theoretical time complexity of $O(log(n))$ for the temporal resolution and $O(m)$ for the world resolution, where m is the maximum number of nested worlds. Our experimental evaluation showed that even when used as a base graph—without time and many-worlds—our MWG implementation outperforms a state of the art graph database, Neo4j, for both mass and single inserts. A direct comparison with a state of the art time series database, influxDB, showed that although the MWG is not just a simple time series, but a fully temporal graph, the temporal resolution performance is comparable or in some cases even faster than time series databases. The experimental validation showed that the MWG is very well suited for what-if analysis, especially when only a small percentage of nodes changes. Regarding the support for prescriptive analytics, we showed that the MWG implementation is able to handle efficiently hundreds of millions of nodes, timepoints, and hundreds of thousands of independent worlds.

Part III

Reasoning over distributed data and combining domain knowledge with machine learning

6

A peer-to-peer distribution and stream processing model

The models@run.time paradigm promotes the use of models during the execution of cyber-physical systems to represent their context and to reason about their runtime behaviour. In the previous chapters, we introduced a scalable multi-dimensional graph data model that can cope with the complexity and the diversity of representing and exploring many different alternatives, combined with temporal data. However, the recent trend towards highly interconnected cyber-physical systems with distributed control and decision-making abilities makes it necessary to efficiently reason over distributed data. Coping at the same time with the large-scale, distributed, and constantly changing nature of these systems constitutes a major challenge for analytic processes and their underlying data models. This chapter presents a peer-to-peer distribution mechanism for the data model introduced in the previous chapters. A stream processing model on top of this enables to efficiently reason over distributed and frequently changing data. Reasoning over distributed data becomes more and more crucial, given the trend towards highly interconnected cyber-physical systems with distributed control and decision-making abilities, such as smart grids.

This chapter is based on the work that has been presented in the following paper:

- Thomas Hartmann, Assaad Moawad, François Fouquet, Grégory Nain, Jacques Klein, and Yves Le Traon. Stream my models: Reactive peer-to-peer distributed models@run.time. In *18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MoDELS 2015, Ottawa, ON, Canada, September 30 - October 2, 2015*, pages 80–89, 2015

Contents

6.1 Introduction

Over the past few years the models@run.time paradigm has proven the potential of models to be used not only at design-time but also at runtime to represent the context of cyber-physical systems, to monitor their runtime behaviour and reason about it, and to react to state changes [96], [88]. Reasoning on the state of a cyber-physical system is a complex task, since it relies on the aggregation and processing of various constantly evolving data such as sensor values. As detailed in Chapter 4 and Chapter 5, this requires scalable data models that can cope with the complexity and the diversity of representing and exploring many different alternatives, combined with temporal data. Therefore, we introduced a scalable multi-dimensional graph data model (*cf.* Chapter 4 and Chapter 5)—to represent the context of CPSs—that can cope with the complexity and the diversity of representing and exploring many different alternatives, combined with temporal data. However, the recent trend towards highly interconnected cyber-physical systems with distributed control and decision-making abilities makes it necessary to efficiently reason over distributed data.

To fulfil their tasks, these systems typically need to share context and state information between computational nodes. Unlike in the previous chapter, where a node denoted a node in the context of a graph data model, in this chapter a node refers to a computational node, *i.e.,* any computer system reading, writing, or processing data in the context of a cyber-physical system. Given the fact that our approach promotes the use of runtime data models to represent the state and context information of CPSs, the runtime models of distributed CPSs must also be distributed. Moreover, as shown in the previous chapters, runtime models of complex CPSs can get very large and the underlying data can change very frequently. This makes it difficult to share this information efficiently.

Let us consider the smart grid case study as a concrete example. Smart grids are characterised as very complex and highly distributed CPSs [303], where various sensor data and information from the electric topology must be aggregated and analysed. To support reasoning and decision-making processes, we use the smart grid model presented in Section 1.2.2. The state of the smart grid, *i.e.,* its runtime model, is continuously updated with a high frequency from various sensor measurements (like consumption or quality of power supply) and other internal or external events (*e.g.,* overload warnings). In reaction to these state changes, different actions can be triggered. However, reasoning and decision-making processes are not centralised but distributed over smart meters, data concentrators, and a central system [142], making it necessary to share context information between these nodes. The fact that runtime models of smart grids, depending on the size of a city or country, can reach millions of elements and thousands of distributed nodes, challenges the efficiency of sharing context information.

These challenges are not specific to the smart grid but also arise in many other large-scale, distributed cyber-physical systems, where state and context information change frequently. For example, advanced automotive systems, process control, environmental control, avionics, and medical systems [221].

Despite the fact that models@run.time enable the abstraction of such complex sys-

tems during runtime, to the best of our knowledge, there is no approach tackling the *i)* **large-scale**, *ii)* **distributed**, and *iii)* **constantly changing nature** of these systems at the same time [149], [301]. This chapter introduces a distributed models@run.time approach combining ideas from asynchronous, reactive programming, peer-to-peer distribution, and large-scale models@run.time. The introduced distribution and stream processing model allows to distribute our previously proposed multi-dimensional graph data model (*cf.* Chapter 4 and Chapter 5) in a peer-to-peer manner and to efficiently reason over distributed, frequently changing data.

First of all, since models@run.time are continuously updated during the execution of a system, they cannot be considered as bounded but can change and grow indefinitely [119]. Therefore, we define models as observable streams of model chunks, where every chunk contains data related to one model element (*e.g.,* a meter). This stream-based interpretation of models, allows to process models chunk-by-chunk regardless of their global size. Secondly, we distribute and exchange these model chunks between computational nodes in a peer-to-peer manner and on-demand to avoid the exchange of full runtime models. That peer-to-peer distribution can lead to highly scalable implementations has, for example, also been discussed in [197]. Moreover, the use of a lazy loading strategy allows to transparently access the complete virtual model from every node, although chunks are actually distributed across nodes. Thirdly, we leverage observers, an automatic reloading mechanism of model chunks (in case of changes), and asynchronous operations to enable a reactive programming style, allowing a system to dynamically react to context changes.

We integrated our approach into the KMF [147], [151] by entirely rewriting its core to apply a thoroughly reactive and asynchronous programming model. Evaluated on an industrial-scale smart grid case study, inspired by the Creos project, we demonstrate that our approach enables frequently changing, reactive distributed models and can scale to millions of elements distributed over thousands of nodes, while the distribution and model access remains fast enough to enable reactive systems.

The remainder of this chapter is as follows. Section 6.2 presents our approach of reactive distributed models at runtime, which we evaluate in Section 6.3. In Section 6.4 we discuss the need for asynchronicity, to distribute models before we conclude in Section 6.5.

6.2 Reactive distributed models@run.time

This section details our approach of reactive peer-to-peer distributed models@run.time. It begins with an overview of our proposition. It then describes how runtime models are split into chunks to allow to define models of arbitrary size as observable, continuous streams of chunks. Next, this section details how these chunks together with peer-to-peer distribution techniques, lazy loading, and automatic chunk reloading are used to transparently distribute runtime models over computational nodes. Finally, this section presents how asynchronous programming empowers the reactivity of systems regarding changes and events.

6.2.1 Overview: distributed models as data stream proxies

The goal of this contribution is to enable *i)* large-scale, *ii)* distributed, and *iii)* constantly changing models@run.time that can scale to millions of elements distributed over thousands of nodes, while keeping the distribution and model access fast enough to enable reactive systems. To address the distribution and the context sharing need, we propose a concept of runtime models, which are virtually complete and spread over the computational nodes of a distributed CPS. Indeed, every model element can be accessed and modified from every node, regardless on which nodes the model element is physically present. To tackle the large-scale aspect, data is never copied *a priori*. Instead, runtime models are considered as proxies of data, loading the related data only on-demand. This is achieved by splitting runtime models into streams of data chunks, where every chunk corresponds to one model element. These data chunks are physically distributed in a peer-to-peer manner, using distribution strategies similar to those used for media sharing. Finally, reactive programming concepts and a fine-grain (*i.e.,* per model element) load and update strategy are used together, to cope with the constantly changing nature of model elements. Asynchronous operations allow to address the inherent uncertainty of network communications and the reactive aspect empowers models to dynamically react, by observing changes on the shared data stream. These characteristics are closely interlinked and we claim that the combination of the three can offer distributed and scalable models@run.time, able to deal with constantly changing model elements. This approach is depicted in Figure 6.1 and detailed in the rest of this section.

6.2.2 Models@run.time as streams

In a formal way, we denote by N the set of the connected nodes. Every node $n_i \in N$, consists of a tuple of unique id_i and an (infinite) sequence of model chunks S_{n_i}. $S_{n_i} = \{s_{i1}, \ldots, s_{ij}, \ldots\}$. A model chunk corresponds to the state chunks introduced and formalised in the previous chapters (*cf.* Chapter 5). Each model chunk s_{ij} contains the state of one runtime model element me_k. As detailed in Chapter 4 and Chapter 5, considering the dimensions time and alternative worlds, a semantic model element can in fact map to several state chunks—depending on the time and world. However, for the sake of simplicity, we neglect the time and world dimensions in the following explanations. We assume that each state chunk has a unique *id* [207], which is automatically generated when a model element is created during runtime. In our proposed multi-dimensional graph data model (*cf.* Chapter 5) this unique *id* is a 3-tuple of $(id, time, world)$. For reasons of simplification, in this chapter, we generally speak about a unique *id*, instead of the 3-tuple. Each model element contains a set of attributes and a set of relationships to other model elements (*cf.* Chapter 5).

Our global, distributed runtime model M, is thus the aggregation of all these distributed model elements. As argued, runtime models of cyber-physical systems need to be continuously updated to reflect state changes of the underlying systems. This makes it difficult to consider them as bounded [119]. Moreover, as we have already defined a mapping between the model element updates and the model chunks s_{ij}, we

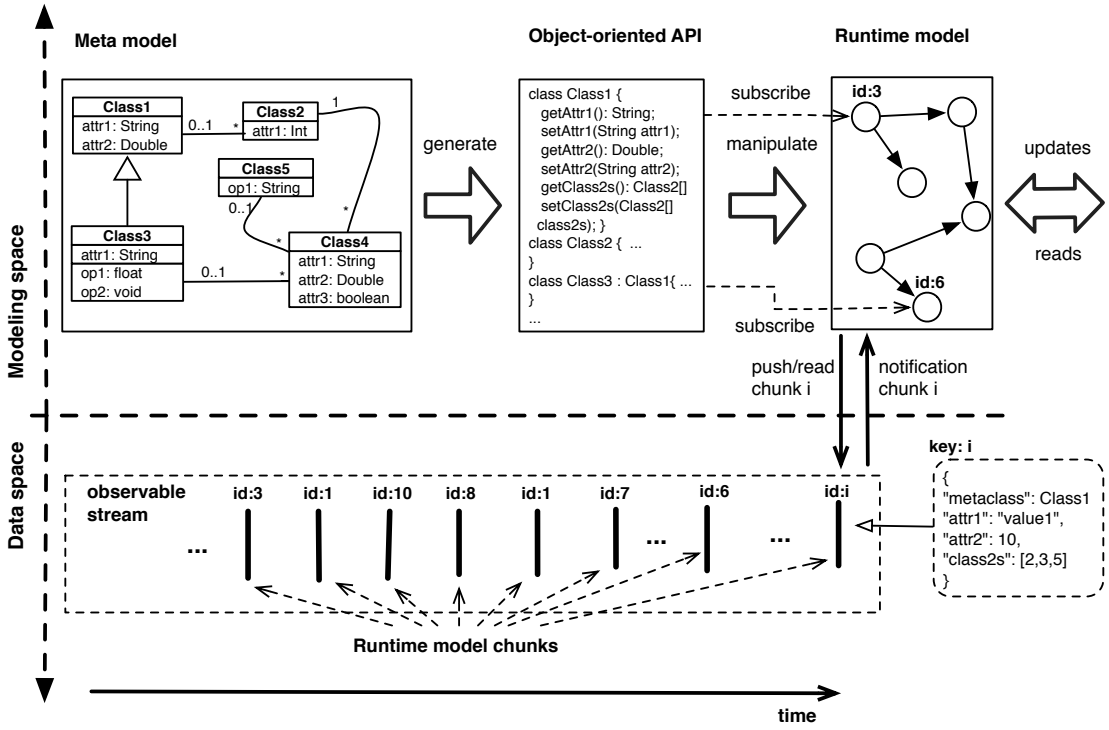


Figure 6.1: Models as continuous streams of chunks

can consider that M is the virtual aggregation of all the model chunks created on all the nodes: $M = \bigcup_{n_i \in N} S_{n_i}$. This is described in Figure 6.1.

Next, we define two functions, *serialize* and *unserialize*, for every model element. The function *serialize* takes a model element as input and creates a compact JSON format *(similar to BSON)* of the runtime model element as output. It contains the unique *id* and data of the model element. The *id* of a model element is immutable, meaning that it cannot be changed during the whole lifetime of an element. Similarly, the *unserialize* function takes a JSON representation of the model element as input and creates a model element as output. The *id* of model elements together with the *serialize* and *unserialize* functions, allow us to split models in a number of chunks.

It is important to note that for each point in time a model still consists of a finite number of chunks. However, considering the fact that a model can continuously, *i.e.,* infinitely, evolve over time, a model can be interpreted as an infinite stream of model chunks, where every model element changed, is added to the stream. Newly created model elements are considered in the same way as changes. To delete elements we define an explicit function *delete*. This function removes an element from all relations where it is referenced. In addition, all elements contained in the deleted one are recursively considered as deleted. Streams are naturally ordered by time, *e.g.,* based on a clock strategy [216].

The definition of a unique *id* for every model element and the way we split models into chunks also allow us to leverage a lazy loading [259] strategy for chunks. Indeed, references are resolved on-demand, meaning that the actual data of model elements are

loaded only when accessed by an explicit request, or by traversing the model graph. For *one-to-one* relationships, the chunks only contain the unique *id* of the target model element, and a (primitive) array of *id*s in case of *one-to-many* relationships. If a relationship changes, the chunk is updated. An example of a chunk can be seen in the lower right corner of Figure 6.1. With this strategy, we enable the loading of model elements on-demand, regardless if they come from a file, local or remote database, or—as discussed in subsection 6.2.3—if they are distributed in a peer-to-peer manner. Since model element chunks can potentially be modified by a concurrent (local or remote) process, we reload model element chunks when they are accessed again. To reduce the overhead caused by this reloading, we use a caching strategy [199] to decide wether an element needs to be reloaded or not. This is a trade-off between the freshness of data and overhead of reloading.

In Section 6.3, we demonstrate that this approach enables to efficiently process large-scale models that do not necessarily fit completely into main memory, by allowing to manipulate models chunk-by-chunk. In addition, this lays the foundation for our peer-to-peer-based distribution approach.

6.2.3 Distributed models@run.time

This subsection describes the peer-to-peer distribution of data chunks. We first discuss how we enable distributed runtime models and how we uniquely identify distributed model elements. Then, we outline the generic content delivery network (CDN) interface to bridge the gap between the model and data space. Finally, we discuss how we distribute data chunks using distributed hash tables (DHTs).

The idea of this contribution is to offer a virtually complete view of runtime models, even though they are actually distributed over several nodes. To ensure consistency between runtime models, they all conform to the same meta model. To avoid the need of a-priori replication, we use runtime models as data proxies. The task of model proxies is to decompose all model operations into data chunk operations. This is the responsibility of a so-called *content delivery network*, which provides operations to retrieve *(get)* and share *(put)* data chunks. Figure 6.2 depicts the distribution architecture of our approach. As can be seen in the figure, every computational node operates on his own instance of the model. Whenever the model is traversed, the corresponding state chunks are loaded from the stream (or local cache) and vice versa, when a computational node changes the model, the corresponding state chunks of the changed elements are pushed to the stream so that other computational nodes can be notified about the changes.

Since model elements can be created on any node, we first have to refine our *id* generation strategy to avoid *id* overlaps. Consensus algorithms like RAFT [257] or Paxos [136] are able to offer strong consistency guaranties. However, they are not designed for very high volatility like needed for object creation. Therefore, we define our *id*s with the goal to reduce the amount of consensus requests based on a leader approach [257]. We use 64 bits *id*s composed of two parts. A 20 bits $prefix$ (most significant bit (MSB))

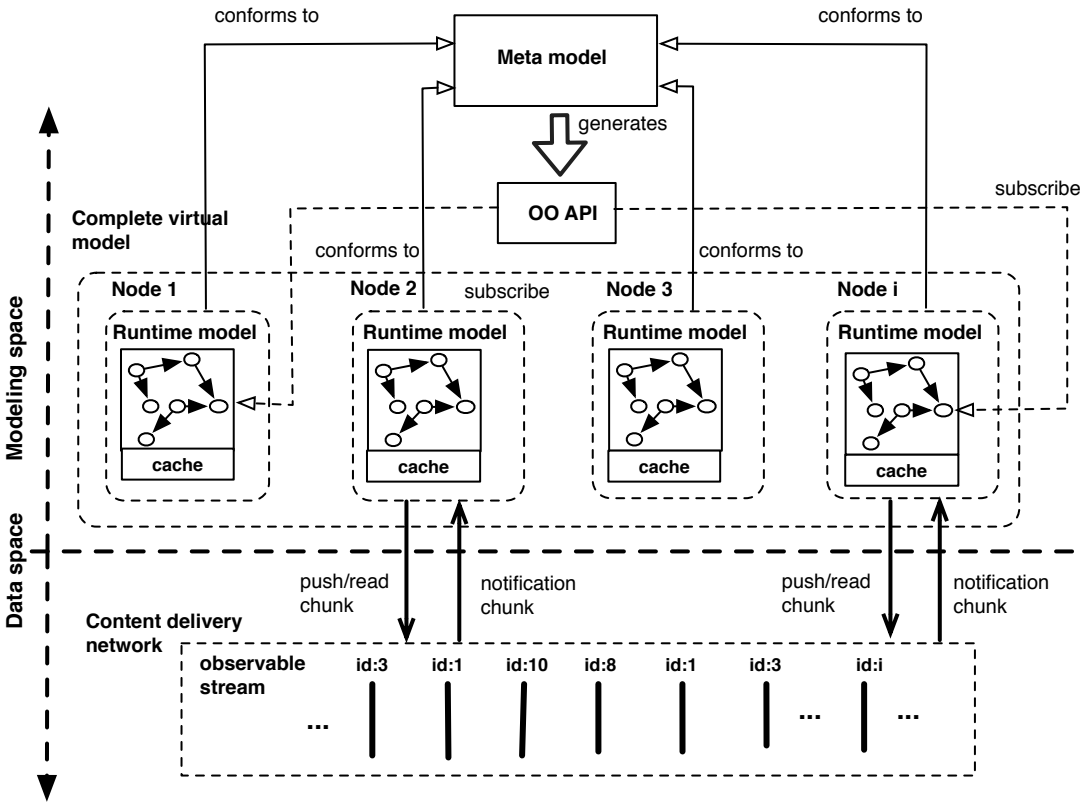


Figure 6.2: Distribution model

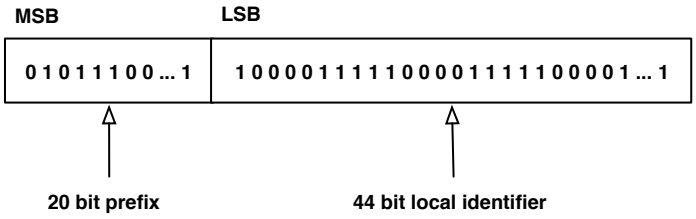


Figure 6.3: Composition of ids for distributed models

is negotiated at the time a node connects to the CDN and is from this point assigned to the node. The remaining 44 bits are used for locally generated identifiers (least significant bit (LSB)). This composition of *id*s is depicted in Figure 6.3.

Prefixes are allocated in a token ring. Besides being a simple implementation for the $prefix$ negotiation, it also allows to reuse $prefixes$ in case the 20 bit limit is reached, *i.e.,* in case more than 2^{20} participants are involved. The number of concurrent connections is limited by the $prefix$ size. With 20 bits $prefix$es we enable more than one million connections without a risk of collision. Managing *prefixes* in a token ring and reusing them, allows us to use more than one million connections, but with an increasing number of connections the risk of collisions also increases. By using 44 bit for local identifiers every node can create 2^{44} objects per $prefix$ (about $17,592$ billions). If a node needs to create more objects, another $prefix$ must be requested from the CDN. As depicted in Figure 6.2, every node relies on a content delivery network, which is responsible for the data chunk exchange strategy, $prefix$ negotiation,

and network related operations. Different implementations of the CDN can support different distribution scenarios. We now focus on peer-to-peer distribution. Listing 4 illustrates a simplified interface definition of a CDN driver.

Listing 4 CDN interface

```
interface ModelContentDelveryDriver {
  atomicGet(byte[] key, Callback<byte[]> callback);
  get(byte[] key, Callback<byte[]> callback);
  put(byte[] key, byte[] val, Callback callback);
  multicast(long[] nodes,byte[] val, Callback callback);
}
```

The method *atomicGet* is used during the $prefix$ negotiation phase and requires a consensus like algorithm, *e.g.,* RAFT. The methods *get* and *put* are the two primary methods to load, store, and share data chunks. These operations can be implemented using a multicast dissemination strategy or with more advanced concepts like distributed hash table algorithms. These algorithms offer partitioning of storage to replicate data over the network. Finally, the method *multicast* is used for the dissemination of modification events to all subscribed nodes. All these methods are asynchronous and use callbacks to inform about their execution results.

A content delivery network driver needs to be instantiated at each node to enable nodes to collaborate to exchange data chunks. Like the PeerCDN project [321], the CDN implementation used in our approach relies on the Kademlia [236] distributed hash table to implement the *get* and *put* operations. Since Kademlia scales very well with the number of participants, we leverage it on top of a WebSocket communication layer.

Another possibility would be using Gossip-like protocols [133] (especially in case of deep network topologies) for propagating this information to the participating peers. Such approach could be, for example, built with GossipKit [304], [225].

6.2.4 Reactive models@run.time

As discussed, a key requirement for models@run.time-based systems is to be able to quickly react to state changes. In order to ensure reactivity, we make our streams of model chunks observable [154]. We enable runtime models to subscribe to these observable streams. Therefore, we define an API that allows to specify which model elements should be observed. This is usually domain-specific knowledge. Then, whenever one of these runtime model elements changes (regardless if due to local or remote changes) the observer (a runtime model) is notified. Different runtime models can observe different model elements, depending on which changes are important for this observer. For example, the top of Figure 6.1 shows that the runtime model subscribes to changes of the two runtime model elements with $id = 3$ and $id = 6$. This means that whenever one of the model elements with $id = 3$ or $id = 6$ changes, the observer

(runtime model) is notified. The information, which runtime model observes which model elements, is managed by the CDN and stored in a DHT. Listing 5 shows how the generated API can be used to subscribe for model element changes.

Listing 5 Subscription for model changes

```
runtimeModel.subscribeAll(false);
runtimeModel.subscribe(3, new Callback() {
    /* callback code */ };
);
runtimeModel.subscribe(6, new Callback() {
    /* callback code */ };
);
}
```

There are two important concepts to note. First, explicitly subscribing only to elements that are important for the computational node hosting this runtime model reduces the unnecessary propagation of information through the network. Secondly, it can be specified what should happen, *i.e.,* what code should be executed, when the observed change occurs. This allows a reactive programming style by declaratively specifying what should happen if a certain event occurs. As can be seen in Listing 5, the second parameter of the *subscribe* method is a callback, meaning that this is a non-blocking code. Since distributed systems are inherently asynchronous this non-blocking capability is key [146]. Without the support of non-blocking operations this would mean that a computation node is blocked until the awaited event occurs. For example, if a data concentrator intends to read the consumption value of an associated smart meter, with a blocking operation the concentrator (thread) would be blocked until the value arrives. Since this can take several seconds the computation time of the concentrator is wasted and cannot be used for something else in the meantime. Blocking operations are therefore contradictory to the requirement that models@run.time-based systems need to be able to quickly react to state changes.

Current standard modelling frameworks, like EMF, are strictly synchronous. This makes them inappropriate for highly distributed and asynchronous applications. Thus, we completely rewrote the core of KMF to apply a thoroughly reactive and asynchronous programming model. In fact, every method call in KMF is asynchronous and therefore non-blocking. This principle is shown on the right side of Figure 6.4.

The left side of Figure 6.4 shows in comparison a synchronous, *i.e.,* blocking operation call. As can be seen in the figure, non-blocking operation calls do not block the calling process until the triggered operation is finished. Therefore, the caller process can do something else in the meantime. As soon as the operation execution finishes, the caller is notified using a callback. To avoid deeply nested callbacks, which is occasionally referred to as *callback hell* [145], every method call in KMF immediately returns a *KDefer* object, which is similar to a *Future* or *Promise*. These objects enable the definition of a control flow by specifying that the execution of one *KDefer* depends on the result of another *KDefer* and so on. Listing 6 shows how this looks like.

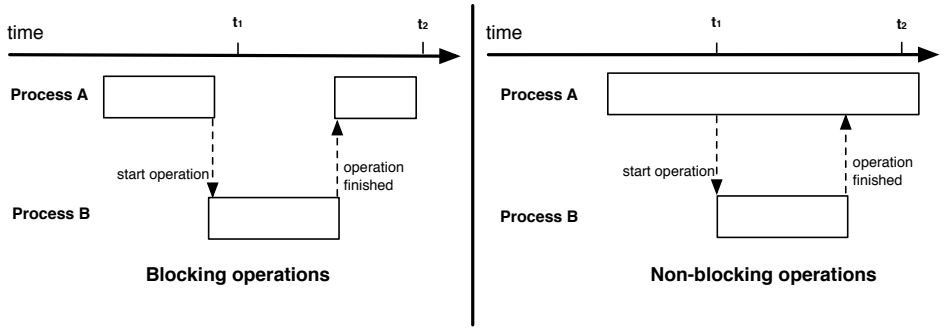


Figure 6.4: Blocking and non-blocking operation calls

Listing 6 Asynchronous method calls with KDefer

```
KDefer filter = runtimeModel.createDefer();
KDefer defer = class2.getClass2s();
filter.wait(defer);
filter.setJob(new KJob() { /* filter class2s */ });
```

Listing 6 filters the results of the getter of *class*2*s*. However, the getter is asynchronous and therefore filtering the result can only start when the getter is executed. This is realised by defining that the filter has to wait for the results of the getter. It is important to note that this is not an active wait. Instead, the control flow immediately continues (non-blocking) and the execution of the filter object is delayed until the getter finishes. To make it easier to traverse models (without the need to deal with callbacks) we define a traversal language on top of KMF. The execution of *KDefer*s are transparently mapped to processes in KMF. A task scheduler system allows to specify a specific strategy. This allows a map-reduce-like [128] approach to horizontally scale by parallelising method calls.

6.3 Evaluation

In this section we evaluate our reactive peer-to-peer distributed models@run.time approach. We show that it can scale to runtime models with millions of elements distributed over thousands of nodes, while the distribution and model access remain fast enough to react in near real-time. Our evaluation is based on a smart grid case study, inspired from a real-world smart grid project. We implemented and integrated our approach into the Kevoree Modeling Framework.

6.3.1 Evaluation setting

We evaluate our approach in terms of its capability to tackle our three main requirements: *i)* large-scale models, *ii)* distributed models, and *iii)* frequently changing models. For this evaluation we use the smart grid model presented in Section 1.2.2 and vary it in size and distribution. Experiments are executed on a 2.6 GHz Core i7 CPU with

16GB RAM and SSD, using the Java version of KMF. We use Docker [27] containers to emulate many different computational nodes. Every presented value is averaged from 10 measurements. The evaluation experiments are available on GitHub[1].

6.3.2 Scalability for large-scale models

In this benchmark we investigate the scalability characteristics of our approach for large-scale models. Therefore, we read a number of model elements and analyse how the performance of this is impacted by the model size. As discussed, complex cyber-physical systems often need to leverage very large models for their reasoning tasks. However, many operations performed on shared models only use a small fraction of the complete model for their reasoning activities. As argued in this chapter, this is often due to the distributed nature of processing tasks. For instance, in the smart grid scenario, every concentrator mainly needs the part of the model representing its district. Therefore, we evaluate the performance of reading a constant number of model elements (25 elements) and analyse how this is impacted by the model size. In this experiment, we increase the number of model elements step by step to more than 1.5 million elements, while the model is distributed over two nodes. In the next subsection we analyse the effect of a highly distributed model. For each model size, we read 25 elements from the model. More specifically, we read consumption values of smart meters in order to approximate the electric loading. The read operations are performed on one node, which communicates through a WebSocket communication protocol with the other node. Since models are composed of object graphs, the performance of read operations usually differs depending if a model is very deep or wide. For this reason, we evaluated both scenarios: once we increased the model size in depth and once in width. Our results are presented in Figure 6.5.

It is important to note that the time to load the model elements is barely affected by the model size. In fact, scalability for models which are large in width, is nearly constant, while for models which are large in depth is nearly linear. In this experiment, we demonstrated that our distributed models@run.time approach allows distributed read operations in an order of magnitude of milliseconds (*between 12 and 28 ms*) in a model with millions of elements. From this experiment, we can conclude that our concept of model elements, which act as proxies on a stream of data chunks, is suitable for large-scale models@run.time.

6.3.3 Scalability for large-scale distribution

In this experiment we investigate the ability of our distributed runtime model approach to collaborate with a huge number of nodes through a shared common context. We evaluate the capacity of the model to propagate changes to a huge amount of collaborating nodes. This large scale distribution is representative for smart grid architectures. To conduct the experiments we used five physical computers (Intel Core i7 with 16GB

[1] https://github.com/kevoree/xp-models15/

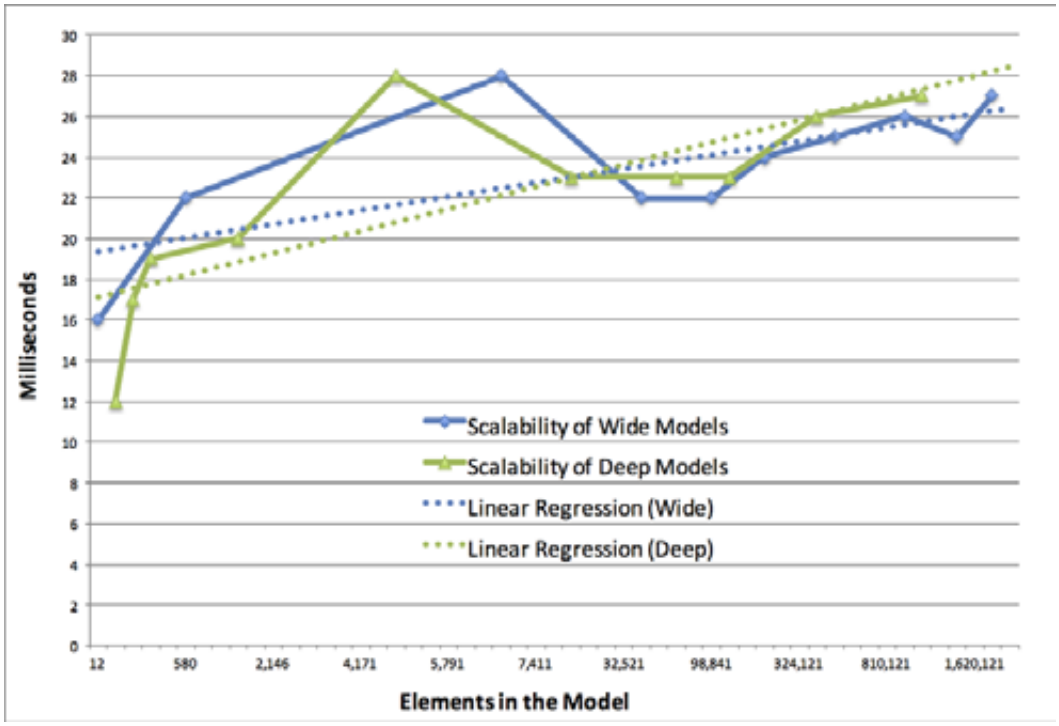


Figure 6.5: Scalability of read operations for large-scale models

RAM), connected through a local area network. On each computer we sequentially started 200 virtual docker nodes using the Decking tool [24] (from 200 up to 1000 nodes) and measured the propagation time of model updates. The 200 docker nodes per physical computer result from a limit of the Linux Kernel. We simulate changes in the smart grid topology, which have to be propagated to all nodes. For this, every five seconds one of the nodes (virtual machines) in the network is updating a value and propagates this change. Changes are propagated using the described observer technique. We measure the required time for propagating the changes. Figure 6.6 shows the results in five scales, represented by the probability spectral density (grouped by 10 ms), reaching from **200** to **1000** collaborating nodes.

The spectral density reflects the probability of each latency depending on the number of nodes. With this benchmark, we demonstrate that our approach can handle a high number of collaborating nodes, while the latency remains low. The raw results are shown in Table 6.1.

6.3.4 Scalability for frequently changing models

CPSs and their associated sensors lead to frequent updates in their associated context models. Therefore, in this benchmark we evaluate the ability of our distributed data stream concept to partly update runtime models with a high frequency. In the following benchmarks we use two nodes, connected through a WebSocket connection. In a first benchmark, we investigate the highest possible volatility of a model element. On a

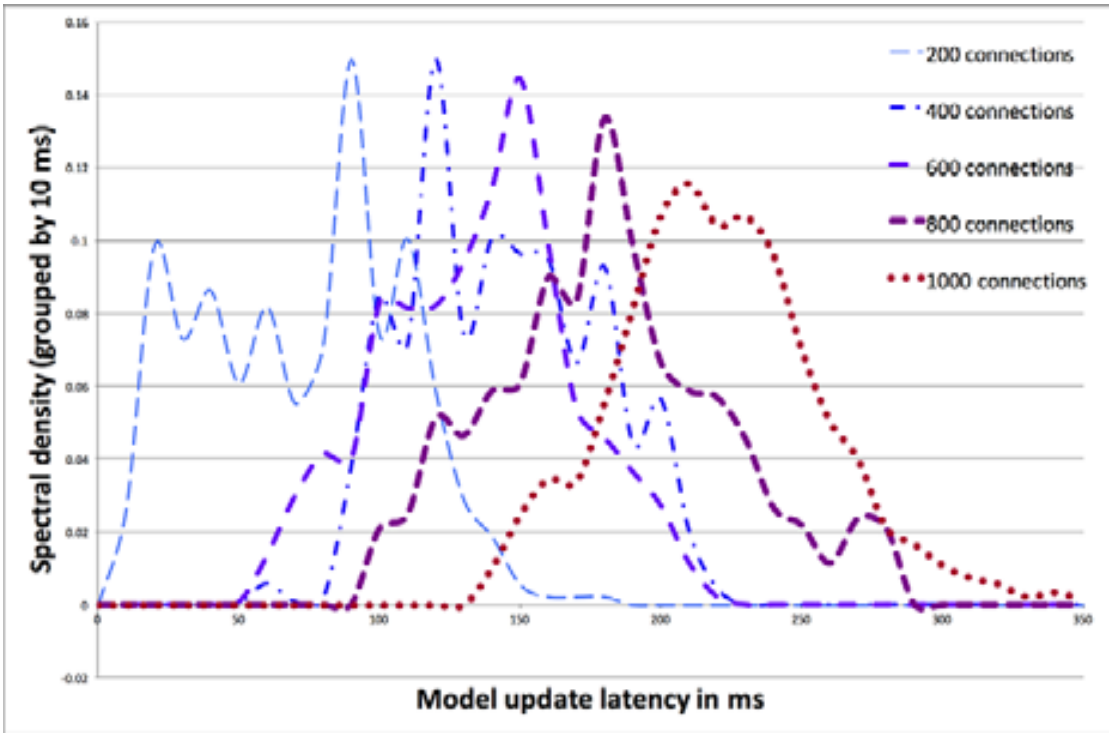


Figure 6.6: Spectral probability density of the model update latency

Table 6.1: Measured latency (in ms) to propagate changes

Nodes Nb.	Min(ms)	Max(ms)	Avg(ms)
200	11	188	88.01
400	63	220	128.75
600	87	253	169.52
800	102	289	185.62
1000	141	355	224.66

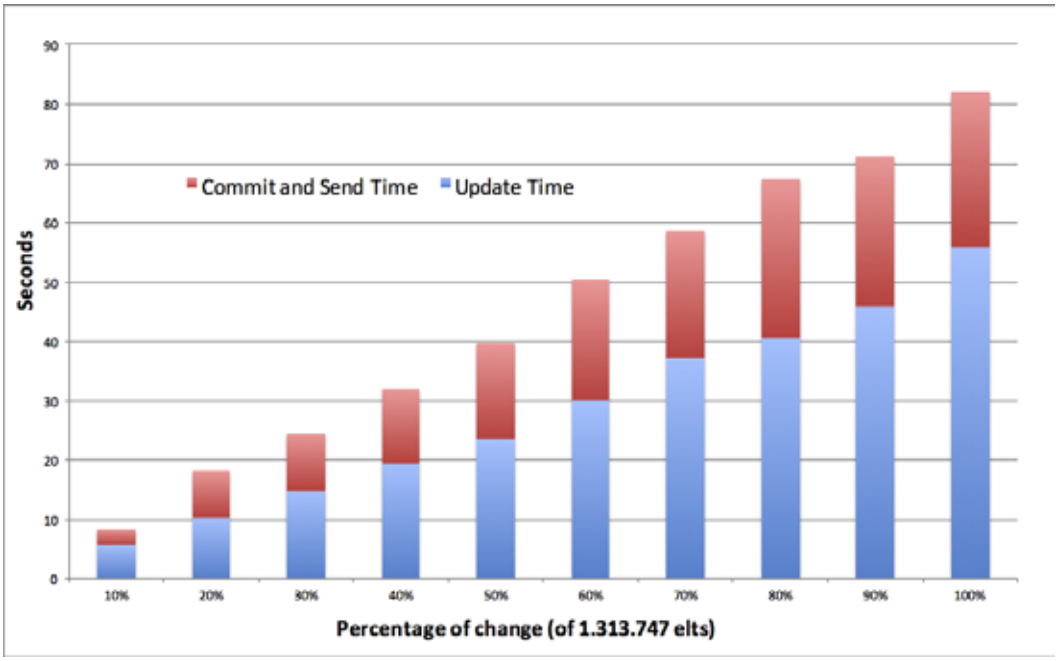


Figure 6.7: Required time for update operations of different size

model with 1.5 million elements, we evaluate how frequently a single element can be updated per second. We evaluated the time to update the value of an attribute on one node and to send an update notification to the node. We measured the maximal frequency (based on the average latency) our implementation is able to handle: **998 updates per second** for a model with 1.5 million elements.

In a second benchmark we evaluated the ability of our approach to handle changes of different size in large models. Therefore, we first updated a small percentage of the model and then increased the percentage of changes step by step. We measured the time needed to update the model and inform the other node about the change (context sharing). Figure 6.7 presents our results.

The results show that our approach approximately scales linear to the percentage of changes. For changes of a small part of the model *(around 10% which is equivalent to 150,000 elements)* our approach remains below 10 seconds. Only if 70% or more of the model changes the update and propagation time exceeds one minute. These results show that our approach is able to handle a high volatility of model elements while still offering good latency properties.

6.4 Discussion: distribution and asynchronicity

The border between large-scale data management systems and models is becoming more and more blurry as models@run.time progressively gains maturity through large-scale and distribution mechanisms. It is clearly important to evaluate the limits and the potential reuse of each domain. In this context, for example, Gwendal *et al.,* [124]

present a framework to handle complex queries on large models by translating object constraint language (OCL) queries to Tinkerpop Gremlin. Despite the feasibility of mapping models@run.time into distributed databases, which takes care of data replication, it quickly leads to many limitations in practice. Indeed, to mimic synchronous calls, heavy and costly distributed algorithms have to be involved, such as consensus or RAFT. However, because every communication can fail, the uncertainty is at the heart of the distribution. Instead of hiding it, most of nowadays software stacks exploit explicit asynchronous programming to scale their distributed computations. In this trend we can mention the well-known asynchronous JavaScript and XML (AJAX), the API of NodeJS server, or distributed P2P communication, which are by default asynchronous. Therefore, beyond the ability to distribute in a scalable manner models@run.time over nodes, our contribution also includes, at its core, an asynchronous layer in models. We are convinced that this change is inescapable if runtime models should go beyond the barrier of computer memory in order to exploit the power of distributed systems. Moreover, asynchronous modelling can pave the way to define the semantic of partial and large-scale models. Benelallam *et al.,* [90] also identify the efficient processing of very large models as one key requirement for the adoption of MDE in industrial contexts. The presented techniques can also be useful in the context of collaborative (meta) modelling, as for example discussed in [189] and in [79].

6.5 Conclusion

Cyber-physical systems, such as smart grids, are becoming more and more complex and distributed. Despite the fact that models@run.time enable the abstraction of such complex systems during runtime and to reason about it, the combination of the *i)* large-scale, *ii)* distributed, and *iii)* constantly changing nature of these systems is a big challenge. These characteristics are closely interlinked: The increasing complexity of cyber-physical systems naturally leads to bigger models, which are—due to their size—also more difficult to distribute or replicate. Finally, the distributed aspect inherently leads to asynchronicity and this in turn requires the ability to dynamically react to events, instead of actively waiting. This chapter introduced a distributed models@run.time approach, combining ideas from reactive programming, peer-to-peer distribution, and large-scale models@run.time. First, since models@run.time are continuously updated during the execution of a system, they cannot be considered as bounded but can change and grow indefinitely. We defined models as observable streams of model chunks. This stream-based interpretation of models allows to process models chunk by chunk regardless of their global size. Secondly, we distribute and exchange these model chunks between computational nodes in a peer-to-peer manner and on-demand to avoid the necessity to exchange full runtime models. The use of a lazy loading strategy, allows to transparently access the complete virtual model from every node, although chunks are actually distributed across nodes. Thirdly, we leverage observers, an automatic reloading mechanism of model chunks (in case of changes), and asynchronous operations to enable a reactive programming style, allowing a system to dynamically react to context changes. We integrated our approach into the Kevoree Modeling Framework and evaluated it on an industrial-scale smart grid case study. We demonstrated that this approach can enable frequently changing, reactive

distributed models and can scale to millions of elements distributed over thousands of nodes, while the distribution and model access remains fast enough to enable reactive systems. The experimental validation showed that our peer-to-peer distribution and stream processing model for our proposed multi-dimensional graph data model allows to efficiently reason over distributed, frequently changing data.

7

Weaving machine learning into data modelling

This chapter presents an approach to weave machine learning directly into data modelling. It suggests to decompose learning into reusable, chainable, and independently computable micro learning units, which are modelled together with and on the same level as domain data. This allows to seamlessly integrate domain knowledge and machine learning into the data model introduced in the previous chapters.

This chapter is based on the work that has been presented in the following papers:

- under submission at International Journal on Software and Systems Modeling (SoSyM): Thomas Hartmann, Assaad Moawad, Francois Fouquet, and Yves Le Traon. The Next Evolution of MDE: A Seamless Integration of Machine Learning into Domain Modeling
- Thomas Hartmann, Assaad Moawad, François Fouquet, Yves Reckinger, Tejeddine Mouelhi, Jacques Klein, and Yves Le Traon. Suspicious electric consumption detection based on multi-profiling using live machine learning. In *2015 IEEE International Conference on Smart Grid Communications, SmartGridComm 2015, Miami, USA, November 2-5, 2015*

Contents

7.1 Introduction

In order to meet future needs, software systems need to become increasingly intelligent. A prominent example are cyber-physical systems and Internet of Things applications. Advances in software, embedded systems, sensors, and networking technologies have led to a new generation of systems with highly integrated computational and physical capabilities, which nowadays are playing an important role in controlling critical infrastructures, like the power grid. Such systems face many predictable situations for which behavioural models can be already defined at design time of the system. In order to react to critical overload situations, for example, the maximum allowed load for customers can be restricted. This is called *known domain knowledge*. In addition, intelligent systems have to face events that are unpredictable at design time. For instance, the electric consumption of a house depends on the number of persons living there, their activities, weather conditions, used devices, and so forth. Although such behaviour is unpredictable at design time, it is identifiable and a hypothesis about it can be already formulated and solved later by observing past situations, once data becomes available. Sutcliffe *et al.,* [298] suggest to call this *known unknown*.

To make smart decisions, intelligent systems have to continuously refine behavioural models that are known at design time with what can be learned only from live data, to solve known unknowns.

The goal of the contribution behind this chapter is to combine both into a single data model—more specifically, into the proposed multi-dimensional graph data model introduced in the previous chapters. This would allow to seamlessly integrate machine learning into this model, in a way that parts of the model are learned, others are computed, based on domain knowledge, and some others are measured.

Machine learning algorithms can help to solve unknown behaviours by extracting commonalities over massive datasets. Peter Norvig describes machine learning and artificial intelligence as "*getting a computer to do the right thing when you don't know what that might be*" [251]. Learning algorithms can infer behavioural models based on past situations, which represent the learned common behaviour. However, in cases where datasets are composed of independent entities which behave very differently, finding one coarse-grained common behavioural model can be difficult or even inappropriate. This applies particularly for the domain of cyber-physical systems and IoT. For example, considering the electrical grid, the consumption of a factory follows a very different pattern than the consumption of an apartment. Searching for commonalities between these entities is not helpful. To sum up, coarse-grained learning alone, which is based on the "*law of large numbers*", can be inaccurate for systems which are composed of several elements, which behave very differently.

Instead, following a *divide and conquer* strategy, learning on finer granularities can be considerably more efficient for such problems [330], [144]. This is, for example, used in text, sentiment [205], and code analysis [219], where a segmentation by the domain of words can help to reduce complexity. Similarly, multi-granular representations [331] have been applied to solve hierarchical or micro-array-based [135] learning problems. Aggregating simple learning units [255] has been successfully used to build probabilistic

prediction models [114]. In accordance to the pedagogical concept [184], we refer to small fine-grained learning units as **micro learning**. We claim that micro learning is appropriate to solve the various known unknown behavioural models in systems which are composed of several elements, which behave very diverse and can be significantly more accurate than coarse-grained learning approaches.

Applying micro learning on systems, such as the electric grid, can potentially lead to many fine-grained learning units. Furthermore, they must be synchronised and composed to express more complex behavioural models. Therefore, an appropriate structure to model learning units and their relationships to domain knowledge is required. Frameworks like TensorFlow [69], GraphLab [229] or Infer.NET [95] also divide machine learning tasks into reusable pieces, structured with a model. They propose a higher level abstraction to model the learning flow itself by structuring various reusable and generic learning subtasks. While these approaches solve the complexity of complicated learning tasks, they focus only on modelling the learning flow, without any relation to domain data and its structure. This makes it necessary to express domain knowledge and data in different models, using different languages and tools and leads to a separation of domain data, knowledge, known unknowns, and associated learning methods. This requires a complex mapping between these concepts for every micro learning unit. A similar conclusion has been drawn by Vierhauser *et al.,* [313] for monitoring system of systems.

To address this complexity, in this chapter we propose to weave micro machine learning seamlessly into data modelling.

Specifically, our approach aims at:

- Decomposing and structuring complex learning tasks with reusable, chainable, and independently computable micro learning units to achieve a higher accuracy compared to coarse-grained learning.
- Seamlessly integrating behavioural models, which are known at design time, behavioural models that need to be learned at runtime, and domain models in a single model, expressed with one modeling language using the same modeling concepts.
- Automating the mapping between the mathematical representation, expected by a specific machine learning algorithm, and the domain representation [95] and independently updating micro learning units to be fast enough to be used for live learning.

We take advantage of the modelled relationships between domain data and behavioural models (learned or known at design time), which implicitly define a fine-grained mapping of learning units and domain data.

Let us consider a concrete use case. In order to be able to detect anomalies and predict potential problems, like electric overload, before they actually happen, various data collected in a smart grid, *e.g.,* consumption data, must be monitored and profiled.

The important smart grid entities for the context of this chapter are *smart meters* and *concentrators* (*cf.* Section 1.2.2).

For various tasks, like electric load prediction or detection of suspicious consumption values, customers' consumption data need to be profiled independently and in real time. This is challenging due to performance requirements but also due to the large number of profiles, which need to be synchronised for every new value. To model such scenarios, we need to express a relation from a machine learning profiler to the consumption of a customer. Since the connections from smart meters to concentrators vary over time, a concentrator profiler depends on the profiles of the currently connected meters. A coarse-grained profiler at the concentrator level will not take into account the connection changes at real-time and their implications in predicting the electric load. Coarse-grained profiling alone can be very inaccurate in such cases.

Another example where micro learning and composing complex learning from smaller units can be significantly more accurate than coarse-grained learning are recommender systems. These can be composed from one micro learning unit per customer and one micro learning per product. Again, using only coarse-grained profiles for customers and products can be inaccurate. In case of recommender systems, micro learning can even be combined with coarse-grained learning by taking in addition to the individual profiles of customers and products also the coarse-grained profiles of every purchase into account.

The bottom line is that micro learning units and combining them to larger learning tasks is especially useful for systems which are composed of multiple independent entities which behave very differently. CPSs and IoT systems are domains where these characteristics apply specifically.

We evaluate our approach on the smart grid case study and show that:

- Micro machine learning for such scenarios can be more accurate than coarse-grained learning.
- The performance is fast enough to be used for live learning.

We implemented and integrated our approach into the open-source modelling framework KMF, which was specifically designed for the requirements of CPSs and IoT. More specifically, we integrated this approach into the multi-dimensional graph data model, proposed in this thesis (*cf.* Chapter 4 and Chapter 5), which is the core data structure underlying KMF.

The remainder of this chapter is as follows. Section 7.2 presents our model-based micro machine learning approach. We discuss the meta model definition used in our approach and present a modelling language to seamlessly model machine learning and domain data. In Section 7.3 we evaluate our approach on a smart grid case study, followed by a discussion in Section 7.4. A conclusion is presented in Section 7.5.

7.2 Combining learning and domain modelling

In this section we first discuss the objectives of our approach. Then, we present the meta model definition (meta meta model) which we use for the implementation of our approach and detail what exactly micro learning units are. Next, we present the syntax and semantic of our modelling language and show concrete examples of its usage. This section ends with presenting important implementation details.

7.2.1 Objectives

In order to weave micro machine learning into domain modelling we need to extend modelling languages to model learned attributes and relations and "default" ones seamlessly together. It requires modelling languages to allow to specify in a fine-grained way, *what* should be learned, *how* (algorithm, parameters) something should be learned, and *from what* (attributes, relations, learned attributes, learned relations) something should be learned. To be appropriate for live learning, this fine-grained learning units need to be independently computable and updatable.

We use a meta meta model to define this weaving. A meta meta model specifies the concepts which can be expressed in a concrete meta model, *i.e.,* it specifies what can be expressed in meta models conforming to it. This allows domain modes to express learning problems. Based on this, we can define a concrete modelling language, providing the necessary constructs to weave machine learning into domain modelling.

7.2.2 Meta meta model

We first specify the meta model definition (meta meta model) underlying our approach. This definition, shown in Figure 7.1, is inspired by MOF/EMOF and extended with concepts to express machine learning directly in the domain modelling language. Section 7.2.4 describes the modelling language we built around this meta meta model and defines the formal semantic and syntax of the language. Elements related to machine learning are depicted in the figure in light grey. We focus on these elements since other parts comply with standard meta model definitions, like EMOF or MOF. As can be seen in the figure, we define meta models consisting of an arbitrary number of meta classes and enums. Meta classes in turn have an arbitrary number of properties. Properties are attributes, relations, or what we call *"specified properties"*. Specified properties are either *"learned properties"* or *"derived properties"*. Learned properties are relations or attributes, which will be learned by a specific machine learning algorithm. A concrete learning algorithm can be specified with the specification *"using"*. Parameters for the learning algorithm can be defined with the specification *"parameter"*. The *"feature"* specification allows to access properties from other meta classes or enums.

Derived properties are similar to learned properties, however derived properties don't

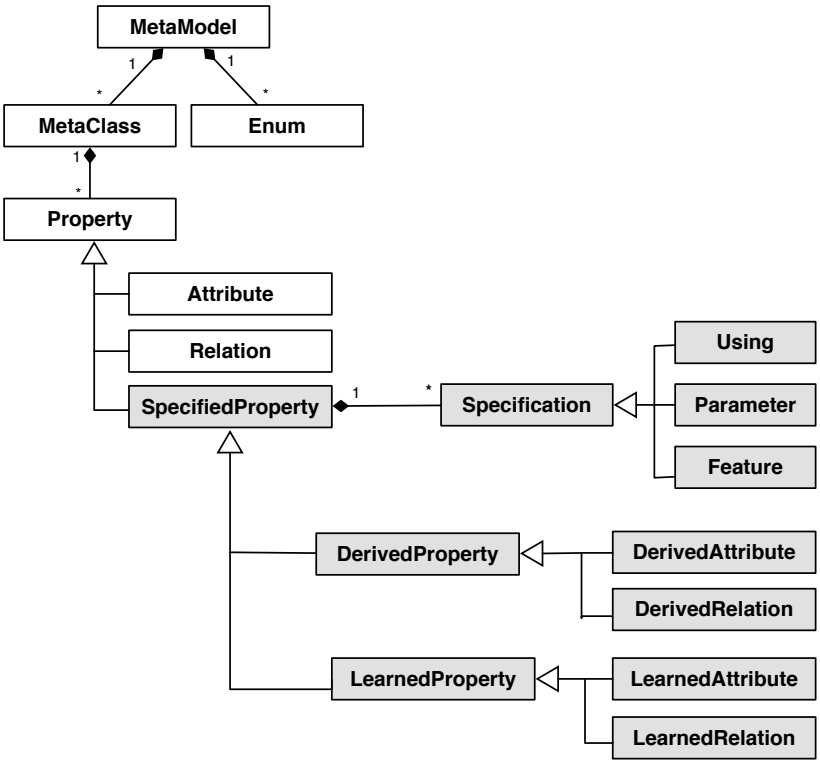


Figure 7.1: Meta meta model

have a state associated, *i.e.,* they don't need to be trained but simply compute a value. The value of a derived attribute is calculated from the values of attributes of other meta classes. Whereas the value of a learned attribute depends on a state and past executions, *i.e.,* on learning. As we will see in Section 7.2.6, this is reflected by the fact that for derived properties we only generate so-called *"infer"* methods whereas for learned properties we generate *"learn"* and *"infer"* methods.

7.2.3 Micro learning units

The core elements of our approach are micro learning units. As explained in Section 7.1 we use the term "micro learning unit" to refer to small fine-grained learning units. These units are designed to decompose and structure complex learning tasks with reusable, chainable, and independently computable elements. Figure 7.2 illustrates a concrete example of a micro learning unit and set it into relation to the meta and instance levels. In the top left of the figure we see the definition of a `SmartMeter` meta class. Besides two attributes, `activeEnergy` and `reactiveEnergy`, one derived property named `aboveThreshold` and one learned property named `powerProbabilities` are defined. As will be detailed in Section 7.2.6, specifying the learned property `powerProbabilities` results in automatically generating the necessary code for the mapping between the internal representation of a machine learning algorithm and domain models. The machine learning algorithm will be "weaved" inside the meta model instances, in this case of `SmartMeter` instances. As illustrated, the micro learning unit

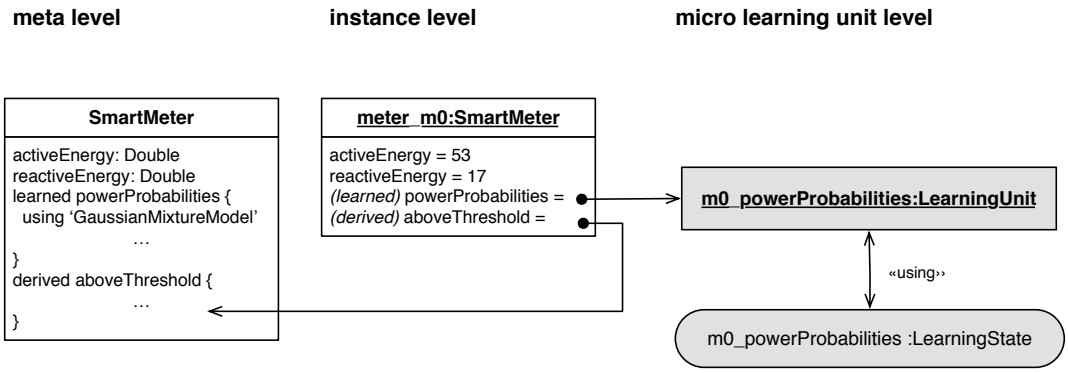


Figure 7.2: Schematic representation of a micro learning unit

is an instance of a learning algorithm, contained in an object and related to a state. It is also related to the instance of the `SmartMeter` class, or more specifically to the learned attribute. In fact, every instance of a `SmartMeter` class has its own (automatically generated) instance of a micro learning unit. Technically, micro learning units are realised as special nodes of the multi-dimensional graph data model (*cf.* Chapter 5). Like any other node in this data model, micro learning units are therefore mapped to one or several (taking the time and alternative world dimensions into account) state chunks.

As can be seen in the figure, machine learning (via learned properties) can be seamlessly integrated and mixed with domain modelling. Section 7.2.4 presents our proposed modelling language and details how this can be defined within the concrete syntax of this language. The resultant ability to seamlessly define relationships from learned properties to domain properties and to other learned properties—and vice versa from domain properties to learned properties—enables composition, reusability, and independent computability/updates of micro learning units. An additional advantage of independent micro learning units is that they can be computed in a distributed way. Basically, every learning unit can be computed on a separate machine. Such distribution strategy relies on a shared model state, as for example presented in [177]. The computation can then be triggered in a BSP way [156] over this shared state.

Our approach is built in a way that the same learning models can be used in several tasks without the need to duplicate it. For example, in the smart metering domain, the electricity consumption profile of a user can be used to: predict the electrical load, classify users according to their profile, or to detect suspicious consumption behaviour. The possibility to compose micro learning units allows a segregation of learning concerns. In case an application requires a combination of different machine learning techniques, it is not necessary to mash traditional algorithms for each step together. Instead, independent micro learning units can be composed in a divide-and-conquer manner to solve more complex learning problems. This is shown in more detail in Section 7.2.5. In addition, the learning algorithm itself is encapsulated and the mapping between the domain model and the data representation expected by the respective learning algorithm is automatically generated. In this way, the learning algorithm can be easily changed without the need to change the interface for the domain application.

The possibility to derive attributes from others, allows to create richer models. In fact, *ensemble methods* in the machine learning domain, derive stronger machine learning models from weaker machine learning models by combining the results of the smaller units. In our data model, we enable ensemble methods from several learned attributes (learnt through different weaker machine learning models) by creating a derived attribute that combines their results.

Even though our approach promotes micro learning, there are nonetheless scenarios where it is helpful to also learn coarse-grained behaviour, *e.g.,* the consumption profile of all customers. Therefore, we allow to specify a scope for learned properties. The default scope is called `local` and means that the learning unit operates on an per instance level. For coarse-grained learning we offer a `global` scope, which means that the learning unit operates on a per class level, *i.e.,* on all instances of the specified class.

7.2.4 Modelling language

In this section we introduce our modelling language to enable a seamless definition of domain data, its structure, and associated learning units. This language is inspired by the state of the art in meta modelling languages (*e.g.,* UML [253], SysML [164], EMF Ecore [104]). The semantic of the language follows the one of UML class diagrams extended with learning units. Many modelling languages, like UML, are graphical. Advantages of graphical modelling languages are usually a flatter learning curve and better readability compared to textual modelling languages. On the other hand, textual modelling languages are often faster to work with, especially for experts. Also, editors and programming environments are easier to develop and less resource hungry for textual languages. A recent study of Ottensooser *et al.,* [258] showed that complex processes and dependencies are more efficient to express in a textual syntax than a graphical one. For these reasons we decided to first implement a textual modelling language. For future work we plan to propose an additional graphical modelling language. The multi-dimensional graph data model (*cf.* Chapter 5), proposed in this dissertation, is then generated based on the model definition expressed in this language.

In the following, we first present the semantic of the language followed by a definition of its syntax. Then, we illustrate by means of the concrete smart grid use case how this language can be used to express different combinations of machine learning and domain modelling.

7.2.4.1 Semantic

Our modelling language follows the formal descriptive semantic and axioms of UML class diagrams, as defined in [333]. We first present the necessary formalism of UML class diagrams and then extend this formalism to include axioms for weaving learned and derived properties into our language.

Definition 18 *Let* $\{C_1, C_2, ..., C_n\}$ *be the set of concrete meta classes in the meta model, we have* $\forall x\ (C_1(x) \vee C_2(x) \vee ... \vee C_n(x))$ *is an axiom, where* $\forall x, \forall i, j, (C_i(x) = C_j(x)) \Rightarrow C(i = j)$

In this definition we state that any object x should be at least (inheritance) an instance of one of the meta classes defined in the meta model. Additionally, given an object x all meta classes verifying $C(x)$ should be linked by a relationship of inheritance following classical UML semantics and as defined in [333]. This inheritance model is not described here for sake of simplicity and to keep the emphasis on learning aspects.

Definition 19 *For each meta attribute att of type T in C, we have:* $\forall x, y\ C(x) \wedge (att(x, y) \rightarrow T(y))$ *is an axiom.*

In the second definition, we are stating that if x is an instance of a meta class C, which has a certain meta attribute *att* of type T, the value y of this meta attribute should always be of type T.

Definition 20 *For each relationship rel from meta class* C_1 *to another meta class* C_2*, we have:*
$\forall x, y\ (C_1(x) \wedge rel(x, y)) \rightarrow C_2(y)$ *is an axiom.*

In this definition, if a meta class C_1 has a relationship *rel* to a meta class C_2, and x is an instance of C_1, having a relation *rel* to y, this implies that y should be an instance of C_2.

Definition 21 *For each relationship rel from meta class* C_1 *to* C_2*, if* $'e_1..e_2'$ *is its multiplicity value, we have:*
$\forall x\ C_1(x) \rightarrow (e_1 \leq ||y|rel(x, y)|| \leq e_2)$ *is an axiom.*
Similarly, for each meta attribute att in C_1*, if* $'e_1..e_2'$ *is its multiplicity value, we have:*
$\forall x\ C_1(x) \rightarrow (e_1 \leq ||y|att(C_1, x) = y|| \leq e_2)$ *is an axiom.*

In Definition 21, we state that an attribute or a relationship can have minimum and maximum bounds defined in the meta model, and any instance of the meta class should have its attributes and relationships respecting these bounds.

Following the same approach, we extend the classical UML definition of meta class, by adding two new kinds of properties: learned and derived attributes and relations. In particular, a meta learned attribute *learnedatt*, in a meta class C, is a typed attribute of a type T that represents a known unknown in the business domain. It is learned using a machine learning hypothesis. This hypothesis can be created from a parametrized machine learning algorithm, its parameters, a set of features extracted from the business domain, and a past learned state that represents the best fitted model of the learning algorithm to domain data. A meta derived attribute *derivedatt*, is very similar to the *learnedatt* with the only difference that the deriving algorithm

does not depend on a past state but only on extracted features. In other terms, a meta derived attribute, has a type T, a set of extracted features, a deriving parametrised algorithm and its parameters. The same definition applies for learned and derived relations that behave in the same manner than attributes with only a different result type (*e.g.,* collection of nodes as output).

A step called **feature selection** in the meta modelling of C_x is required in order to specify the dependencies needed in order to learn *learnedatt* or derive *derivedatt*. The feature selection can be done only over meta attributes reachable within the host meta class C_x. We define this reachability operation by the following:

Definition 22 $reach\colon (metaClass \times metaAtt) \mapsto boolean$
$reach(C_x, a) = att(C_x, a) \vee learnedatt(C_x, a) \vee derivedatt(C_x, a)$
$\vee(\exists C_y | rel(C_x, C_y) \wedge reach(C_y, a))$

In this definition, a meta attribute a is considered as reachable from a meta class C_x, either if it is a meta attribute, meta learned attribute, or meta derived attribute within the meta class C_x itself, or if C_x has a relationship to another class C_y, which contains a or it can be reachable from there, using recursively another relationship.

Definition 23 *Let F be the set of features to extract in order to learn learnedatt in a meta class C, we have:*
$\forall f \in F, (f! = learnedatt) \wedge reach(C, f)$ *is an axiom.*
Similarly, in order to derive derivedatt, we have:
$\forall f \in F, (f! = derivedatt) \wedge reach(C, f)$ *is an axiom.*

In other words, a meta learned or derived attribute can extract their features from the meta attributes defined within the meta class C (except itself to avoid circular reasoning) or reachable from its relationships in a recursive way.

Definition 24 *To summarize, a meta learned attribute learnedatt has a type T, a set of feature extractions F, a parameterised learning algorithm $alg_{p_1,...,p_n}$, a set of parameters $p_1, ..., p_n$, and an learned state LS.*
Moreover, we have: $\forall x, y\ C(x) \wedge (learnedatt(x, y) \rightarrow T(y))$
$\wedge\ y = alg_{p_1,...,p_n}(eval(F), LS)$ *is an axiom.*

Similarly, a meta derived attribute derivedatt has a type T, a set of feature extractions F, a parameterised learning algorithm $alg_{p_1,...,p_n}$, a set of parameters $p_1, ..., p_n$.
We have: $\forall x, y\ C(x) \wedge (derivedatt(x, y) \rightarrow T(y))$
$\wedge\ y = alg_{p_1,...,p_n}(eval(F))$ *is an axiom.*

In Definition 24, we present that the meta learned or derived attribute is typed in the same manner of classical meta attributes (Definition 19), and the type has to be always respected. By extension, learned and derived relations follow strictly the same definition than learned and derived attributes and therefore will not be repeated

here. Moreover, the learned attributed is calculated by executing the parameterised learning algorithm over the extracted features and the learned state. For the derived attributed, it is calculated by executing the parameterised deriving algorithm over only the extracted features. Both learned and derived properties are considered as specified properties, because they require some specifications (features, parameters, algorithm), in order to be calculated. This is depicted in our meta meta model in Figure 7.1. Finally, at an instance level, an object state is composed by the state of its classical attributes, relationships, and the states of each of its learned attributes.

As our model has a temporal dimension, every meta attribute has a time dimension, and by extension, the learned state has as well a temporal dimension. All meta attributes, relationships, states, and parameters are replaced by their temporal representation (For example: $att \mapsto att(t)$). For feature extraction, it is possible to extract the same attributes but coming from different points in time as long as the attributes are reachable.

7.2.4.2 Syntax

The syntax of our textual modelling language is inspired by Emfatic [123] and is an extension of the language defined in [150]. Listing 7 shows its formal grammar. The parts in italic show the language extensions.

Listing 7 Grammar of our modelling language

```
metaModel ::= (class | enum)*
enum ::= 'enum' ID '{' ID (',' ID)* '}'
class ::= 'class' ID parent? '{' property* '}'
property ::= annot* ( 'att' | 'rel' ) ID : ID spec?
parent ::= 'extends' ID (',' ID)*
annot ::= ( 'learned' | 'derived' | 'global' )
spec ::= '{' (feature | using | param )* '}'
param ::= 'with' ID ( STRING | NUMBER )
feature ::= 'from' STRING
using ::= 'using' STRING
```

This grammar reflects the classic structure of object-oriented programs. Multiplicities of relationships (indicated by the keyword `rel` are by default unbounded, *i.e.,* too many. Explicit multiplicities can be defined using the `with` clause, *e.g.,* `with maxBound *` or `with minBounds 1`. Meta models are specified as a list of meta classes (and enums). `Classes`, `Enums` and their `Properties` are defined similar to Emfatic. To distinguish static, learned, and derived properties, we introduce annotations for attribute and relation definitions. In addition to this, a specification block can optionally refine the behaviour expected from the corresponding property. A specification can contain statements to declare the algorithm to use, feature extraction functions, and meta parameters to configure the used algorithms. Feature extraction statements are using string literals where a OCL-like notation [49] is used to navigate to reachable properties.

7.2.5 Model learning patterns

Similarly to how modelling methodologies have led to design patterns to solve common problems, in this subsection we describe patterns to weave machine learning into models. We describe how our language can be used on the concrete smart grid use case with different combinations of machine learning and domain modelling. The section starts with a simple domain model, then explains different combinations of domain data and learning, and ends with a more complex example on how different learnings can be composed.

7.2.5.1 Weaving learned attributes into domain classes

Let's start with a simple example. Listing 8 shows the definition of a class `SmartMeter`. It contains two attributes `activeEnergy` and `reactiveEnergy` and a relation to a `customer`. These are the typical domain attributes, defining a `SmartMeter` class.

In this class, we define a learned attribute `anomaly` that automatically detects abnormal behaviour, based on profiling active and reactive energy. To do so, we specify to use a Gaussian anomaly detection algorithm as learning algorithm. In this example, the attribute `anomaly` can be seamlessly accessed from all `SmartMeter` instances. In fact, the attribute can be used similar to "normal" ones (*i.e.,* not learned), however, instead of the default getter and setter methods, the generated API will provide a `train` and an `infer` method. This example shows how learned attributes can be seamlessly woven into domain classes.

Listing 8 Meta model of a smart meter with anomaly detection

```
class SmartMeter {
        att activeEnergy: Double
        att reactiveEnergy: Double
        rel customer: Customer
        learned att anomaly: Boolean {
          from "activeEnergy"
          from "reactiveEnergy"
          using "GaussianAnomalyDetection"
        }
}
```

7.2.5.2 Defining a learning scope for coarse-grained learning in domain models

Listing 9 shows an example of a power classification problem. In this listing, first an enumeration `ConsumptionType` with three categories of consumption types (low, medium and high) is defined. Then, we extend the class `SmartMeter` to add a global `classify` attribute, which classifies users according to their consumption behaviours. It learns from `activeEnergy`, `reactiveEnergy`, and `nbResidents`.

This example shows coarse-grained learning, where all instances of a domain class contribute to one learning unit. It demonstrates that attribute extractions cannot only happen at the level of attributes of the current instance but also to any reachable attribute from the relation of the current instance. In this example, the attribute `nbResidents`, which is the number of residents within the household of each customer, is extracted from a concrete `Customer` instance of a concrete `SmartMeter` instance. Moreover it shows how to specify the machine learning hyper-parameters (here the learning rate and regularization rate) within the learned attribute using the keyword `with`.

Listing 9 Meta model of a power classifier

```
enum ConsumptionType { LOW, MEDIUM, HIGH }

class SmartMeter{
        [...]
        global learned att classify: ConsumptionType {
          from "customer.nbResidents"
          from "activeEnergy"
          from "reactiveEnergy"
          with learningRate 0.001
          with regularizationRate 0.003
          using "LinearClassifier"
        }
}
```

7.2.5.3 Modelling relations between learning units and domain classes

Listing 10 shows the meta class of a `SmartMeterProfiler`. In a first step, we define that such profilers have relationships to `SmartMeter` instances and vice versa. Then, we extract several attributes from this relationship. For instance, we get the hour of the day, the active and reactive energy, and calculate the square value. Attribute extractions can be any mathematical operations over the attributes that are reachable from the relationships defined within the class. In this example, the profiler learns the probabilities of the different power consumptions, hourly based, using a Gaussian mixture model algorithm [179].

7.2.5.4 Decomposing complex learning tasks into several micro learning units

For the last example, we show how to use domain information to derive an advanced profiler at the concentrator level, using the fine-grained profilers at the smart meters. First, we define a class `Concentrator` that contains relations to the connected smart meters. Then, we define a `ConcentratorProfiler` with a relation to a `Concentrator` and vice versa. Inside this profiler, we derive an attribute `powerProbabilities` using

Listing 10 Meta model of a smart meter profiler

```
class SmartMeterProfiler {
        rel smartMeter: SmartMeter with minBound 1 with maxBound 1
        learned att powerProbabilities: Double[] {
          from "Hour(smartMeter.time)"
          from "smartMeter.activeEnergy^2"
          from "smartMeter.reactiveEnergy^2"
          using "GaussianMixtureModel"
        }
}

class SmartMeter {
        [...]
        rel profile: SmartMeterProfiler
}
```

the keyword `derived` and using an `aggregation` function that combines the probabilities from the fine-grained profiles. This example shows how fine-grained machine learning units can be combined to larger machine learning units.

Listing 11 Meta model of a concentrator and its profiler

```
class Concentrator {
        rel connectedSmartMeters: SmartMeter
        rel profile:ConcentratorProfiler
}

class ConcentratorProfiler {
        rel concentrator: Concentrator
        derived att powerProbabilities: Double[] {
          from concentrator.connectedSmartMeters.profile
          using "aggregation"
        }
}
```

7.2.6 Framework implementation details

Our approach is implemented into the Kevoree Modeling Framework and via plugins integrated as a full modelling environment into IntelliJ IDE[1]. The development process with KMF follows a classical MDE approach, starting with a meta model definition. The complete *LL* grammar of our extended modelling language is available as open-source[2]. KMF contains a code generator, based on Apache Velocity [64], to generate APIs for object-oriented languages. Currently, our generator targets Java and TypeScript.

[1] https://www.jetbrains.com/idea/
[2] https://github.com/kevoree-modeling/dsl

The generated classes can be compared to what is generated by frameworks like EMF. In the following, we focus on the machine learning extensions. According to what is defined in the meta model, our code generator "weaves" the concrete machine learning algorithms into the generated classes and also generates the necessary code to map from a domain representation (domain objects and types) to the internal mathematical representation expected by the learning algorithms (double arrays, matrices, etc) and vice versa. Various machine learning algorithms can be integrated in our framework. Currently, we implemented the following algorithms:

- *Regression*: Live linear regression
- *Classification*: Live decision trees, Naive Bayesian models, Gaussian Bayesian models
- *Clustering*: KNN, StreamKM++
- *Profiling*: Gaussian Mixture Models (Simple & Multinomial)

For every derived property our generator adds an `infer` method to the generated class, which contains the code to compute the property according to its meta model definition. Similar, for every learned property our generator adds an `infer` to read the state of the learning unit and a `train` method to trigger the injected learning algorithm.

Since KMF targets CPSs and IoT applications it has a strong focus on performance. The core data structure underlying KMF is the multi-dimensional graph data model introduced in Chapter 5.

Since relationships between domain classes and micro learning units are explicitly defined, they can be used during runtime to infer for which changes a micro learning unit needs to be recomputed. This is realised using change listeners and an asynchronous message bus. As a result, our framework supports fully independent updates of learning units. Leveraging the underlying multi-dimensional graph data model this can even be done in a distributed manner.

7.3 Evaluation

In this section we evaluate our approach based on two key performance indicators: *1)* can micro machine learning be more accurate than coarse-grained learning and *2)* is the performance of using micro machine learning fast enough to be used for live learning and thus for live analytics.

7.3.1 Experimental Setup

We evaluate our approach on the smart grid use case. We implemented a prediction engine for customers' consumption behaviour using our modelling framework. This

engine predicts the consumption behaviour based on live measurements coming from smart meters. We implemented this evaluation twice, once with a classical coarse-grained approach and another time with our micro learning based-approach. The goal is to demonstrate that our micro learning-based approach can be more accurate while remaining fast enough to be used for live learning.

For our experiments we consider 2 concentrators and 300 smart meters. We use publicly available smart meter data from households in London[3]. The reason why we use publicly available data for this experiment, instead of data from our industrial partner Creos Luxembourg S.A., is that this data is confidential what would prohibit to publish this data for reproducibility. Our experiments are based on 7,131,766 power records, from where we use 6,389,194 records for training and 742,572 records for testing. The used training period is 15/08/2012 to 21/11/2013 and the testing period from 21/11/2013 to 8/01/2014.

For the first experiment, we use a coarse-grained profiler on the concentrators. All smart meters send their data regularly to concentrators where the sum of all connected smart meters is profiled. In a second experiment we use our micro learning-based approach and use one individual profiler for every smart meter and define an additional profiler for every concentrator, which learn from the individual profilers of the connected smart meters. As learning algorithm we use in both cases Gaussian mixture models, with 12 components, profiling the consumption over a 24 hours period, resulting in 2-hours resolution (24/12=2). We train the profilers for both cases during the training period, then we use them in the testing period to estimate/predict the power consumptions for this period.

We simulate regular reconfigurations of the electric grid, *i.e.,* we change the connections from smart meters to concentrators. This scenario is inspired by the characteristics of a typical real-world smart grid topology, as described in [173]. Every hour we randomly change the connections from smart meters to concentrators. At any given point in time, each concentrator has between 50 and 200 connected meters.

We performed all experiments on an Intel Core i7 2620M CPU with 16 GB of RAM and Java version 1.8.0_73. All experiments are available at GitHub[4].

7.3.2 Accuracy

First, we compare the coarse-grained profiling to the micro learning approach to predict the power consumption over the testing set. Figure 7.3 shows the results of this experiment. In both plots, the blue curve represents the testing dataset, *i.e.,* the real power consumption that has to be predicted.

The coarse-grained profiler is not affected by the topology changes. In fact, the profiler at the concentrator level has learned an average consumption that is always replayed without considering the connected smart meters. This explains the periodic, repetitive

[3]http://data.london.gov.uk/dataset/smartmeter-energy-use-data-in-london-households
[4]https://github.com/kevoree-modeling/experiments

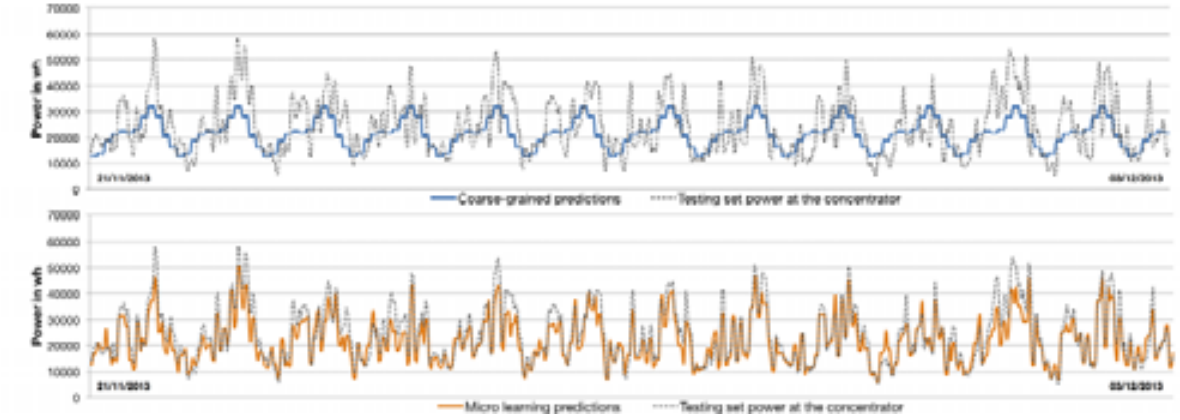


Figure 7.3: Coarse-grained profiling *(top)* vs micro learning profiling *(bottom)*

aspect of the prediction curve.

In contrary, the micro learning approach defines a profiler on the concentrator as a composition of the profilers of all connected smart meters, as shown in the meta model in Listing 11. In case the topology changes, *e.g.,* a smart meter disconnects, the concentrator profiler (composed of several smart meter profilers) will no longer rely on the profiler of the disconnected smart meter. As depicted in Figure 7.3, for the micro machine learning profiling, the plotted curve is significantly closer to the curve of the real testing set than the coarse-grained learning; although both uses the same profiling algorithm: a Gaussian mixture model. For readability reasons we only display the first 12 days of predictions. Prediction curves in case of micro learning are very close (even hard to distinguish) to the real testing set.

We plot the histogram of the prediction errors for both, coarse-grained and micro learning in Figure 7.4. It shows the distribution of the prediction error of both cases. Overall, micro learning leads to an average error of 3,770 wh, while coarse-grained learning leads to an average error of 6,854 Wh. In other words, the error between the prediction and real measurement is divided by two. Knowing that the average power consumption over the whole testing set is 24,702 Wh, we deduce that the micro learning profiling has an accuracy of 85%, while coarse-grained learning has an accuracy of 72%. The accuracy is calculated by $(1 - avgError/avgPower)$. We can conclude that micro learning can be significantly more accurate than coarse-grained learning.

A noticeable result is that the same algorithm can lead to a better accuracy when used at a smaller level and combined with the domain knowledge. Therefore, we argue that this decision is very important and motivate by itself the reason why we focus this contribution on offering modelling abstractions for this purpose.

7.3.3 Performance

In terms of performance, Table 7.1 shows the time needed in seconds to load the data, versus the time needed to perform the live profiling for different numbers of users and

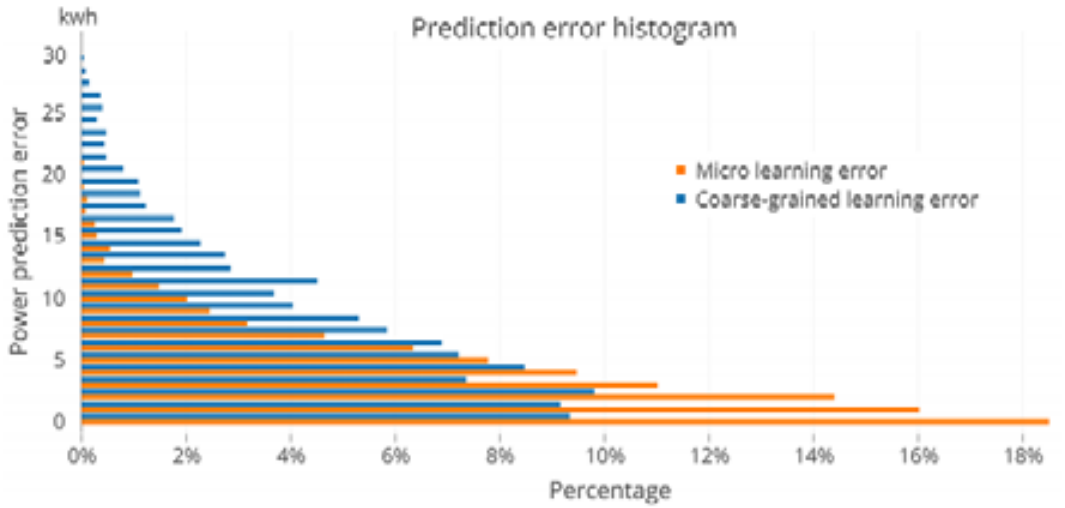


Figure 7.4: Power prediction error histograms

Table 7.1: Loading time and profiling time in seconds. Scalability test over 5000 users and 150 millions power records

Number of users	**Number of records**	**Loading data time in s.**	**Profiling time in s.**
10	283,115	4.28	1.36
50	1,763,332	21.94	7.20
100	3,652,549	44.80	14.44
500	17,637,808	213.80	67.12
1000	33,367,665	414.82	128.53
5000	149,505,358	1927.21	564.61

power records. For instance, for 5000 users and their 150 million power records, it takes 1927 seconds to load and parse the whole dataset from disk (around 32 minutes, given that the dataset is around 11 gb large). However, only 564 seconds are spent for profiling (less than 10 minutes).

Another observation that can be deduced from Table 7.1 is that both loading and training time are linear with the number of records loaded ($O(n)$ complexity). A considerable performance increase can be achieved by distributing and parallelising the computation, especially using micro learning, where every profile can be computed independently. We decided to present results without the usage of a distributed storage backend (*e.g.,* HBase [10]). This would pollute computation times due to networking and caching effects. However, our results allow to meet the performance requirements of case studies like the smart grid. Indeed, during these experiments our modelling framework ingest more than 60,000 values per seconds on a single computer. This is

comparable to data processing frameworks like Hadoop [105].

7.4 Discussion: meta learning and meta modelling

Weaving machine learning into domain modelling opens up interesting possibilities in the intersection of meta learning and meta modelling. Meta learning is about learning the parameters of the learning class itself and adapting these parameters to the specific business domain, where the learning is applied to. The following points are considered as typical meta learning problems:

- Changing the inference algorithm.
- Adding or removing more input attributes.
- Modifying the math expression of an attribute.
- Changing learning parameters (for ex. learning rate).
- Chaining or composing several learning units.

Such changes can be introduced during the execution of the system, reflecting new domain knowledge, which needs to be injected. Therefore, considering that we model learning parameters, this makes it necessary to enable meta class changes at runtime. However, changing learning algorithms or parameters can occur more often than meta model changes. This opens up the reflection on new research directions about frequent meta model updates.

We developed our modelling framework for micro learning. Nonetheless, as discussed, we support fine-grained but also coarse-grained learning. However, our framework—and approach—is clearly designed for micro learning and is therefore mainly useful for systems which are composed of several elements which behave differently. Examples for such systems are CPSs, IoT, and recommender systems. For systems dealing mainly with large datasets of "flat data", *i.e.,* unstructured data without complex relationships between, our model-based micro learning approach is less beneficial. Our approach is mostly beneficial for systems dealing with complex structured and highly interconnected domain data, which have to continuously refine behavioural models that are known at design time with what can be learned only from live data to solve known unknowns.

7.5 Conclusion

Coarse-grained learned behavioural models do not meet the emerging need for combining and composing learnt behaviours at a fine-grained level, as for instance required for CPSs and IoT systems, which are composed of several elements which are diverse in

their behaviours. In this chapter we proposed an approach to seamlessly integrate micro machine learning units into domain modelling, expressed in a single type of model, based on one modelling language. This allows to automate the mapping between the mathematical representation expected by a specific machine learning algorithm and the domain representation. We showed that by decomposing and structuring complex learning tasks with reusable, chainable, and independently computable micro learning units the accuracy compared to coarse-grained learning can be significantly improved. We demonstrated that the ability to independently compute and update micro learning units makes this approach fast enough to be used for live learning and, therefore, for live analytics. This makes it possible to weave machine learning directly into the distributed, multi-dimensional graph data model introduced in this dissertation.

Part IV

Industrial application and conclusion

8

Industrial application: electric overload prediction and warning

This chapter presents an industrial application using the techniques presented in the previous parts of this dissertation. More specifically, it describes how model-driven live analytics is applied in order to realise an intelligent near real-time electric overload prediction and warning system.

This chapter is based on the work that has been presented in the two following papers:

- Thomas Hartmann, Assaad Moawad, François Fouquet, Yves Reckinger, Jacques Klein, and Yves Le Traon. Near real-time electric load approximation in low voltage cables of smart grids with models@run.time. In *Proceedings of the 31th Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016*
- Thomas Hartmann, Assaad Moawad, François Fouquet, Yves Reckinger, Tejeddine Mouelhi, Jacques Klein, and Yves Le Traon. Suspicious electric consumption detection based on multi-profiling using live machine learning. In *2015 IEEE International Conference on Smart Grid Communications, SmartGridComm 2015, Miami, USA, November 2-5, 2015*

Contents

8.1 Context

In this section, we first describe the partnership between Creos and the SnT, University of Luxembourg. Then, we detail the REASON project in which context this dissertation is conducted in.

8.1.1 The Creos partnership

Creos, a member of the Enovos Group, owns and manages the electricity and natural gas networks in Luxembourg. Currently, more than 650 people are working at Creos. The company is responsible for the planning, realisation, extension, maintenance, management and breakdown service of the high, medium, and low tension electricity networks and the high, medium, and low pressure natural gas pipelines it manages. The Creos networks include some 9,000 kilometres of electricity lines and approximately 1,900 kilometres of natural gas pipelines. Almost 240,000 customers are connected to the electricity network and approximately 45,000 customers to the natural gas network. Creos also provides management and meter reading, processing of customer consumption data, and billing for network access.

Creos is a member of the Smart Grid Luxembourg 2.0 consortium, which goal is to explore and develop future energy services in smart grids. Therefore, several technologies of data transmission (*e.g.,* PLC, fiber, GPRS ...) are tested in various pilot projects by Creos. The primary objective of the Smart Grid Luxembourg 2.0 consortium is to set up a technical infrastructure that collects data from smart meters (electricity, gas). A second objective is to analyse the interaction with users, and a third objective is to develop an architecture that is compliant with future challenges of the European energy and water market (clear, integration of renewable energy, electric vehicles, etc.). This is to initiate a "breeding ground" for innovative services including opportunities with the potential to create new service companies in Luxembourg.

Alongside technological tests, Creos launched a research project together with the SnT, University of Luxembourg, to explore and provide ideas how the new information and communication technologies can provide added value for smart grids. This research project is where the work behind this PhD thesis is conducted in.

8.1.2 The REASON project

In the context of this partnership, we developed a live monitoring and analytics system, called REASON. REASON is based on the model-driven live analytics approach presented in this dissertation. The objective of REASON is to analyse the collected data in near real-time and to support decision-making processes based on this analysis. This can help to make the electricity grid able to dynamically react and adapt itself to evolving contexts. Besides giving recommendations, REASON also provides an interactive user interface (UI), which visualises the collected, computed, and learned

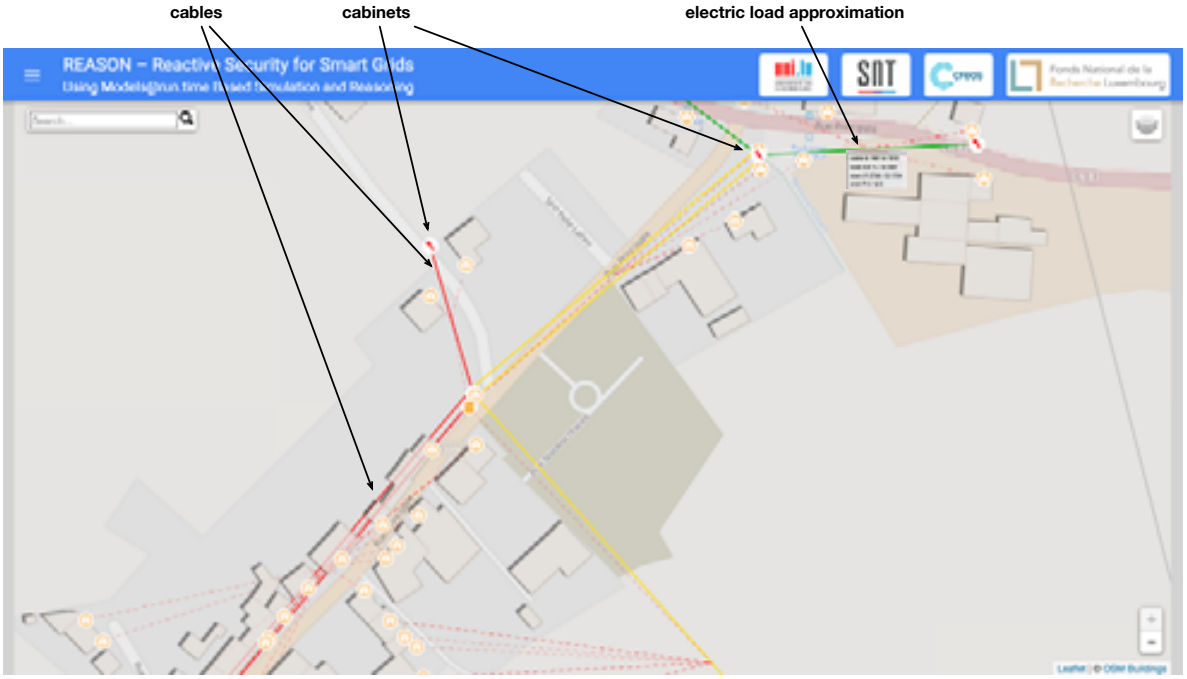


Figure 8.1: REASON: a near real-time monitoring and analytics tool for Creos

data. Figure 8.1 shows a screenshot of this UI. As can be seen in the figure, REASON displays all smart metes, concentrators, the physical, as well as the communication topology, in form of an interactive geographical map. Since this system is based on the multi-dimensional graph data model (and its temporal aspects), described in Chapter 4 and Chapter 5 of this thesis, REASON allows to show (and analyse) the state of the grid at any point in the past. The ability of this data model to represent and explore many different actions, allows REASON to simulate hypothetical actions, *e.g.,* to connect or disconnect certain cables. Due to the fact, that our proposed data model seamlessly combines machine learning and domain data, for example, the consumption profile and therefore the expected future consumption of customers can be displayed. The possibility to distribute our data model over several nodes allows to deploy an instance of REASON on every data concentrator.

One major feature implemented in REASON is a near real-time electric overload prediction and warning system. The goal is to anticipate the load in cables and to simulate corrective actions—*e.g.,* topology changes—in order to find suitable counter-reactions, before the overload actually occurs. In fact, REASON is able to approximate the load in electrical cables, based on simplified electrical models and learned consumption profiles of customers, and to predict potential overload situations. In addition, it can simulate different actions and their effects on the load in order to find appropriate counter-reactions. This feature is detailed in the following.

The rest of this chapter is structured as follows. First, in Section 8.2 we present the meta model used in REASON. Then, Section 8.3 introduces the problem of electric overload risks in electricity cables. Section 8.4 presents how we approximate the electric load based on simplified electrical formulas and models. In Section 8.5, we detail how we predict the electric consumption of customers in order to predict the electrical load

in cables. We evaluate our near real-time electric overload prediction and warning system in Section 8.6. Finally, this chapter concludes in Section 8.7.

8.2 Smart grid meta model

In a first step, we defined together with the domain experts from Creos the meta model used in REASON. This meta model defines the smart grid topology structure, the essential attributes and relations, and domain knowledge, necessary for the context of this case study. Listing 12 shows a simplified version of it, defined in our textual meta modelling language (*cf.* Chapter 7).

The presented meta model is then used to generate a corresponding model, the multi-dimensional graph data model (*cf.* Chapter 5), which is used during runtime. This model is continuously updated with sensed data and is the foundation of our near real-time electric overload prediction and warning system.

8.3 Electric overload prediction and warning

The ever-increasing demand for energy and the increasingly complex grid structure, *e.g.,* due to an integration of renewable energies and distributed micro generations, entails a high overload risk in electricity networks. This motivates the interest of Creos—and other electricity grid operators—to predict the electric load in the network to anticipate overload risks. Therefore, we are working together with Creos on a feature, built into REASON, to continuously analyses the state of the grid based on a runtime model, using machine learning techniques, live measurements, and domain knowledge to derive and predict the electric load in cables in near real-time. This, combined with the possibility to simulate different actions and their impacts, allows to take appropriate counter reactions before the overload actually occurs. For example, by opening/closing fuses, the physical network can be reconfigured to balance the load or, in future scenarios, electric vehicles could be forced to delay their charge cycles or to transfer electricity back to the grid in peak times [165]. Another strategy is to remotely limit the maximal allowed power consumption for certain customers.

Computing the electric load in cables is challenging and requires complex and computational intensive power flow calculations and up-to-date measurements of electric consumption. These are usually based on a static and therefore often outdated view of the physical grid topology. Thus, the electric load in cables is usually only calculated in case of major topology changes. For this reason, such tools are ill-suited for near real-time calculations, as needed for our electric overload prediction and warning system. In Section 8.4 of this chapter, we present how our model-driven live analytics approach can be used to approximate the load in cables in near real-time.

Anticipating the electric load in cables depends on an accurate prediction of customers

Listing 12 Smart grid meta model used in REASON

```
abstract class Entity {
  att serialNumber: String
  att communicationActive: Boolean
  att activeEnergyConsumed: Double
  att reactiveEnergConsumed: Double
  att activeEnergyProduced: Double
  att reactiveEnergProduced: Double

  rel registeredBy: Entity
  rel registeredEntities: Entity
  rel location: Location
  rel cables: Cable
}

class SmartMeter : Entity {
  att maxAllowedPower: Double
  att durationToRead: Double
  att electricityActive: Boolean
  att highPowerCurrentActive: Boolean
  att distanceToConcentrator: Double

  rel customer: Customer

  derived att isRepeater: Boolean {
    from "registeredEntities"
    using "deriveIsRepeater"
  }
}

class Customer {
  att name: String
}

class Concentrator: Entity {
}

class Location {
  att address: String
  att latitude: String
  att longitude: String
}

class Cable {
  att payload: String
  att material: String
  att size: Double
  att remark: String
  att startPoint: Location
  att endPoint: Location

  rel entities: Entity
}
```

consumption behaviour. Instead of approximating the load in cables based on current consumption values, the use of predicted values allows to forecast the electric load in cables. Section 8.5 details how the techniques presented in this dissertation allow to seamlessly integrate learned consumption profiles into the multi-dimensional domain data model.

We apply the proposed multi-dimensional graph data model (*cf.* Chapter 5) to combine learning customers' consumption behaviour and approximating the electric load in cables into a single model. This model is continuously updated with sensed data. On top of this model we then built our near real-time electric overload prediction and warning system. By taking the size of cables into account, we can approximate the maximum capacity of each cable and create alarms if the load is likely to reach (based on the prediction) a threshold value, *e.g.,* 75% of its capacity. REASON can be used to simulate different actions, *e.g.,* changing the topology, to anticipate the impacts of these actions, and to eventually find a solution to avoid the potential overload. This has been developed together with Creos for the near real-time electric overload prediction and warning system and also for technicians to decide wether—and when—it is safe to disconnect a cable for maintenance reasons.

8.4 Electric load approximation

In this section, we describe how we approximate the electric load in cables and how we simulate the impacts on the load in case of changes (*e.g.,* cable disconnections, topology reconfigurations). The objective is to build an electric load monitoring system as well as a decision support system for technicians. By combining the electric load approximation with predicted consumption values (detailed in the next section), the electric load in cables cannot just be approximated for the current time but can also be predicted.

8.4.1 General considerations

To solve the problem of a dynamic anticipation of the electrical load in the grid under certain planned/unplanned events, we combine our smart grid model (*cf.* Section 1.4) and active data (continuously updated at runtime). Based on this model, we analyse the state of the grid and apply electrical formulas, based on a simplified electrical model, in relation with reactive/active aspects that are beyond the scope of common simulation tools. The simplified electrical formulas are domain knowledge and integrated as such into our multi-dimensional graph data model (*cf.* Chapter 5). This enables to consider dynamic changes, (*e.g.,* in the physical grid topology but also in the measured values, like consumption data). Figure 8.2 shows an overview of our approach.

In a first step, we derive the current topology from our model-based abstraction of the grid. The model is continuously updated from the live measurements of the smart grid.

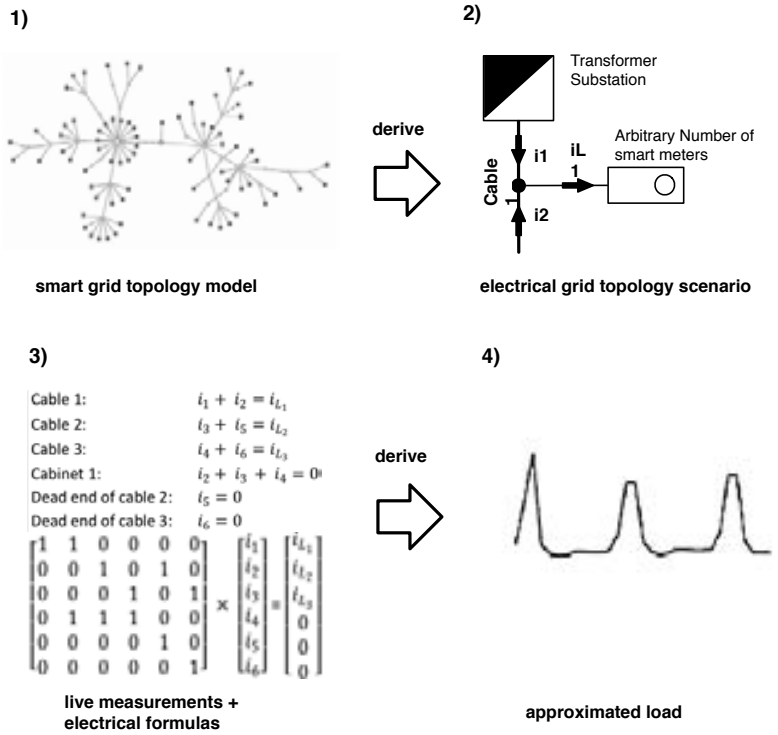


Figure 8.2: From the smart grid model (1) we first infer the electrical topology scenario (2), then combine it with live measurements (or predictions) and apply the appropriate electrical formulas (3), to finally derive the load approximation (4)

Then, based on the derived physical topology, we apply simplified electrical formulas for the load approximation. In a final step, we solve the formulas and calculate the electric load. In the following we describe these steps in more detail.

The fundamental physical law on which the following calculations are based on is Kirchhoff's current law [256]. It says that *"the sum of all currents flowing into a node is equal to the sum of the currents flowing out of this node"*, or more formal:

$$\sum_{k=1}^{n} I_k = 0, \textit{ where } n \textit{ is the total number of currents flowing towards or away from the node.}$$

If we apply this on our topology model, we can derive four basic rules for the electric load approximation:

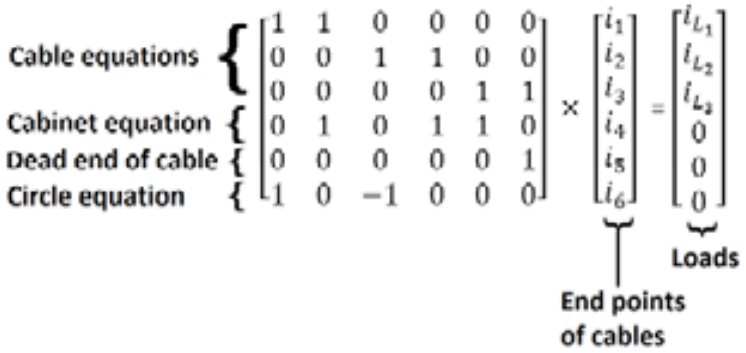
1) For every cable we need one current calculation for the ends of the cable, i_1 and i_2. Since we only have the consumption values of all smart meters and our topology model, which specifies—among other things—which smart meter is connected to which physical cable, all loads of a cable can be summed up as: $I_L = \sum_j i_{load_j}$ (neglecting the active and reactive impact form the cable, *e.g.,* losses, generation). These loads can be considered as a current flow out of the cable (to the consumer) and according to Kirchhoff's current law, we can derive following equation: $i_1 + i_2 = I_L$. i_1 and i_2 are the dominating values for the electric load considerations since they determine the cable loads.

2) We can apply Kirchhoff's current law for all cabinets, meaning that all currents of cables j connected to a cabinet will sum up: $\sum_j i_{cabinet_j} = 0$.

3) For a dead end cable the current at one end is 0.

4) For each circle (cables are directly or indirectly connected in a circular way) the point that is from a physical point of view the nearest to the transformer substation has to be determined. On this point the two cables that are part of the circle must carry the same current: $i_1 = i_2$.

Those rules allow us to calculate the currents at both ends of every cable, independently of the grid structure. In any topology with n cables we implicitly have $2 * n$ unknowns (current at the start and end of each cable) and we, therefore, need $2 * n$ equations to solve the system. Since we have as many equations as unknowns the system to solve will be a square matrix and have always one solution. For example, if we consider the three cables in Figure 8.5a, the equation system to solve would look like the following example:



Each row corresponds to one equation. The columns of the matrix represent for each cable the loads i_1 and i_2 for the ends of the cable. This means that the first two columns belong to cable 1, the next two to cable 2 and so forth.

In order to approximate the electric load of all cables we have to traverse the topology graph, detect the different scenarios regarding the above described four rules, and build and solve the equation system. In the following subsections we describe the different scenarios in more detail and show how we derive the necessary equations. We assume that the smart grid topology in Luxembourg consists of multiple subgraphs and transformer substations are not interconnected. In special cases, where this is not true, our load approximation will yield slightly inaccurate results. We can reduce the complexity of the equation system by deriving one equation system per transformer substation. This can be parallelised so that all equation systems can be independently calculated at the same time. By changing the state of fuses and/or cables in our topology model we can simulate how the electric load in all cables will be effected.

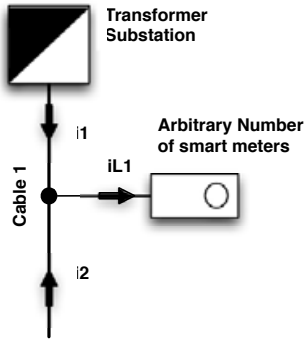


Figure 8.3: Single cable on a substation

8.4.2 Topology scenarios

In this section we introduce the different topology scenarios we have to consider in order to approximate the electric load.

8.4.2.1 Single cable

The first topology scenario we look at is a single cable on a cabinet or transformer substation. Figure 8.3 shows the corresponding topology excerpt. The arrows on cable 1 indicate the conceptual flow of the loads i_1 and i_2. The Figure shows an arbitrary number of smart meters connected to cable 1. The sum of all loads of the smart meters are indicated by load I_{L_1}. We are only interested in the load of the low-voltage cable (cable 1), not in cables connecting meters to the low-voltage cables. We can derive following equations:

Cable 1: $i_1 + i_2 = i_{L_1}$

Dead end of cable 1: $i_2 = 0$

System to be solved:

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} = \begin{bmatrix} i_{L_1} \\ 0 \end{bmatrix}$$

8.4.2.2 Cabinet connecting several cables

The next scenario is a cabinet connecting several cables. Figure 8.4 illustrates an excerpt of a corresponding topology, to clarify this scenario. We assume that all fuses in cabinet 1 are closed, so that cables 1, 2, and 3 are connected. For each cable we have again two loads for both cable ends. On each cable an arbitrary number of smart meters is connected, which individual loads are summed up in one load value for each cable. Cable 2 and 3 have dead ends (no other cable is connected to this cable end). Therefore, we can derive the equations below:

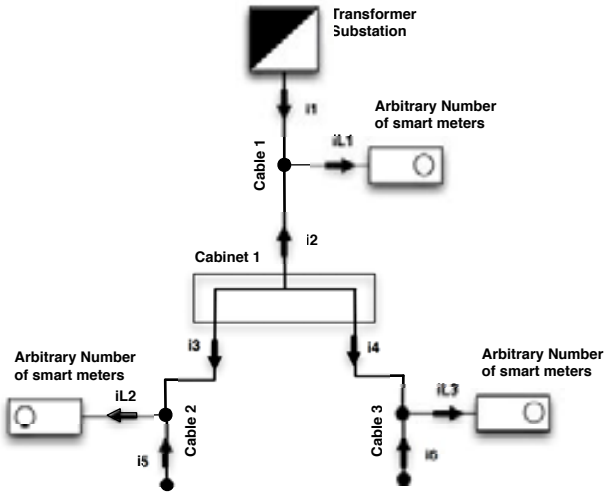


Figure 8.4: A cabinet connecting several cables

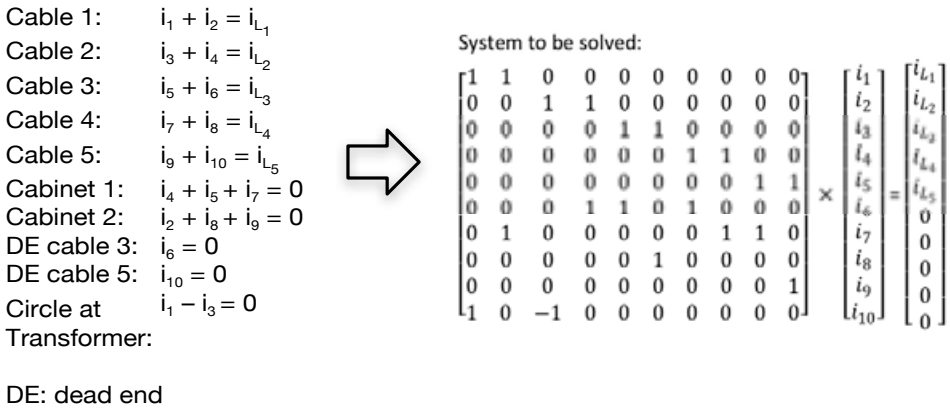
Cable 1: $i_1 + i_2 = i_{L_1}$
Cable 2: $i_3 + i_5 = i_{L_2}$
Cable 3: $i_4 + i_6 = i_{L_3}$
Cabinet 1: $i_2 + i_3 + i_4 = 0$
DE cable 2: $i_5 = 0$
DE cable 3: $i_6 = 0$

DE: dead end

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} i_1 \\ i_2 \\ i_3 \\ i_4 \\ i_5 \\ i_6 \end{bmatrix} = \begin{bmatrix} i_{L_1} \\ i_{L_2} \\ i_{L_3} \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

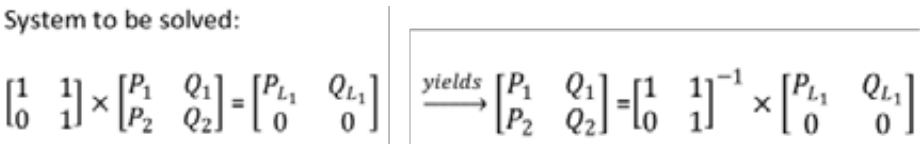
8.4.2.3 Parallel cables

The most complicated scenario are parallel cables, which can appear in different types. First, several cables can start at the same transformer and end at the same cabinet. Second, parallel cables can appear between two cabinets. This means that several cables start at the same cabinet and all of them end at the same cabinet. Last but not least, we have to consider "indirect parallel cables". These start at the same substation but not necessarily end immediately at the same cabinet. If a cable ends at a cabinet and is there connected to another cable, which ends at the cabinet where other cables starting at the substation ends, they indirectly form a circle. These three scenarios are sown in Figure 8.5. Figure 8.5a shows parallel cables at a transformer, 8.5b parallel cables at a cabinet, and 8.5c indirect parallel cables. For 8.5c we can derive following equations:



8.4.3 Considering active and reactive energy

The calculations so far are only valid for purely resistive loads (only active power). However, in a real grid the current always has a reactive component. To take this fact into account, we apply complex numbers: $i_1 = i_{1_{active}} + j * i_{1_{reactive}}$, where j is the imaginary complex number, with ($j^2 = -1$). To simplify the approximations we assume that the voltage at each point is equal to 230 V. Since we have the active and reactive energy from the smart meter measurements (customers' consumption data), we can simplify the calculation by taking the active power P and reactive power Q into account, instead of calculating first the current and divide it into an active and reactive part: $S_1 = P_{1_{active}} + j * Q_{1_{reactive}}$ where j is the complex number. The principal to establish the equations stays the same. On the first topology example (see Figure 8.3) this looks like below:



8.4.4 Deriving the electric load

To approximate the load in cables, we first create all necessary equations. After this step, we solve the matrix equations and calculate for every cable the components P_1, Q_1, P_2, and Q_2 (the two ends of a cable). With this, it is possible to calculate the electric load i_1 and i_2 for every cable:

$$i_1 = \frac{\sqrt{P_1^2+Q_1^2}}{\sqrt{3}*230},\ i_2 = \frac{\sqrt{P_2^2+Q_2^2}}{\sqrt{3}*230}$$

In order to simulate the impacts on the electric load if, for example, a cable would be disconnected, we can simply update the topology model with the disconnected cable

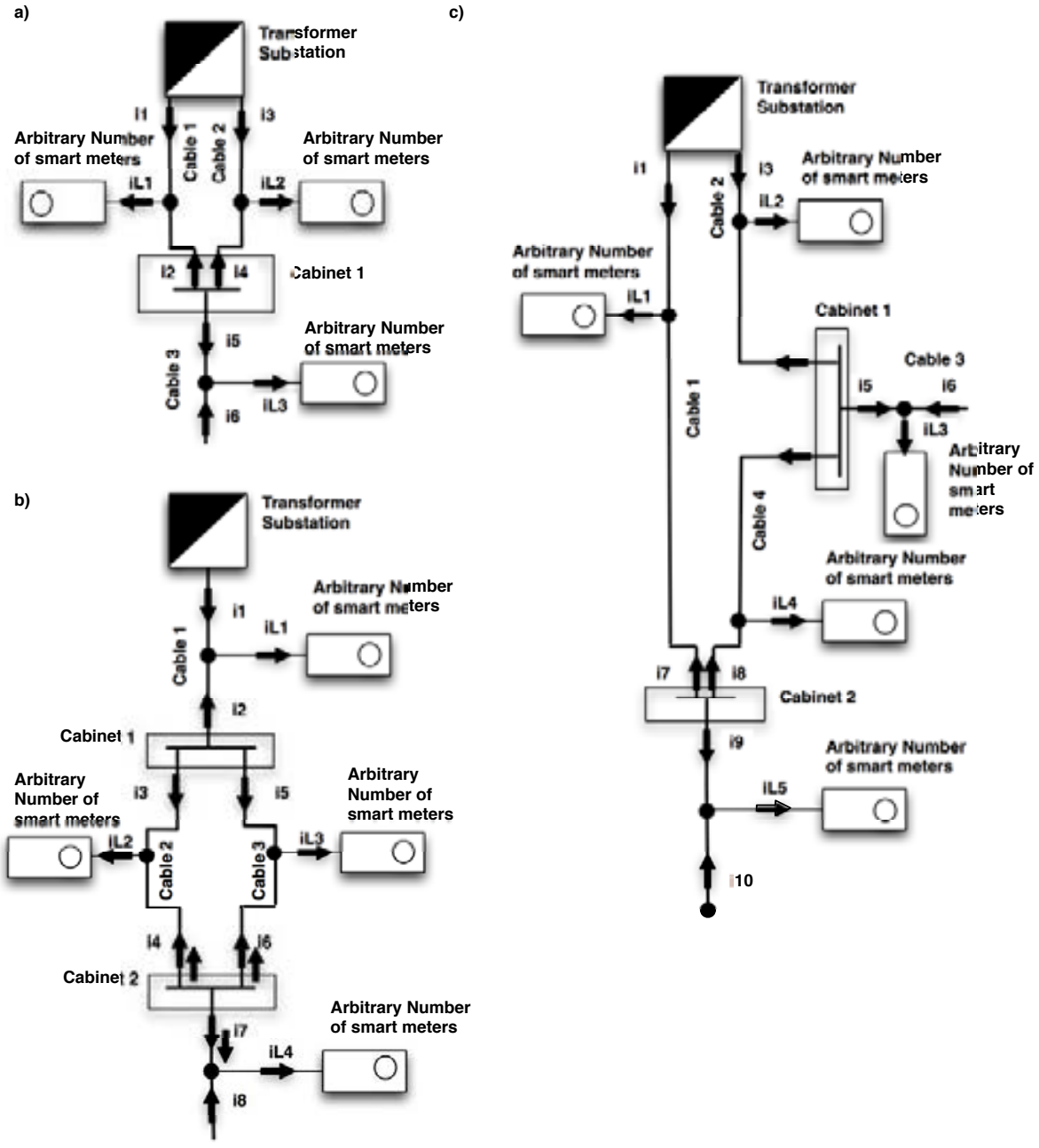


Figure 8.5: Parallel cables: a) at a transformer substation, b) at cabinets, c) indirect parallel cables

and trigger the load approximation. Therefore, the concerned equations are recreated and the load in the cables is updated accordingly.

8.4.5 Integration into the smart grid meta model

Finally, we can extend the smart grid meta model definition with two derived properties: *electricLoad* and *overloadRisk*. This is shown in Listing 13.

Listing 13 Smart grid meta model used in REASON

```
class Cable {
  [...]
  derived att electricLoad: Double {
    using "approximateElectricLoad"
  }

  derived att overloadRisk: Boolean {
        using "deriveOverloadRisk"
  }
}
```

As can be seen in the listing, the electric load is computed in a function *approximateElectricLoad*. This function is implemented with the described strategy to approximate the load. The derived *overloadRisk* attribute is computed in a function *deriveOverloadRisk*. Depending on the material, size, and load of the cable, this function yields if there is an overload risk.

8.5 Predicting consumption behaviour

The previous section described how the electric load in cables can be approximated, using our model-driven live analytics approach. In order to provide timely warnings it is necessary to predict the load in cables in advance. This requires to accurately predict customers' consumption behaviour. In this section, we describe how we apply our approach to learn customers' consumption behaviour by combining learning and modelling.

8.5.1 General considerations

As detailed in Chapter 7 of this dissertation, the presented multi-dimensional domain data model enables to seamlessly integrate machine learning. We leverage this feature to learn the consumption behaviour of customers. Therefore, we add a profiler to every smart meter. Whenever the model is updated with a new consumption value, the profile of the corresponding smart meter is automatically updated. Updates of

profiles are computed in live. The following sections detail live learning and Gaussian mixture models, which are internally used to build the profiles.

8.5.2 Live machine learning

From observing large sequences of data, machine learning and pattern recognition algorithms can build models that reflect or represent, to a certain degree of accuracy, the domain or the environment on which they are trying to learn from. In real-world environments, large sequences of data may not be available in advance, may take too much time to gather, or they can be very expensive in terms of computation power to process in a batch mode. For a reactive system, operating in (near) real-time, fast response times are a crucial requirement. In order to address this, we use live machine learning algorithms [211] with the following characteristics (*cf.* Chapter 7):

- The algorithms should be able to create or update the models whenever new data arrives (on the fly).
- The computational effort needed for a single update should be minimal and should not depend on the amount of data observed so far.
- The update should only depend on the latest observed value and should not explicitly require access to old data.
- The generated models should be compact and should not grow significantly with the number of observed instances.

8.5.3 Gaussian mixture models

In this section, we explore modelling power consumption usage by probability density functions (PDFs) based on kernel density estimates (KDEs). Particularly, we use Gaussian mixture models (GMMs), which are known to be a powerful tool in approximating distributions, even when their form is not close to Gaussian [316]. A GMM is a probabilistic model that assumes that all data points are derived from a mixture of Gaussian distributions with unknown parameters. Mixture models are basically generalising k-means clustering.

Definition 25 *In a nutshell, A Gaussian mixture model of M components, provides the following probability distribution function of an event x happening:*

$p(x) = \sum_{j=1}^{M} w_j K_{\sigma_j}(x)$, *with*
$K_{\sigma_j}(x) = (2\pi\sigma^2)^{-1/2} \exp^{-(x-x_j)^2/(2\sigma^2)}$.

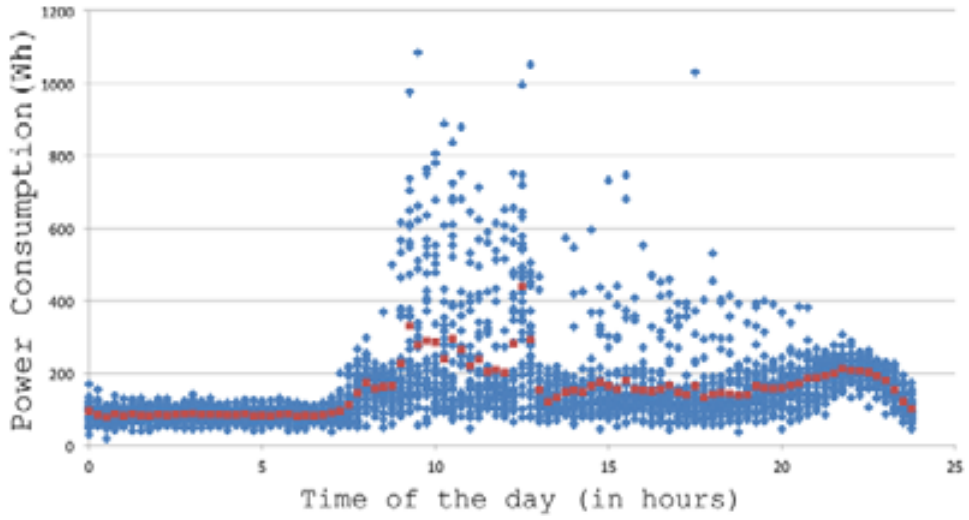


Figure 8.6: Power consumption measures (in blue) and average values (in red)

$K_{\sigma_j}(x)$ is one Gaussian component of the mixture model, with an average of x_j, a standard deviation of σ_j and a weight w_j. These parameters must be learned from the data on the fly.

We implemented an online learning algorithm, based on [210], that is able to update the Gaussian mixture model in near real-time. This algorithm is integrated into KMF and used in our software monitoring and alerting system.

8.5.4 Profiling power consumption

In order to build consumption profiles in near real-time, we feed the measured consumption values from smart meters to our profiler and process them online. In Luxembourg, each smart meter reports its consumption values every 15 minutes for electricity (and 60 minutes for gas). Figure 8.6 shows an example of measurements from one customer (one smart meter) over a period of 24 hours.

The measurements were taken on 31 weekdays (Monday to Friday). The x-axis in the figure represents the time of the day and the y-axis the consumption (active energy consumed) in *Wh*. Every blue point in the figure corresponds to one measurement value. For example, if we take the time 6.00 am, every point along the y-axis belongs to one measurement for one day at 6.00 am. In red, the figure shows the average for every time, *i.e.,* the average value over all days at every measured time (15 minute intervals). Based on these measurements, we can create our consumption profiles for this customer by feeding these measures to our profiler (in real-time) and processing them online. For every new value the consumption profile of the customer will be refined (recalculated). Figure 8.7 shows the profile (the Gaussian mixture model), constructed for the example of Figure 8.6.

As an example, the power usage of this user is quite predictable at midnight (varying

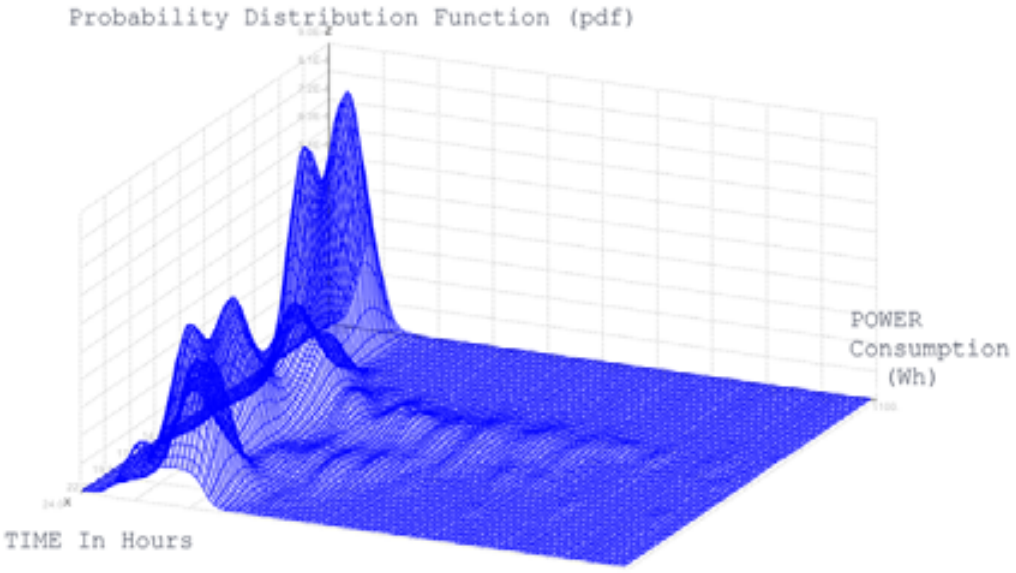


Figure 8.7: Probability distribution function (pdf) of the consumption values from Figure 8.6 built with live machine learning

between 0 and 200 Wh). This is reflected in the profile by a Gaussian Kernel with low variance and we are quite confident (with high probability) that the next midnight measure will be also between 0-200 Wh. However, if we compare this with the consumption at noon, where the user consumes between 0 and 1000 Wh, the profiling has a higher variance, the probability is distributed over a wider range, and thus the prediction is less accurate. In such situations, having a contextual profiling can help to significantly increase the accuracy of the prediction. For instance, during weekends at noon, the consumption may be varying in a less wider range than during weekdays at noon.

8.5.5 Integration into the smart grid meta model

As described in Chapter 7, we integrate the learning of consumption profiles directly into the model. This is shown in Listing 14.

This model can now be used in analytic processes to seamlessly predict the electric load in electricity cables.

8.6 Evaluation

In this section we evaluate our electric overload prediction and warning system. Therefore, we first detail the experimental setup. Then, we evaluate the performance and accuracy of the electric load approximation, before we investigate the efficiency and accuracy of consumption profiling.

Listing 14 Extended smart grid meta model used in REASON

```
class SmartMeterProfiler {
  [...]
  rel smartMeter: SmartMeter [1..1]
  learned att powerProbabilities: Double[] {
    from "Hour(smartMeter.time)"
    from "smartMeter.activeEnergy^2"
    from "smartMeter.reactiveEnergy^2"
    using "GaussianMixtureModel"}
  }
}

class SmartMeter {
     [...]
     rel profile: SmartMeterProfiler
}
```

8.6.1 Experimental Setup

In order to evaluate our electric overload prediction and warning system, we use the smart gird meta model, presented in Section 8.2. This model is applied on a smart grid testbed deployed in Luxembourg. It contains three transformer substations, 218 smart meters, 30 cables, and 27 cabinets.

The number of 10 cables and 100 smart meters per substation is representative for three phase grids, like the ones in Germany, Switzerland, Austria, or Luxembourg. Furthermore, cables of different substations are usually not interconnected. Therefore, the electric load in cables can be independently approximated for the cables of each transformer and can be parallelised.

To evaluate the accuracy and performance of the consumption value prediction, we investigate one cable and 30 connected smart meters. We use in total a number of 631,355 consumption values (of these 30 meters) to train the learning. Since the electric load in a cable depends on the consumption of all connected meters, we aggregate their consumption values for the evaluation of the accuracy. Then, we use a test dataset of 2,778 values (timepoints) to test the accuracy of the prediction. The used training period is 15/08/2012 to 21/11/2013 and the testing period from 21/11/2013 to 17/01/2014.

We used this model to evaluate our approach in terms of performance and accuracy, to validate its suitability to be used in a near real-time simulation system for electric load prediction and warning, in low-voltage cables. We run the experiments on a 2.6 GHz Intel Core i7 with 8 GB of RAM.

Scenario	Overall	Creating	Solving
Transformer Substation 1 (103 meters, 12 cables)	191 ms	190 ms (99.95%)	$\leq$ 1 ms (0.05%)
Transformer Substation 2 (71 meters, 10 cables)	157 ms	156 ms (99.94%)	$\leq$ 1 ms (0.06%)
Transformer Substation 3 (56 meters, 8 cables)	143 ms	142 ms (99.93%)	$\leq$ 1 ms (0.07%)

Table 8.1: Performance evaluation

8.6.2 Performance of electric load approximation

To evaluate the performance of electric load approximation, we changed the topology several times and recalculated the electric load in all cables. We divided the calculation in two steps: *1)* traversing the smart grid graph, finding the topology scenarios and building the equations, and *2)* solving the matrix equation system. For the latter we use a BLAS library [20]. For each of the three scenarios (every transformer substation), we randomly changed the topology (cable connections) 100 times and measured the average times for the recalculation. We neglected I/O operations as far as possible by caching all data instead of reading it from disk. The results of this evaluation are shown in Table 8.1. As can be seen, the costly part is the creation of the equations. This is not a surprise, since our algorithms have to traverse the model graph, detect the scenarios, resolve the appropriate consumption data (right time and customers), and derive the equations. We then gradually increased the complexity of the grid topology (number of cables) in order to evaluate the scalability of our approach and found that the time to approximate the electric load is about linear. This is shown in Figure 8.8. Since our approach allows to independently build and solve the equation systems for every transformer substation, the overall time is determined by the number of cores (to parallelise) and the most complex subgraph. The recalculation time of less than 2 ms, in average, fulfils the near real-time requirement.

8.6.3 Accuracy of electric load approximation

In order to evaluate the accuracy of our model-based approach, we compared our results with the results of the power flow calculation tool (DIgSILENT [26]), which is currently used by Creos. DIgSILENT is a powerful tool for modelling of generation-, transmission-, distribution- and industrial grids, and the analysis of these grids' interactions. It includes advanced features like an AC Newton-Raphson method and a linear DC method. Further, it supports any combination of meshed 1-, 2-, and 3-phase AC and/or DC systems and takes reactive power limits into consideration. In short, DIgSILENT uses advanced and difficult to calculate algorithms, which are based on a

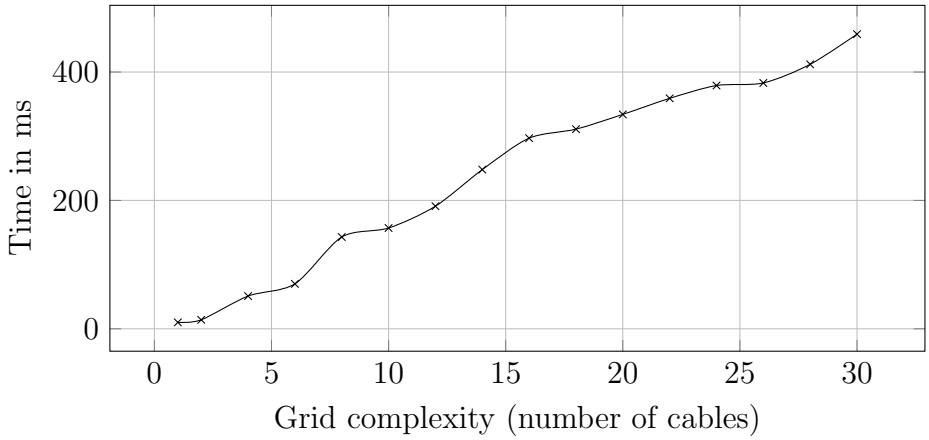


Figure 8.8: Scalability of electric load approximation

deep representation of electrical and physical laws. On the other hand, our proposed model-based approach is based on comparatively simple and easy to calculate approximations of these laws. For the evaluation, the power flow calculation department of Creos took a snapshot of the smart grid topology and consumption data, created manually a static configuration for the power flow calculation tool, and calculated the exact loadings. We analysed the results for several different scenarios and cables and compared it to our approximation approach. For each cable we compared the calculated values for the active as well as reactive energy, at the beginning and ending of the cables, and the cable loading. We found that our approximation approach remaining very accurate with deviations below 5%. The biggest discrepancy we found is 5.77% and the smallest 0.07%. In average, we got an deviation of only 1.89%. This shows that our approach is able to dynamically recalculate the electric load in cables in near real-time while still is very accurate.

8.6.4 Efficiency of electric consumption prediction

The core live learning algorithm GMM to update the profiles of the whole dataset for 30 customers and 631,355 consumption values takes roughly one second. This is around 1,670 nanoseconds per consumption value. Considering that the interval of consumption reading in Luxembourg is 15 minutes, we are able to process around 538,922 consumption readings during one cycle on a single core.

Assessing the fact that the computation is conducted in a single thread on a classical computer processor (single core on an Intel i7 processor), we can consider that our approach is fast enough to be used in a live monitoring system. For example, in the case of Luxembourg, with approximately 550,000[1] inhabitants (and much less households), a standard laptop is sufficient to profile the consumption values of the whole country in live.

[1] http://www.luxembourg.public.lu/fr/societe/population/

Figure 8.9: Accuracy of electric consumption prediction over time

8.6.5 Accuracy of electric consumption prediction

In order to measure the accuracy of our electric consumption prediction, we compare our predicted consumption values to actual ones. Therefore, we consider one cable and 30 connected meters. We compare the sum of the actual consumption values of the connected meters to the sum of the predicted values. As test dataset we use 2,778 timepoints and compare the actual values to the predicted ones. The results are shown in Figure 8.9. The actual consumption over the tested time period (of all smart meters connected to the cable) is 20,211,043 wh and the predicted one is 17,787,342.65. This corresponds to an error rate of 18.81% or an **accuracy of 81,19%**.

Even though the presented approach is relatively accurate, building highly accurate profiles is challenging. This comes especially because electric consumption depends on the context [86], *e.g.,* geographical area, number of residents, temperature, date (vacation, weekday, weekend), type of the heating system, habits of inhabitants, etc. Given the high number of context parameters, it is very difficult to build reliable profiles. For example, a private customer can have a very different consumption profile depending on the weather conditions or during vacation compared to working days. Moreover, context parameters can depend on each other, *e.g.,* the temperature on a winter Sunday afternoon may have different influence on the load profile than the temperature on a summer Monday afternoon. Due to this variability, a single consumption profile per customer, which simply computes an average consumption can be inaccurate. Therefore, we are working together with Creos on a multi-context profiling method [179]. Instead of building one profile per customer, we create multiple profilers, one per context and customer. The context, for example, can include features like: user type (individual, family, industry, commercial), temporal context (season of the year, month, weekday, holidays), and so forth. We are currently investigating how this impacts efficiency and accuracy of the predictions.

8.7 Conclusion

In this chapter, we presented a concrete industrial application of the concepts proposed in this dissertation. More specifically, a near real-time electric overload prediction and warning system has been presented. Such a system is of great interest for Creos, and other grid providers, to better anticipate the load in increasingly complex grid structures (*e.g.,* due to the integration of renewable energies). Anticipating the load in cables is challenging, since it relies on the combination of live measurements (customers' consumption values), domain knowledge (electrical formulas to derive the electric load), learning rules (consumption profiles of customers), and the topology structure (connections of smart meters and cables). The presented system is build on top of the multi-dimensional graph data model and the model-driven live analytics approach introduced in this thesis in order to address these challenges. Therefore, we defined, together with domain experts of Creos, a smart grid meta model to represent the grid topology and to define the domain knowledge (in form of electrical formulas) necessary to approximate the load in cables. The time dimension (*cf.* Chapter 4) of our proposed graph data model is exploited to represent topology changes, data measured at different frequencies, and learned information. Live learning, *i.e.,* building profiles of customers' consumption behaviour, is seamlessly integrated into the data model and is leveraged by analytic processes in the same way than measured or computed (based on domain knowledge) values. Last but not least, the ability of the presented graph data model to represent and explore many different alternatives (*cf.* Chapter 5) allows to simulate different actions and their impacts. This makes it possible—*e.g.,* in case of a predicted overload risk—to automatically explore different actions, derive the hypothetical load in cables and, based on this, to suggest an action to avoid a potential overload before it actually happens. Hence, this makes prescriptive analytics possible and shows that the presented multi-dimensional graph data model can enable model-driven live analytics.

More precisely, we have shown that this approach is able to approximate and predict the load in cables with a high accuracy and is able to simulate the impacts of topology changes in near real-time. The presented idea, which has been developed in cooperation with our industrial partner Creos, has been implemented as a monitoring system to detect potential overloads in cables as well as for technicians to decide wether it is safe to disconnect a cable for maintenance.

9

Conclusion

This chapter concludes the dissertation and presents future research directions.

Contents

This chapter is organised as follows. Section 9.1 summarises the contributions of this dissertation before Section 9.2 discusses potential directions for future work.

9.1 Summary

Recent studies, for example from McKinsey [43], emphasise the tremendous importance of data analytics by calling it the *"next frontier for competition"*. Others even compare the value of data with the value of oil, referring to data as *"the new oil"* [53]. To turn data into valuable insights or actionable intelligence, we need to process and "understand" the large amounts of data collected from various sources. Data analytics has the potential to help us to better understand our businesses, environment, physical phenomena, bodies, health, and nearly every other aspect of our lives. However, turning collected data into competitive advantages remains a big challenge.

Most of todays data analytic techniques are processing data in a pipeline-based way: they first extract the data to be analysed from different sources,*e.g.,* databases, social medias, or stream emitters, copy them into some form of usually immutable data structures, stepwise process it, and then produce an output. By parallelising the processing steps, *e.g.,* based on the map-reduce programming model [128], these techniques are able to mine huge amounts of data in comparatively little time and can find all kind of useful correlations. While this is suitable for tasks like sorting vast amounts of data, analysing huge log files, or mining social medias for trends (even in near real-time) it is less suitable for analytics of domains with complicated relationships between data, where several different analytic techniques and models need to be combined with domain knowledge and machine learning in order to refine raw data into deep understanding [121]. For such analytics, a pipeline-based approach has severe drawbacks and easily leads to an inefficient "blindly store everything and analyse it later" approach, which is referred to as the "big data pathology" or "big data trap" [190].

Cyber-physical systems and IoT are such domains, where traditional approaches fail to envision sustainable techniques for analysing data in live in order to support decision-making processes. For such systems it is crucial to quickly analyse the sensed data and to draw conclusions out of it. In this context, we identified four main challenges, which are addressed in this disseration:

- Analysing data in motion
- Exploring hypothetical actions
- Reasoning over distributed data in motion
- Combining domain knowledge and machine learning

This work presented a novel approach, called **model-driven live analytics** for cyber-physical systems, which aims at addressing these challenges. The proposed techniques have been exemplified and evaluated on a smart gird case study.

In the first part of this dissertation we introduced the context and the challenges we were facing when analysing data of complex CPSs, like smart grids, in live. Thereafter, we presented the background and discussed the state of the art in Part II.

The third part focused on analysing data in motion and what-if analysis. Chapter 4 first presented a continuous semantic for temporal data and a corresponding data model. We showed that this data model is able to represent time as a first-class property, crosscutting any model element and any relationship. Besides enabling a seamless navigation in time and space, we showed that this data model can significantly outperform classical solutions for both, required storage and time to analyse this data. In Chapter 5, we extended this temporal data model, by combining graphs and temporal data, into a multi-dimensional graph model. This allows intelligent systems to explore alternative actions in parallel in order to gradually converge towards their goals. We showed that the proposed graphs are able to handle efficiently hundreds of millions of nodes, timepoints, and hundreds of thousands of independent worlds. The concepts behind this multi-dimensional graph data model are the foundations of our model-driven live analytics approach and the major contribution of this dissertation.

In the fourth part, Chapter 6 discussed how the previously defined data model supports the distributed, large-scale, and continuously changing nature of complex cyber-physical systems. More specifically, we presented an approach based on a combination of ideas from reactive programming, peer-to-peer distribution, and large-scale models@run.time. This approach enables to define distributed models as observable streams of chunks that are exchanged between nodes in a peer-to-peer manner. A lazy loading strategy allows to transparently access the complete virtual data model from every node, although chunks are actually distributed across computational nodes. We demonstrated that this approach can enable frequently changing, reactive distributed models and can scale to millions of elements distributed over thousands of nodes, while the distribution and model access remains fast enough to enable reactive systems. In Chapter 7, we proposed to integrate learning directly into domain modelling. We argued that coarse-grained learned behavioural models do not meet the emerging need for combining and composing learnt behaviours at a fine-grained level, for instance for CPSs and IoT systems, which are composed of several elements which are diverse in their behaviours. Instead, we proposed an approach to seamlessly integrate micro machine learning units into domain modelling, expressed in a single type of model, based on one modelling language. We showed that by decomposing and structuring complex learning tasks with reusable, chainable, and independently computable micro learning units the accuracy compared to coarse-grained learning can be significantly improved. We demonstrated that the ability to independently compute and update micro learning units makes this approach fast enough to be used for live learning.

Finally, Chapter 8 introduced an industrial application of the proposed model-driven live analytics approach. In this chapter we showed on a concrete real-world use case how model-driven live analytics can be applied to build a live analytic system, able to anticipate the load in cables and to simulate corrective actions in order to find suitable counter reactions before an overload actually occurs.

9.2 Future research directions

This section describes potential future research directions.

9.2.1 Searching and selecting appropriate actions

In this dissertation, we proposed methods to explore many alternative actions in parallel in order to enable intelligent systems to converge towards their goals (*cf.* Chapter 5). Every action potentially leads to an alternative state from where a set of other actions can be applied and so forth. However, the resulting set of possible actions can be potentially too large to be fully explored, especially given the near real-time requirements such systems usually face. Therefore, techniques to reduce the search space are needed.

Algorithms, like greedy, genetic, evolutionary, and multi-objective evolutionary algorithm (MOEA) are candidates for reducing large search spaces. An interesting future research direction would be to investigate how such algorithms can be integrated into the proposed model-driven live analytics approach. One of the resulting challenges would be how to evaluate the state of a runtime model, *i.e.,* how to express fitness functions for models. With the development of the Polymer [244] framework we did a first step into this direction. Polymer aims at providing a model-driven approach to define fitness functions and mutation operators without MOEA encoding knowledge.

Other approaches, which would be interesting to investigate, include machine learning algorithms, like reinforcement learning, and a combination of different techniques.

9.2.2 Reinforcement learning for improved action selection

A very interesting future research direction is the usage of reinforcement learning in our approach. Reinforcement learning is used to help software agents to take in particular environments the actions to maximise their rewards. It is inspired by behaviourist psychology. A reward feedback is required for the agent to learn its behaviour. Reinforcement learning is a promising technology to enable intelligent systems to select the most appropriate actions based on previously learned behaviour. Therefore, integrating reinforcement learning into our model-driven live analytics approach is planned as future work.

9.2.3 Encoding continuously evolving data

Even though, the temporal data model presented in this thesis (*cf.* Chapter 4) is able to efficiently represent continuously evolving data and is far more efficient than snapshotting, it still stores every changed model element. Considering, for example sensor data, this can easily lead to a huge amount of data, which is not just costly

to store but also to analyse. Jacobs [190] even calls the modelling of timed data as enumerations of discrete timed values, *"big data pathology"* or *"big data trap"*. A future research direction is to investigate how timed data—especially flat data, like sensor values—can be more efficiently represented. In addition, it would be interesting to see, if the defined temporal semantic, which foresees to resolve for each point in time the last valid version, could be improved for specific types of data. For example, if we consider temperature data, it would be more accurate to retrieve an average between two values instead of the last valid (measured) one. Another idea would be to just store a value if it is "significantly different" from the previous one, *e.g.,* specified with a threshold value in the domain model.

In a first work towards this goal [245] we used polynomials to represent and store sensor data. Instead of storing every value, we only store a polynomial as long as the sensor values can be adequately represented with it, *i.e.,* within an accepted error tolerance. As a side effect this representation also approximates values between two explicit measurements. This approach could be even further generalised by, for example, storing data as polynomial of polynomials, or other mathematical functions.

9.2.4 Meta model evolution

Currently we don't foresee an evolution of meta models over time. In future work we want to investigate how an independent evolution of meta models could be achieved. Meta classes can be represented using the same concepts and techniques than used for regular objects. Every object in a graph can then be simply associated to the object, representing its meta class. Since meta classes at runtime would be normal objects, they also would automatically be able to evolve in time. Just like any other object. As a consequence, for every point in time, every object is associated to one version of a meta class. If the meta class evolves, a new version of the object would point to a new version of the meta class object. This would enable different objects to use different versions of a meta model. State chunks could then be migrated (semi) automatically on the fly.

An interesting work in this direction is presented in [125]. Gwendal *et al.,* discuss a model-driven technique to map UML/OCL conceptual schemas to an abstraction layer on top of different graph databases. This allows to generate database-level queries, *e.g.,* for Tinkerpop Gremlin, via an intermediate graph meta model.

9.2.5 Memory management for analytics

Analysing the state of CPSs requires to process large amounts of data. Our analytics framework is implemented in Java. Processing massive data, in languages like Java, results in creating many temporary objects in main memory. One big advantage of Java is its automatic memory management and garbage collection. However, a big disadvantage coming with garbage collection is that it is unpredictable, comparably slow, and can severely delay the execution of programs. This is especially problematic

considering near real-time requirements.

Every new object is created in the *heap* memory section. If the heap reaches a certain size, garbage collection is triggered and all objects, which are "not reachable", *i.e.,* not referenced by an active object, are marked as garbage. Analytic processes usually load and analyse lots of data. This means that the heap is quickly filled and garbage collection is triggered. The garbage collection process now needs to check the whole heap space for unused objects. Garbage collection can be triggered at any time and finding not reachable objects in large graphs is expensive. This is a common challenge in many big data frameworks, *e.g.,* based on map-reduce approaches, and can significantly slow down analytic processes.

For future work we plan to investigate a more specialised memory management and layout. Instead of relying only on a heap memory section for objects and an automatic garbage collection, we plan to divide the object memory in two zones: 1) a specific zone for objects used for analytics; this zone will be managed manually or by our framework, 2) the usual heap memory zone for all other objects; this zone is managed by Java's garbage collector. We pan to use Java's *unsafe* API for managing this new memory zone off heap. This allows to manually (by our framework) allocate memory for objects used for analytic processes, aside from heap memory. Since these objects reside outside of the heap zone, this would avoid unnecessary garbage collector runs. As soon as an analytic process does not need an object anymore, it manually (by our framework) removes the object from the off heap memory zone. This is inspired by techniques used for high-frequency trading, *e.g.,* ChronicleMap [50].

9.2.6 Data sharding

How to efficiently shard data in the context of model-driven live analytics is a major future research direction. Besides generic approaches to horizontally partition data, *e.g.,* based on unique identifiers of data (the x datasets go to partition y, the next x datasets to partition z, and so forth), or solely relying on an underlying key-value store, investigating how domain knowledge could be leveraged for sharding seems to be promising. For example, in the smart grid use case presented in this dissertation, data of geographically close smart meters and data concentrators could be stored together. Such information could be expressed in the meta model. Another idea would be to apply machine learning algorithms to learn how data should be distributed and reallocate data accordingly.

An interesting approach in the context of user-generated content services is presented by Delbruel *et al.,* [131]. They present a technique to use global predictions in large-scale decentralised systems to forecast where content will likely be consumed and use this information to distribute data accordingly. In [130] and [129] the authors show, for the domain of video serving, that tags on a per-user basis, which are associated to videos, can be used as markers of a video's geographic diffusion, with some tags strongly linked to well identified geographic areas. A more general consideration about programming of large-scale distributed systems is given by Taïani [302]. He discusses three main topics: 1) modular fault-tolerance (in peer-peer overlays), 2) component-

based development and programmability (in gossip-based protocols), and 3) interactive performance analysis (in grid middleware).

9.3 Outlook

The presented model-driven live analytics approach pursues the idea of model-driven engineering further and brings it to the domain of data analytics. In some respects, it can be seen as an advancement of the models@run.time paradigm. Similar to this paradigm, model-driven live analytics suggests to use domain models as abstractions which are simpler than the reality. These models reflect the knowledge of domain experts in form of domain laws and learning rules and drive analytic processes. Model-driven live analytics combines various areas of research, such as software engineering, machine learning, model-driven engineering, models@run.time, databases, and big data analytics. While this thesis focuses on developing model-driven live analytics for cyber-physical systems, we believe that the usefulness of the presented concepts goes beyond this domain and that model-driven live analytics can be applied as a general data analytics method.

List of papers and tools

Papers included in the dissertation:

- 2016
  - Thomas Hartmann, Assaad Moawad, François Fouquet, Yves Reckinger, Jacques Klein, and Yves Le Traon. Near real-time electric load approximation in low voltage cables of smart grids with models@run.time. In *Proceedings of the 31th Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016*
- 2015
  - Thomas Hartmann, Assaad Moawad, François Fouquet, Grégory Nain, Jacques Klein, and Yves Le Traon. Stream my models: Reactive peer-to-peer distributed models@run.time. In *18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MoDELS 2015, Ottawa, ON, Canada, September 30 - October 2, 2015*, pages 80–89, 2015
  - Thomas Hartmann, Assaad Moawad, François Fouquet, Yves Reckinger, Tejeddine Mouelhi, Jacques Klein, and Yves Le Traon. Suspicious electric consumption detection based on multi-profiling using live machine learning. In *2015 IEEE International Conference on Smart Grid Communications, SmartGridComm 2015, Miami, USA, November 2-5, 2015*
- 2014
  - Thomas Hartmann, François Fouquet, Grégory Nain, Brice Morin, Jacques Klein, Olivier Barais, and Yves Le Traon. A native versioning concept to support historized models at runtime. In *Model-Driven Engineering Languages and Systems - 17th International Conference, MODELS 2014, Valencia, Spain, September 28 - October 3, 2014. Proceedings*, pages 252–268, 2014
  - Thomas Hartmann, François Fouquet, Grégory Nain, Brice Morin, Jacques Klein, and Yves Le Traon. Reasoning at runtime using time-distorted contexts: A models@run.time based approach. In *The 26th International Conference on Software Engineering and Knowledge Engineering, SEKE 2014, Vancouver, BC, Canada, July 1-3, 2013.*, pages 586–591, 2014
  - Thomas Hartmann, François Fouquet, Grégory Nain, Brice Morin, Jacques Klein, and Yves Le Traon. Model-based time-distorted contexts for efficient temporal reasoning. In *The 26th International Conference on Software Engineering and Knowledge Engineering, SEKE 2014, Vancouver, BC, Canada, July 1-3, 2014.*, pages 746–747, 2014
  - Thomas Hartmann, François Fouquet, Jacques Klein, Grégory Nain, and Yves Le Traon. Reactive security for smart grids using models@run.time-based simulation and reasoning. In *Smart Grid Security - Second International Workshop, SmartGridSec 2014, Munich, Germany, February 26, 2014, Revised Selected Papers*, pages 139–153, 2014

– Thomas Hartmann, François Fouquet, Jacques Klein, Yves Le Traon, Alexander Pelov, Laurent Toutain, and Tanguy Ropitault. Generating realistic smart grid communication topologies based on real-data. In *2014 IEEE International Conference on Smart Grid Communications, SmartGridComm 2014, Venice, Italy, November 3-6, 2014*, pages 428–433, 2014

Papers not included in the dissertation:

- Assaad Moawad, Thomas Hartmann, François Fouquet, Grégory Nain, Jacques Klein, and Yves Le Traon. Beyond discrete modeling: A continuous and efficient model for iot. In *18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MoDELS 2015, Ottawa, ON, Canada, September 30 - October 2, 2015*, pages 90–99, 2015
- Assaad Moawad, Thomas Hartmann, François Fouquet, Jacques Klein, and Yves Le Traon. Adaptive blurring of sensor data to balance privacy and utility for ubiquitous services. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015*, pages 2271–2278, 2015
- Assaad Moawad, Thomas Hartmann, François Fouquet, Grégory Nain, Jacques Klein, and Johann Bourcier. Polymer - A model-driven approach for simpler, safer, and evolutive multi-objective optimization development. In *MODELSWARD 2015 - Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development, ESEO, Angers, Loire Valley, France, 9-11 February, 2015.*, pages 286–293, 2015

Papers currently under submission:

- under submission at ACM/USENIX EuroSys 2017: Thomas Hartmann, Assaad Moawad, Francois Fouquet, Gregory Nain, Romain Rouvoy, Yves Le Traon, and Jacques Klein. PIXEL: A Graph Storage to Support Large Scale What-If Analysis
- under submission at International Journal on Software and Systems Modeling (SoSyM): Thomas Hartmann, Assaad Moawad, Francois Fouquet, and Yves Le Traon. The Next Evolution of MDE: A Seamless Integration of Machine Learning into Domain Modeling
- under submission at IEEE Computer Magazine: Thomas Hartmann, Assaad Moawad, Francois Fouquet, Gregory Nain, Jacques Klein, Yves Le Traon, and Jean-Marc Jezequel. Model-Driven Analytics: Connecting Data, Domain Knowledge, and Learning

Participated in the development of the following software systems during the dissertation:

- *mwDB:* A many-world graph storage and processing framework, used as core for the latest version of the Kevoree Modeling Framework (KMF) (`https://github.com/kevoree-modeling/mwDB`)
- *KMF*: A modelling, code generation, and analytics framework for cyber-physical systems and IoT applications (`https://github.com/kevoree-modeling/framework`)

- *Kevoree Polymer*: A multi-objective optimisation framework on top of models (`https://github.com/dukeboard/kevoree-genetic`)
- *SGT Gen*: A smart grid topology generator (`https://github.com/thomashartmann/smartgrid-topology-generator`)
- *REASON*: A smart grid dashboard and analytics tool for visualising and analysing smart grid data in near-real time (closed source)

Bibliography

[1] 2013 Hype Cycle Special Report Evaluates the Maturity of More Than 1,900 Technologies. [Online]. Available: http://www.gartner.com/newsroom/id/2575515. Accessed: November 2015.

[2] AllegroGraph. [Online]. Available: http://allegrograph.com/. Accessed: April 2016.

[3] Apache Cassandra. [Online]. Available: http://cassandra.apache.org/. Accessed: March 2016.

[4] Apache Flink. [Online]. Available: https://flink.apache.org/. Accessed: February 2016.

[5] Apache Giraph. [Online]. Available: http://giraph.apache.org/. Accessed: February 2016.

[6] Apache Gora. [Online]. Available: http://gora.apache.org/index.html. Accessed: February 2016.

[7] Apache hadoop. [Online]. Available: https://hadoop.apache.org. Accessed: March 2016.

[8] Apache Hama. [Online]. Available: https://hama.apache.org/. Accessed: February 2016.

[9] Apache Hawq. [Online]. Available: http://hawq.incubator.apache.org/. Accessed: March 2016.

[10] Apache HBase. [Online]. Available: https://hbase.apache.org/. Accessed: March 2016.

[11] Apache Impala. [Online]. Available: http://impala.io/. Accessed: March 2016.

[12] Apache Mesos. [Online]. Available: http://mesos.apache.org/. Accessed: February 2016.

[13] Apache Pig. [Online]. Available: https://pig.apache.org/. Accessed: March 2016.

[14] Apache Spark. [Online]. Available: http://spark.apache.org/. Accessed: March 2016.

[15] Apache TinkerPop. [Online]. Available: http://tinkerpop.apache.org/. Accessed: April 2016.

[16] Apache ZooKeeper. [Online]. Available: https://zookeeper.apache.org/. Accessed: February 2016.

[17] ArrangoDB. [Online]. Available: https://www.arangodb.com/. Accessed: April 2016.

[18] Atlas Graph. [Online]. Available: https://github.com/Netflix/atlas/wiki/Graph. Accessed: July 2016.

[19] Berkeley DB. [Online]. Available: http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html. Accessed: March 2016.

[20] BLAS (Basic Linear Algebra Subprograms). [Online]. Available: http://www.netlib.org/blas/. Accessed: July 2016.

[21] blazegraph. [Online]. Available: https://www.blazegraph.com/product/. Accessed: April 2016.

[22] CDO. [Online]. Available: http://wiki.eclipse.org/CDO. Accessed: April 2016.

[23] Cypher query language. [Online]. Available: https://neo4j.com/docs/developer-manual/current/#cypher-query-lang. Accessed: April 2016.

[24] decking. Create, manage and run clusters of Docker containers. [Online]. Available: http://decking.io/. Accessed: December 2015.

[25] Dgraph. [Online]. Available: https://dgraph.io/. Accessed: April 2016.

[26] DIgSILENT. [Online]. Available: http://www.digsilent.de/. Accessed: December 2015.

[27] docker. Build, Ship, Run. [Online]. Available: http://www.docker.com/. Accessed: December 2015.

[28] Eclipse Modeling Project. [Online]. Available: https://eclipse.org/modeling/. Accessed: July 2016.

[29] eclipse. [Online]. Available: https://eclipse.org/. Accessed: July 2016.

[30] FlockDB. [Online]. Available: https://github.com/twitter/flockdb. Accessed: April 2016.

[31] Galois. [Online]. Available: http://iss.ices.utexas.edu/?p=projects/galois. Accessed: February 2016.

[32] GraphAware Neo4j TimeTree. [Online]. Available: https://github.com/graphaware/neo4j-timetree. Accessed: April 2016.

[33] GraphBase. [Online]. Available: http://graphbase.net/. Accessed: April 2016.

[34] graphdb-benchmarks. [Online]. Available: https://github.com/socialsensor/graphdb-benchmarks. Accessed: April 2016.

[35] GraphDB. [Online]. Available: http://ontotext.com/products/graphdb/. Accessed: April 2016.

[36] InfiniteGraph. [Online]. Available: http://www.objectivity.com/products/infinitegraph. Accessed: April 2016.

[37] InfiniteGraph. [Online]. Available: http://www.objectivity.com/products/thingspan/. Accessed: April 2016.

[38] influxDB Benchmark. [Online]. Available: https://goo.gl/tQtZET. Accessed: December 2015.

[39] influxdb: Time-Series Data Storage. [Online]. Available: https://influxdata.com/time-series-platform/influxdb/. Accessed: April 2016.

[40] InfoGrid. [Online]. Available: http://infogrid.org/trac/. Accessed: April 2016.

[41] Introduction to What-If Analysis. [Online]. Available: https://support.office.com/en-US/article/Introduction-to-what-if-analysis-22BFFA5F-E891-4ACC-BF7A-E4645C446FB4. Accessed: July 2016.

[42] LevelDB. [Online]. Available: http://leveldb.org/. Accessed: April 2016.

[43] McKinsey & Company. Big data: The next frontier for competition. [Online]. http://www.mckinsey.com/features/big_data. Accessed: December 2015.

[44] Merkle DAG. [Online]. Available: https://github.com/jbenet/random-ideas/issues/20. Accessed: August 2016.

[45] mongoDB. [Online]. Available: https://www.mongodb.com/. Accessed: March 2016.

[46] MongoEMF. [Online]. Available: https://github.com/BryanHunt/mongo-emf/wiki. Accessed: August 2016.

[47] neo4j. [Online]. Available: https://neo4j.com/. Accessed: March 2016.

[48] Noms. [Online]. Available: https://github.com/attic-labs/noms. Accessed: August 2016.

[49] Object Constraint Language. [Online]. Available: http://www.omg.org/spec/OCL/. Accessed: December 2015.

[50] OpenHFT. Chronicle-Map. [Online]. Available: https://github.com/OpenHFT/Chronicle-Map. Accessed: December 2015.

[51] OpenTSDB: The Scalable Time Series Database. [Online]. Available: http://opentsdb.net/. Accessed: July 2016.

[52] OrientDB. [Online]. Available: http://orientdb.com/. Accessed: April 2016.

[53] Perry Rotella. Is Data The New Oil?. [Online]. Available: http://www.forbes.com/sites/perryrotella/2012/04/02/is-data-the-new-oil/. Accessed: December 2015.

[54] redis. [Online]. Available: http://redis.io/. Accessed: April 2016.

[55] Representing time dependent graphs in Neo4j. [Online]. Available: https://github.com/SocioPatterns/neo4j-dynagraph/wiki/Representing-time-dependent-graphs-in-Neo4j. Accessed: April 2016.

[56] RIAK KV. [Online]. Available: http://basho.com/products/riak-kv/. Accessed: March 2016.

[57] RocksDB. [Online]. Available: http://rocksdb.org/. Accessed: April 2016.

[58] RRDtool: logging & graping. [Online]. Available: http://oss.oetiker.ch/rrdtool/. Accessed: July 2016.

[59] samza. [Online]. Available: http://samza.apache.org/. Accessed: February 2016.

[60] SmartMeter Energy Consumption Data in London Households. [Online]. Available: http://data.london.gov.uk/dataset/smartmeter-energy-use-data-in-london-households. Accessed: December 2015.

[61] Sparksee. [Online]. Available: http://www.sparsity-technologies.com/. Accessed: August 2016.

[62] Stanford Large Network Dataset Collection. [Online]. Available: http://snap.stanford.edu/data/. Accessed: April 2016.

[63] Stardock. [Online]. Available: http://stardog.com/. Accessed: April 2016.

[64] The Apache Velocity Project. [Online]. Available: http://velocity.apache.org/. Accessed: December 2015.

[65] Titan. Distributed Graph Database. [Online]. Available: http://titan.thinkaurelius.com/. Accessed: April 2016.

[66] Trident API Overview. [Online]. Available: http://storm.apache.org/releases/current/Trident-API-Overview.html. Accessed: March 2016.

[67] turi - create intelligence. [Online]. Available: https://turi.com/. Accessed: August 2016.

[68] VelocityDB. [Online]. Available: https://velocitydb.com/VelocityGraph.aspx. Accessed: April 2016.

[69] Martın Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[70] Gregory D. Abowd, Anind K. Dey, Peter J. Brown, Nigel Davies, Mark Smith, and Pete Steggles. Towards a better understanding of context and context-awareness. In *Proceedings of the 1st International Symposium on Handheld and Ubiquitous Computing*, HUC '99, pages 304–307, London, UK, UK, 1999. Springer-Verlag.

[71] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.

[72] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: Fault-tolerant stream processing at internet scale. *Proc. VLDB Endow.*, 6(11):1033–1044, August 2013.

[73] Kerstin Altmanninger, Petra Brosch, Gerti Kappel, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. Why model versioning research is needed!? an experience report. In *Proceedings of the Joint MoDSE-MC-CM 2009 Workshop*, 2009.

[74] Sebastian Altmeyer and Nicolas Navet. Towards a declarative modeling and execution framework for real-time systems. *SIGBED Rev.*, 13(2):30–33, April 2016.

[75] S. Massoud Amin and B. F. Wollenberg. Toward a smart grid: power delivery for the 21st century. *IEEE Power and Energy Magazine*, 3(5):34–41, Sept 2005.

[76] Lisa Amini, Henrique Andrade, Ranjita Bhagwan, Frank Eskesen, Richard King, Philippe Selo, Yoonho Park, and Chitra Venkatramani. Spc: A distributed, scalable platform for data mining. In *Proceedings of the 4th International Workshop on Data Mining Standards, Services and Platforms*, DMSSP '06, pages 27–37, New York, NY, USA, 2006. ACM.

[77] Marcelo Arenas and Leopoldo Bertossi. Hypothetical temporal reasoning in databases. *Journal of Intelligent Information Systems*, 19(2):231–259.

[78] Marcelo Arenas and Leopoldo Bertossi. Hypothetical temporal queries in databases. In *Proceedings "ACM SIGMOD/PODS 5th International Workshop on Knowledge Representation meets Databases (KRDB'98): Innovative Application Programming and Query Interfaces*. Citeseer, 1998.

[79] Resmi Ariyattu and François Taïani. Filament : a cohort construction service for decentralized collaborative editing platforms. In *Compas 2016*, Lorient, France, July 2016.

[80] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015.

[81] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, pages 1–16, New York, NY, USA, 2002. ACM.

[82] Bahman Bahmani, Ravi Kumar, Mohammad Mahdian, and Eli Upfal. Pagerank on an evolving graph. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '12, pages 24–32, New York, NY, USA, 2012. ACM.

[83] Paul Baker, Shiou Loh, and Frank Weil. Model-driven engineering in a large industrial context — motorola case study. In *Proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems*, MoDELS'05, pages 476–491, Berlin, Heidelberg, 2005. Springer-Verlag.

[84] Matthias Baldauf, Schahram Dustdar, and Florian Rosenberg. A survey on context-aware systems. *Int. J. Ad Hoc Ubiquitous Comput.*, 2(4), 2007.

[85] Andrey Balmin, Thanos Papadimitriou, and Yannis Papakonstantinou. Hypothetical queries in an olap environment. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 220–231, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[86] Cajsa Bartusch, Monica Odlare, Fredrik Wallin, and Lars Wester. Exploring variance in residential electricity consumption: Household features and building properties. *Applied Energy*, 92(0):637 – 643, 2012.

[87] Sotirios Beis, Symeon Papadopoulos, and Yiannis Kompatsiaris. *New Trends in Database and Information Systems II: Selected papers of the 18th East European Conference on Advances in Databases and Information Systems and Associated Satellite Events, ADBIS 2014 Ohrid, Macedonia, September 7-10, 2014 Proceedings II*, chapter Benchmarking Graph Databases on the Problem of Community Detection, pages 3–14. Springer International Publishing, Cham, 2015.

[88] Nelly Bencomo, Robert B. France, Betty H. C. Cheng, and Uwe Aßmann, editors. *Models@run.time - Foundations, Applications, and Roadmaps [Dagstuhl Seminar 11481, November 27 - December 2, 2011]*, volume 8378 of *Lecture Notes in Computer Science*. Springer, 2014.

[89] Amine Benelallam, Abel Gómez, Gerson Sunyé, Massimo Tisi, and David Launay. Neo4emf, a scalable persistence layer for emf models. In *Proceedings of the 10th European Conference on Modelling Foundations and Applications - Volume 8569*, pages 230–241, New York, NY, USA, 2014. Springer-Verlag New York, Inc.

[90] Amine Benelallam, Abel Gómez, Massimo Tisi, and Jordi Cabot. Distributed model-to-model transformation with atl on mapreduce. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2015, pages 37–48, New York, NY, USA, 2015. ACM.

[91] David Benyon. *Information and Data Modelling*. McGraw-Hill Higher Education, 2nd edition, 1996.

[92] J. W. Berry, B. Hendrickson, S. Kahan, and P. Konecny. Software and algorithms for graph queries on multithreaded architectures. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–14, March 2007.

[93] Pramod Bhatotia, Umut A Acar, Flavio P Junqueira, and Rodrigo Rodrigues. Slider: incremental sliding window analytics. In *Proceedings of the 15th International Middleware Conference*, pages 61–72. ACM, 2014.

[94] Albert Bifet, Geoff Holmes, Richard Kirkby, and Bernhard Pfahringer. Moa: Massive online analysis. *The Journal of Machine Learning Research*, 11:1601–1604, 2010.

[95] Christopher M. Bishop. Model-based machine learning. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 371(1984), 2012.

[96] G. Blair, N. Bencomo, and R. B. France. Models@ run.time. *Computer*, 42(10):22–27, Oct 2009.

[97] Gordon Blair, Yérom-David Bromberg, Geoff Coulson, Yehia Elkhatib, Laurent Réveillère, Heverson B. Ribeiro, Etienne Rivière, and François Taïani. Holons: Towards a systematic approach to composing systems of systems. In *Proceedings of the 14th International Workshop on Adaptive and Reflective Middleware*, ARM 2015, pages 5:1–5:6, New York, NY, USA, 2015. ACM.

[98] X. Blanc, I. Mounier, A. Mougenot, and T. Mens. Detecting model inconsistency through operation-based model construction. In *Proc. 30th Int. Conf. Software Engineering*, pages 511–520, 2008.

[99] Harold Boley. Directed recursive labelnode hypergraphs: A new representation-language. *Artificial Intelligence*, 9(1):49 – 85, 1977.

[100] Flavio Bonomi, Rodolfo Milito, Preethi Natarajan, and Jiang Zhu. *Big Data and Internet of Things: A Roadmap for Smart Environments*, chapter Fog Computing: A Platform for Internet of Things and Analytics, pages 169–186. Springer International Publishing, Cham, 2014.

[101] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers, 1st edition, 2012.

[102] Petra Brosch, Gerti Kappel, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. An introduction to model versioning. In *Proceedings of the 12th International Conference on Formal Methods for the Design of Computer, Communication, and Software Systems: Formal Methods for Model-driven Engineering*, SFM'12, pages 336–398, Berlin, Heidelberg, 2012. Springer-Verlag.

[103] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. Haloop: Efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3(1-2):285–296, September 2010.

[104] Franck Budinsky, David Steinberg, and Raymond Ellersick. *Eclipse Modeling Framework : A Developer's Guide.* 2003.

[105] D Carstoiu, A Cernian, and A Olteanu. Hadoop hbase-0.20. 2 performance evaluation. In *New Trends in Information Science and Service Science (NISS), 2010 4th International Conference on*, pages 84–87. IEEE, 2010.

[106] Rick Cattell. Scalable sql and nosql data stores. *SIGMOD Rec.*, 39(4):12–27, May 2011.

[107] Ciro Cattuto, Marco Quaggiotto, André Panisson, and Alex Averbuch. Time-varying social networks in a graph database: A neo4j use case. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 11:1–11:6, New York, NY, USA, 2013. ACM.

[108] J. C. Cepeda, D.O. Ramirez, and D.G. Colome. Probabilistic-based overload estimation for real-time smart grid vulnerability assessment. In *Transmission and Distribution: Latin America Conf. and Expo. (T D-LA), 2012 6th IEEE/PES*, pages 1–8, 2012.

[109] C. Cetina, P. Giner, J. Fons, and V. Pelechano. Autonomic computing through reuse of variability models at runtime: The case of smart homes. *Computer*, 42(10):37–43, Oct 2009.

[110] Kin-Pong Chan and Ada Wai-Chee Fu. Efficient time series matching by wavelets. In *Data Engineering, 1999. Proceedings., 15th International Conference on*, pages 126–133, Mar 1999.

[111] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[112] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: Taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 85–98, New York, NY, USA, 2012. ACM.

[113] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at facebook-scale. *Proc. VLDB Endow.*, 8(12):1804–1815, August 2015.

[114] Morakot Choetkiertikul, Hoa Khanh Dam, Truyen Tran, and Aditya Ghose. Predicting delays in software projects using networked classification (t). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE '15, pages 353–364, Washington, DC, USA, 2015. IEEE Computer Society.

[115] James Clifford and David S. Warren. Formal semantics for time in databases. *ACM Trans. Database Syst.*, 8(2):214–254, June 1983.

[116] E. F. Codd, S. B. Codd, and C. T. Salley. Providing OLAP (On-Line Analytical Processing) to User-Analysts: An IT Mandate. E. F. Codd and Associates, 1993.

[117] Jeffrey Cohen, Brian Dolan, Mark Dunlap, Joseph M. Hellerstein, and Caleb Welton. Mad skills: new analysis practices for big data. *Proceedings VLDB Endowment*, 2(2):1481–1492, August 2009.

[118] Debora Coll-Mayor, Mia Paget, and Eric Lightner. Future intelligent power grids: Analysis of the vision in the european union and the united states. *Energy Policy*, 35(4):2453 – 2465, 2007.

[119] Benoit Combemale, Xavier Thirioux, and Benoit Baudry. Formally defining and iterating infinite models. In *Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems*, MODELS'12, pages 119–133, Berlin, Heidelberg, 2012. Springer-Verlag.

[120] S. S. Conn. Oltp and olap data integration: a review of feasible implementation methods and architectures for real time data analysis. In *Proceedings. IEEE SoutheastCon, 2005.*, pages 515–520, April 2005.

[121] Daniel Crankshaw, Peter Bailis, Joseph E Gonzalez, Haoyuan Li, Zhao Zhang, Michael J Franklin, Ali Ghodsi, and Michael I Jordan. The missing piece in complex analytics: Low latency, scalable model management and serving with velox. *arXiv preprint arXiv:1409.3809*, 2014.

[122] Jesús Sánchez Cuadrado and Juan de Lara. Streaming model transformations: Scenarios, challenges and initial solutions. In *Theory and Practice of Model Transformations*, pages 1–16. Springer, 2013.

[123] C Daly. Emfatic language reference. `http://www.eclipse.org/epsilon/doc/articles/emfatic/`, 2004.

[124] Gwendal Daniel, Gerson Sunyé, and Jordi Cabot. Mogwaï: a Framework to Handle Complex Queries on Large Models. In *International Conference on Research Challenges in Information Science (RCIS 2016)*, Grenoble, France, June 2016.

[125] Gwendal Daniel, Gerson Sunyé, and Jordi Cabot. UMLtoGraphDB: Mapping Conceptual Schemas to Graph Databases. In *The 35th International Conference on Conceptual Modeling (ER2016)*, Gifu, Japan, November 2016.

[126] Ankur Dave, Alekh Jindal, Li Erran Li, Reynold Xin, Joseph Gonzalez, and Matei Zaharia. Graphframes: an integrated api for mixing graph and relational queries. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2016.

[127] István Dávid, István Ráth, and Dániel Varró. Streaming model transformations by complex event processing. In *Model-Driven Engineering Languages and Systems*, pages 68–83. Springer, 2014.

[128] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[129] Stéphane Delbruel, Davide Frey, and François Taïani. Decentralized view prediction for global content placement. In *Proceedings of the 14th International Workshop on Adaptive and Reflective Middleware*, ARM 2015, pages 10:1–10:3, New York, NY, USA, 2015. ACM.

[130] Stéphane Delbruel, Davide Frey, and François Taïani. Exploring the use of tags for georeplicated content placement. In *2016 IEEE International Conference on Cloud Engineering, IC2E 2016, Berlin, Germany, April 4-8, 2016*, pages 172–181, 2016.

[131] Stéphane Delbruel, Davide Frey, and François Taïani. Mignon: A fast decentralized content consumption estimation in large-scale distributed systems. In *Distributed Applications and Interoperable Systems - 16th IFIP WG 6.1 International Conference, DAIS 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, pages 32–46, 2016.

[132] Dursun Delen and Haluk Demirkan. Data, information and analytics as services. *Decis. Support Syst.*, 55(1):359–363, April 2013.

[133] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '87, pages 1–12, New York, NY, USA, 1987. ACM.

[134] Pedro Domingos and Geoff Hulten. Mining high-speed data streams. In *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '00, pages 71–80, New York, NY, USA, 2000. ACM.

[135] K SRIVASTAVA Durgesh and B Lekha. Data classification using support vector machine. *Journal of Theoretical and Applied Information Technology*, 12(1):1–7, 2010.

[136] Partha Dutta, Rachid Guerraoui, and Leslie Lamport. How fast can eventual synchrony lead to consensus? In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. Int. Conf. on*, pages 22–27. IEEE, 2005.

[137] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, and Geoffrey Fox. Twister: A runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 810–818, New York, NY, USA, 2010. ACM.

[138] Donia El Kateb, François Fouquet, Grégory Nain, Jorge Augusto Meira, Michel Ackerman, and Yves Le Traon. Generic cloud platform multi-objective optimization leveraging models@run.time. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 343–350, New York, NY, USA, 2014. ACM.

[139] Hugh Everett. "relative state" formulation of quantum mechanics. *Rev. Mod. Phys.*, 29:454–462, Jul 1957.

[140] Christos Faloutsos, M. Ranganathan, and Yannis Manolopoulos. Fast subsequence matching in time-series databases. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, SIGMOD '94, pages 419–429, New York, NY, USA, 1994. ACM.

[141] X. Fang, S. Misra, G. Xue, and D. Yang. Smart grid — the new and improved power grid: A survey. *IEEE Communications Surveys Tutorials*, 14(4):944–980, Fourth 2012.

[142] H. Farhangi. The path of the smart grid. *IEEE Power and Energy Magazine*, 8(1):18–28, January 2010.

[143] L. Fejoz, N. Navet, S. M. Sundharam, and S. Altmeyer. Demo abstract: Applications of the cpal language to model, simulate and program cyber-physical systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–1, April 2016.

[144] Clayton R Fink, Danielle S Chou, Jonathon J Kopecky, and Ashley J Llorens. Coarse- and fine-grained sentiment analysis of social media text. *Johns Hopkins APL Technical Digest*, 30(1):22–30, 2011.

[145] Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon Peyton Jones. Calling hell from heaven and heaven from hell. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming*, ICFP '99, pages 114–125, New York, NY, USA, 1999. ACM.

[146] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[147] François Fouquet, Grégory Nain, Brice Morin, Erwan Daubert, Olivier Barais, Noël Plouzeau, and Jean-Marc Jézéquel. An eclipse modelling framework alternative to meet the models@runtime requirements. In *Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems*, MODELS'12, pages 87–101, Berlin, Heidelberg, 2012. Springer-Verlag.

[148] François Fouquet, Erwan Daubert, Noël Plouzeau, Olivier Barais, Johann Bourcier, and Jean-Marc Jézéquel. Dissemination of reconfiguration policies on mesh networks. In *Distributed Applications and Interoperable Systems - 12th IFIP WG 6.1 International Conference, DAIS 2012, Stockholm, Sweden, June 13-16, 2012. Proceedings*, pages 16–30, 2012.

[149] Francois Fouquet, Brice Morin, Franck Fleurey, Olivier Barais, Noel Plouzeau, and Jean-Marc Jezequel. A dynamic component model for cyber physical systems. In *Proceedings of the 15th ACM SIGSOFT symposium on Component Based Software Engineering*, pages 135–144. ACM, 2012.

[150] François Fouquet, Grégory Nain, Brice Morin, Erwan Daubert, Olivier Barais, Noël Plouzeau, and Jean-Marc Jézéquel. Kevoree modeling framework (KMF): efficient modeling techniques for runtime use. *CoRR*, abs/1405.6817, 2014.

[151] Fouquet Francois, Grégory Nain, Brice Morin, Erwan Daubert, Olivier Barais, Noël Plouzeau, and Jean-Marc Jézéquel. Kevoree modeling framework (kmf): Efficient modeling techniques for runtime use. *arXiv preprint arXiv:1405.6817*, 2014.

[152] Davide Frey, Achour Mostefaoui, Matthieu Perrin, Pierre-Louis Roman, and François Taïani. Speed for the elite, consistency for the masses: differentiating eventual consistency in large-scale distributed systems. To appear in the proceedings of SRDS 2016. http://srds2016.inf.mit.bme.hu, July 2016.

[153] Francisco Javier Thayer Fábrega, Francisco Javier, and Joshua D. Guttman. Copy on write, 1995.

[154] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[155] J. C. Georgas, A. v. d. Hoek, and R. N. Taylor. Using architectural models to manage and visualize runtime adaptation. *Computer*, 42(10):52–60, Oct 2009.

[156] A.V. Gerbessiotis and L.G. Valiant. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing*, 22(2):251 – 267, 1994.

[157] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[158] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[159] B. Goertzel. Patterns, hypergraphs and embodied general intelligence. In *The 2006 IEEE International Joint Conference on Neural Network Proceedings*, pages 451–458, 2006.

[160] Abel Gomez, Massimo Tisi, Gerson Sunyé, and Jordi Cabot. Map-based transparent persistence for very large models. In *Fundamental Approaches to Software Engineering*, pages 19–34. Springer, 2015.

[161] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.

[162] Douglas Gregor and Andrew Lumsdaine. The parallel bgl: A generic library for distributed graph computations. In *In Parallel Object-Oriented Scientific Computing (POOSC*, 2005.

[163] Timothy Griffin and Richard Hull. A framework for implementing hypothetical queries. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, SIGMOD '97, pages 231–242, New York, NY, USA, 1997. ACM.

[164] Object Management Group. OMG Systems Modeling Language, Version 1.4. `http://www.omg.org/spec/SysML/1.4/`, June 2015.

[165] Christophe Guille and George Gross. A conceptual framework for the vehicle-to-grid (V2G) implementation. *Energy Policy*, 37(11):4379–4390, 2009.

[166] Pankaj Gupta, Ashish Goel, Jimmy Lin, Aneesh Sharma, Dong Wang, and Reza Zadeh. Wtf: The who to follow service at twitter. In *Proceedings of the 22Nd International Conference on World Wide Web*, WWW '13, pages 505–514, New York, NY, USA, 2013. ACM.

[167] Peter J. Haas, Paul P. Maglio, Patricia G. Selinger, and Wang Chiew Tan. Data is dead... without what-if models. *PVLDB*, 4(12):1486–1489, 2011.

[168] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009.

[169] Jiawei Han, Guozhu Dong, and Yiwen Yin. Efficient mining of partial periodic patterns in time series database. In *Data Engineering, 1999. Proceedings., 15th International Conference on*, pages 106–115, Mar 1999.

[170] Minyang Han and Khuzaima Daudjee. Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems. *Proc. VLDB Endow.*, 8(9):950–961, May 2015.

[171] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. Chronos: A graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 1:1–1:14, New York, NY, USA, 2014. ACM.

[172] Thomas Hartmann, François Fouquet, Jacques Klein, Grégory Nain, and Yves Le Traon. Reactive security for smart grids using models@run.time-based simulation and reasoning. In *Smart Grid Security - Second International Workshop, SmartGridSec 2014, Munich, Germany, February 26, 2014, Revised Selected Papers*, pages 139–153, 2014.

[173] Thomas Hartmann, François Fouquet, Jacques Klein, Yves Le Traon, Alexander Pelov, Laurent Toutain, and Tanguy Ropitault. Generating realistic smart grid communication topologies based on real-data. In *2014 IEEE International Conference on Smart Grid Communications, SmartGridComm 2014, Venice, Italy, November 3-6, 2014*, pages 428–433, 2014.

[174] Thomas Hartmann, François Fouquet, Grégory Nain, Brice Morin, Jacques Klein, Olivier Barais, and Yves Le Traon. A native versioning concept to support historized models at runtime. In *Model-Driven Engineering Languages and Systems - 17th International Conference, MODELS 2014, Valencia, Spain, September 28 - October 3, 2014. Proceedings*, pages 252–268, 2014.

[175] Thomas Hartmann, François Fouquet, Grégory Nain, Brice Morin, Jacques Klein, and Yves Le Traon. Model-based time-distorted contexts for efficient temporal reasoning. In *The 26th International Conference on Software Engineering and Knowledge Engineering, SEKE 2014, Vancouver, BC, Canada, July 1-3, 2014.*, pages 746–747, 2014.

[176] Thomas Hartmann, François Fouquet, Grégory Nain, Brice Morin, Jacques Klein, and Yves Le Traon. Reasoning at runtime using time-distorted contexts: A models@run.time based approach. In *The 26th International Conference on Software Engineering and Knowledge Engineering, SEKE 2014, Vancouver, BC, Canada, July 1-3, 2013.*, pages 586–591, 2014.

[177] Thomas Hartmann, Assaad Moawad, François Fouquet, Grégory Nain, Jacques Klein, and Yves Le Traon. Stream my models: Reactive peer-to-peer distributed models@run.time. In *18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MoDELS 2015, Ottawa, ON, Canada, September 30 - October 2, 2015*, pages 80–89, 2015.

[178] Thomas Hartmann, Assaad Moawad, François Fouquet, Yves Reckinger, Jacques Klein, and Yves Le Traon. Near real-time electric load approximation in low voltage cables of smart grids with models@run.time. In *Proceedings of the 31th Annual ACM Symposium on Applied Computing, Pisa, Italy, April 4-8, 2016.*

[179] Thomas Hartmann, Assaad Moawad, François Fouquet, Yves Reckinger, Tejeddine Mouelhi, Jacques Klein, and Yves Le Traon. Suspicious electric consumption detection based on multi-profiling using live machine learning. In *2015 IEEE International Conference on Smart Grid Communications, SmartGridComm 2015, Miami, USA, November 2-5, 2015.*

[180] Geoffrey Hecht, Benomar Omar, Romain Rouvoy, Naouel Moha, and Laurence Duchien. Tracking the Software Quality of Android Applications along their Evolution. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015)*, page 12. IEEE, November 2015.

[181] Karen Henricksen, Jadwiga Indulska, and Andry Rakotonirainy. Modeling context information in pervasive computing systems. In *Proc. 1st Int. Conf. Pervasive Computing*, Pervasive '02, pages 167–180, 2002.

[182] Shohei Hido, Seiya Tokui, and Satoshi Oda. Jubatus: An open source platform for distributed online machine learning. In *NIPS 2013 Workshop on Big Learning, Lake Tahoe*, 2013.

[183] P. HUBRAL. Time migration—some ray theoretical aspects*. *Geophysical Prospecting*, 25(4):738–745, 1977.

[184] Theo Hug, Martin Lindner, and Peter A Bruck. Microlearning: Emerging concepts, practices and technologies after e-learning. *Proceedings of Microlearning*, 5, 2005.

[185] John Hutchinson, Mark Rouncefield, and Jon Whittle. Model-driven engineering practices in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 633–642, New York, NY, USA, 2011. ACM.

[186] IBM, Paul Zikopoulos, and Chris Eaton. *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*. McGraw-Hill Osborne Media, 1st edition, 2011.

[187] Borislav Iordanov. Hypergraphdb: A generalized graph database. In *Proceedings of the 2010 International Conference on Web-age Information Management*, WAIM'10, pages 25–36, Berlin, Heidelberg, 2010. Springer-Verlag.

[188] Anand Padmanabha Iyer, Li Erran Li, Tathagata Das, and Ion Stoica. Time-evolving graph processing at scale. 2016.

[189] Javier Luis Cánovas Izquierdo and Jordi Cabot. Collaboro: A collaborative (meta) modeling tool. Technical report, PeerJ Preprints, 2016.

[190] Adam Jacobs. The pathologies of big data. *Communications of the ACM*, 52(8):36–44, 2009.

[191] A. J. Jara, D. Genoud, and Y. Bocchi. Big data for cyber physical systems: An analysis of challenges, solutions and opportunities. In *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2014 Eighth International Conference on*, pages 376–380, July 2014.

[192] U. Kang, Hanghang Tong, Jimeng Sun, Ching-Yung Lin, and Christos Faloutsos. Gbase: A scalable and general graph management system. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, pages 1091–1099, New York, NY, USA, 2011. ACM.

[193] U. Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. Pegasus: A peta-scale graph mining system implementation and observations. In *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining*, ICDM '09, pages 229–238, Washington, DC, USA, 2009. IEEE Computer Society.

[194] Pradeeban Kathiravelu, Leila Sharifi, and Luís Veiga. Cassowary: Middleware platform for context-aware smart buildings with software-defined sensor networks. In *Proceedings of the 2Nd Workshop on Middleware for Context-Aware Applications in the IoT*, M4IoT 2015, pages 1–6, New York, NY, USA, 2015. ACM.

[195] Stuart Kent. Model driven engineering. In *Proceedings of the Third International Conference on Integrated Formal Methods*, IFM '02, pages 286–298, London, UK, UK, 2002. Springer-Verlag.

[196] Eamonn Keogh, Stefano Lonardi, and Bill 'Yuan-chi' Chiu. Finding surprising patterns in a time series database in linear time and space. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '02, pages 550–556, New York, NY, USA, 2002. ACM.

[197] A.-M. Kermarrec and F. Taïani. Want to scale in centralized systems? think p2p. *Journal of Internet Services and Applications*, 6(1), 2015. cited By 0.

[198] Anne-Marie Kermarrec, François Taïani, and Juan M. Tirado. Scaling out link prediction with SNAPLE: 1 billion edges and beyond. In *Proceedings of the 16th Annual Middleware Conference, Vancouver, BC, Canada, December 07 - 11, 2015*, pages 247–258, 2015.

[199] Farshad Khunjush and Nikitas J. Dimopoulos. Lazy direct-to-cache transfer during receive operations in a message passing environment. In *Proceedings of the 3rd Conference on Computing Frontiers*, CF '06, pages 331–340, New York, NY, USA, 2006. ACM.

[200] U. Khurana and A. Deshpande. Efficient snapshot retrieval over historical graph data. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 997–1008, April 2013.

[201] Udayan Khurana and Amol Deshpande. Storing and analyzing historical graph data at scale. *arXiv preprint arXiv:1509.08960*, 2015.

[202] Udayan Khurana and Amol Deshpande. Storing and analyzing historical graph data at scale. In Evaggelia Pitoura, Sofian Maabout, Georgia Koutrika, Amélie Marian, Letizia Tanca, Ioana Manolescu, and Kostas Stefanidis, editors, *Proceedings of the 19th International Conference on Extending Database Technology, EDBT 2016, Bordeaux, France, March 15-16, 2016, Bordeaux, France, March 15-16, 2016.*, pages 65–76. OpenProceedings.org, 2016.

[203] Jacques Klein, Jörg Kienzle, Brice Morin, and Jean-Marc Jézéquel. Aspect model unweaving. In LNCS 5795, editor, *In 12th International Conference on Model Driven Engineering Languages and Systems (MODELS 2009)*, pages p 514–530, Denver, Colorado, USA, 2009.

[204] Maximilian Koegel and Jonas Helming. Emfstore: A model repository for emf models. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 307–308, New York, NY, USA, 2010. ACM.

[205] Robert Kohtes. *From Valence to Emotions: How Coarse Versus Fine-grained Online Sentiment Can Predict Real-world Outcomes*. Anchor Academic Publishing (aap_verlag), 2014.

[206] Dimitrios S. Kolovos, Louis M. Rose, Nicholas Drivalos Matragkas, Richard F. Paige, Esther Guerra, Jesús Sánchez Cuadrado, Juan de Lara, István Ráth, Dániel Varró, Massimo Tisi, and Jordi Cabot. A research roadmap towards achieving scalability in model driven engineering. In *Proceedings of the Workshop on Scalability in Model Driven Engineering, Budapest, Hungary, June 17, 2013*, page 2, 2013.

[207] D.S. Kolovos, D. Di Ruscio, A. Pierantonio, and R.F. Paige. Different models for model matching: An analysis of approaches to support model differencing. In *Comparison and Versioning of Software Models, 2009. CVSM '09. ICSE Workshop on*, pages 1–6, May 2009.

[208] Vassilis Kostakos. Temporal graphs. *Physica A: Statistical Mechanics and its Applications*, 388(6):1007–1023, 2009.

[209] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, pages 1–7, 2011.

[210] Matej Kristan and Aleš Leonardis. Multivariate online kernel density estimation. In *Computer Vision Winter Workshop*, pages 77–86, 2010.

[211] Matej Kristan, Danijel Skočaj, and Ales Leonardis. Online kernel density estimation for interactive learning. *Image and Vision Computing*, 28(7):1106–1116, 2010.

[212] Krishna Kulkarni and Jan-Eike Michels. Temporal features in sql:2011. *SIGMOD Rec.*, 41(3):34–43, October 2012.

[213] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 239–250, New York, NY, USA, 2015. ACM.

[214] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 31–46, Berkeley, CA, USA, 2012. USENIX Association.

[215] Alan G. Labouseur, Jeremy Birnbaum, Paul W. Olsen, Jr., Sean R. Spillane, Jayadevan Vijayan, Jeong-Hyon Hwang, and Wook-Shin Han. The g* graph database: Efficiently managing large distributed dynamic graphs. *Distrib. Parallel Databases*, 33(4):479–514, December 2015.

[216] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[217] Ora Lassila and Ralph R. Swick. Resource Description Framework (RDF) Model and Syntax Specification. W3c recommendation, W3C, 1999.

[218] Steve LaValle, Eric Lesser, Rebecca Shockley, Michael S Hopkins, and Nina Kruschwitz. Big data, analytics and the path from insights to value. 52(2):20–32, 2011.

[219] Tien-Duy B. Le, Xuan-Bach D. Le, David Lo, and Ivan Beschastnikh. Synergizing specification miners through model fissions and fusions (t). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE '15, pages 115–125, Washington, DC, USA, 2015. IEEE Computer Society.

[220] N. Leavitt. Complex-event processing poised for growth. *Computer*, 42(4):17–20, April 2009.

[221] E. A. Lee. Cyber physical systems: Design challenges. In *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC)*, pages 363–369, May 2008.

[222] Jay Lee, Behrad Bagheri, and Hung-An Kao. A cyber-physical systems architecture for industry 4.0-based manufacturing systems. *Manufacturing Letters*, 3:18–23, 2015.

[223] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 6:1–6:15, New York, NY, USA, 2014. ACM.

[224] Jessica Lin, Eamonn Keogh, Stefano Lonardi, and Bill Chiu. A symbolic representation of time series, with implications for streaming algorithms. In *Proceedings of the 8th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, DMKD '03, pages 2–11, New York, NY, USA, 2003. ACM.

[225] Shen Lin, François Taïani, and Gordon S. Blair. Facilitating gossip programming with the gossipkit framework. In *Distributed Applications and Interoperable Systems, 8th IFIP WG 6.1 International Conference, DAIS 2008, Oslo, Norway, June 4-6, 2008. Proceedings*, pages 238–252, 2008.

[226] C.-H. Liu, K.-L. Chang, J.J.-Y. Chen, and S.-C. Hung. Ontology-based context representation and reasoning using owl and swrl. In *Communication Networks and Services Research Conf. (CNSR), 2010 8th Annu.*, pages 215–220, 2010.

[227] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, April 2012.

[228] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new framework for parallel machine learning. *CoRR*, abs/1006.4990, 2010.

[229] Yucheng Low, Joseph E. Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M. Hellerstein. Graphlab: A new framework for parallel machine learning. *CoRR*, abs/1408.2041, 2014.

[230] Wenpeng Luan, D. Sharp, and S. Lancashire. Smart grid communication network capacity planning for power utilities. In *Transmission and Distribution Conference and Exposition, 2010 IEEE PES*, pages 1–4, April 2010.

[231] Nadeem Mahmood, S. M. Aqil Burney, and Kamran Ahsan. A logical temporal relational data model. *CoRR*, abs/1002.1143, 2010.

[232] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[233] Norbert Martínez-Bazan, M. Ángel Águila Lorente, Victor Muntés-Mulero, David Dominguez-Sal, Sergio Gómez-Villamor, and Josep-L. Larriba-Pey. Efficient graph management based on bitmap indices. In *Proceedings of the 16th International Database Engineering & Applications Sysmposium*, IDEAS '12, pages 110–119, New York, NY, USA, 2012. ACM.

[234] Norbert Martinez-Bazan, Sergio Gomez-Villamor, and Francesc Escale-Claveras. Dex: A high-performance graph database management system. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering Workshops*, ICDEW '11, pages 124–127, Washington, DC, USA, 2011. IEEE Computer Society.

[235] Ruben Mayer, Christian Mayer, Muhammad Adnan Tariq, and Kurt Rothermel. Graphcep: Real-time data analytics using parallel complex event and graph processing. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, DEBS '16, pages 309–316, New York, NY, USA, 2016. ACM.

[236] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 53–65, London, UK, UK, 2002. Springer-Verlag.

[237] Xiangrui Meng, Joseph Bradley, B Yuvaz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, D Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *JMLR*, 17(34):1–7, 2016.

[238] Meta object facility (MOF) 2.5 core specification, 2015. Version 2.5.

[239] Youshan Miao, Wentao Han, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Enhong Chen, and Wenguang Chen. Immortalgraph: A system for storage and analysis of temporal graphs. *Trans. Storage*, 11(3):14:1–14:34, July 2015.

[240] Justin J Miller. Graph database applications and concepts with neo4j. In *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA March 23rd-24th*, 2013.

[241] Daniele Miorandi, Sabrina Sicari, Francesco De Pellegrini, and Imrich Chlamtac. Internet of things: Vision, applications and research challenges. *Ad Hoc Networks*, 10(7):1497 – 1516, 2012.

[242] Assaad Moawad. *Towards Ambient Intelligent Applications Using Models@ run. time And Machine Learning For Context-Awareness*. PhD thesis, University of Luxembourg, 2016.

[243] Assaad Moawad, Thomas Hartmann, François Fouquet, Jacques Klein, and Yves Le Traon. Adaptive blurring of sensor data to balance privacy and utility for ubiquitous services. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015*, pages 2271–2278, 2015.

[244] Assaad Moawad, Thomas Hartmann, François Fouquet, Grégory Nain, Jacques Klein, and Johann Bourcier. Polymer - A model-driven approach for simpler, safer, and evolutive multi-objective optimization development. In *MODELSWARD 2015 - Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development, ESEO, Angers, Loire Valley, France, 9-11 February, 2015.*, pages 286–293, 2015.

[245] Assaad Moawad, Thomas Hartmann, François Fouquet, Grégory Nain, Jacques Klein, and Yves Le Traon. Beyond discrete modeling: A continuous and efficient model for iot. In *18th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MoDELS 2015, Ottawa, ON, Canada, September 30 - October 2, 2015*, pages 90–99, 2015.

[246] Brice Morin, Olivier Barais, Jean-Marc Jezequel, Franck Fleurey, and Arnor Solberg. Models@ run.time to support dynamic adaptation. *Computer*, 42(10):44–51, October 2009.

[247] Boris Motik. Representing and querying validity time in {RDF} and owl: A logic-based approach. *Web Semantics: Science, Services and Agents on the World Wide Web*, 12 - 13:3 – 21, 2012. Reasoning with context in the Semantic Web.

[248] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM.

[249] Nicolas Navet, Loïc Fejoz, Lionel Havet, and Altmeyer Sebastian. Lean model-driven development through model-interpretation: the cpal design flow. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, 2016.

[250] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *2010 IEEE International Conference on Data Mining Workshops*, pages 170–177, Dec 2010.

[251] Peter Norvig. Artificial intelligence. *NewScientist*, (27), November 2012.

[252] Object Management Group. OMG Common Object Request Broker Architecture, Version 3.3. `http://www.omg.org/spec/CORBA/3.3/`, November 2012.

[253] Object Management Group. OMG Unified Modeling Language, Version 2.5. `http://www.omg.org/spec/UML/2.5/PDF`, March 2015.

[254] National Institute of Standards and Technology. NIST framework and roadmap for smart grid interoperability standards, release 3.0, 2014.

[255] Tony Ohmann, Michael Herzberg, Sebastian Fiss, Armand Halbert, Marc Palyart, Ivan Beschastnikh, and Yuriy Brun. Behavioral resource-aware model inference. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 19–30, New York, NY, USA, 2014. ACM.

[256] Kalil T Swain Oldham. *The Doctrine of Description: Gustav Kirchhoff, Classical Physics, and the" purpose of All Science" in 19th-century Germany.* ProQuest, 2008.

[257] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proc. USENIX Annual Tech. Conf.*, pages 305–320, 2014.

[258] Avner Ottensooser, Alan Fekete, Hajo A Reijers, Jan Mendling, and Con Menictas. Making sense of business process descriptions: An experimental comparison of graphical and textual notations. *Journal of Systems and Software*, 85(3):596–606, 2012.

[259] Javier Espinazo Pagán, Jesúss Sánchez Cuadrado, and Jesús García Molina. Morsa: A scalable approach for persisting and accessing large models. In *Model Driven Engineering Languages and Systems*, pages 77–92. Springer, 2011.

[260] Sang Hyun Park, So Hee Won, Jong Bong Lee, and Sung Woo Kim. Smart home — digitally engineered domestic life. *Personal Ubiquitous Comput.*, 7(3-4):189–196, July 2003.

[261] Mikko Perttunen, Jukka Riekki, and Ora Lassila. Context representation and reasoning in pervasive computing: a review. *Int. Journal of Multimedia and Ubiquitous Engineering*, pages 1–28, 2009.

[262] Russell Power and Jinyang Li. Piccolo: Building fast, distributed programs with partitioned tables. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 293–306, Berkeley, CA, USA, 2010. USENIX Association.

[263] Vijayan Prabhakaran, Ming Wu, Xuetian Weng, Frank McSherry, Lidong Zhou, and Maya Haridasan. Managing large graphs on multi-cores with graph awareness. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 4–4, Berkeley, CA, USA, 2012. USENIX Association.

[264] Dan Pritchett. Base: An acid alternative. *Queue*, 6(3):48–55, May 2008.

[265] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, June 1990.

[266] Lu Qin, Jeffrey Xu Yu, Lijun Chang, Hong Cheng, Chengqi Zhang, and Xuemin Lin. Scalable big graph processing in mapreduce. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 827–838, New York, NY, USA, 2014. ACM.

[267] R. Rajkumar, I. Lee, L. Sha, and J. Stankovic. Cyber-physical systems: The next computing revolution. In *Design Automation Conference (DAC), 2010 47th ACM/IEEE*, pages 731–736, June 2010.

[268] Ragunathan (Raj) Rajkumar, Insup Lee, Lui Sha, and John Stankovic. Cyber-physical systems: The next computing revolution. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 731–736, New York, NY, USA, 2010. ACM.

[269] István Ráth, Gergely Varró, and Dániel Varró. Change-driven model transformations. In Andy Schürr and Bran Selic, editors, *Model Driven Engineering Languages and Systems*, volume 5795 of *Lecture Notes in Computer Science*, pages 342–356. Springer Berlin Heidelberg, 2009.

[270] A. Ray. Autonomous perception and decision-making in cyber-physical systems. In *Computer Science Education (ICCSE), 2013 8th International Conference on*, pages 1–10, April 2013.

[271] Alberto Rodrigues da Silva. Model-driven engineering. *Comput. Lang. Syst. Struct.*, 43(C):139–155, October 2015.

[272] Marko A. Rodriguez. The gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages*, DBPL 2015, pages 1–10, New York, NY, USA, 2015. ACM.

[273] E. Rose and A. Segev. *TOODM: A temporal object-oriented data model with temporal constraints*. Apr 1991.

[274] J. Rothenberg. Artificial intelligence, simulation & modeling. chapter The Nature of Modeling, pages 75–92. John Wiley & Sons, Inc., New York, NY, USA, 1989.

[275] Amitabha Roy, Laurent Bindschaedler, Jasmina Malicevic, and Willy Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 410–424, New York, NY, USA, 2015. ACM.

[276] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 472–488, New York, NY, USA, 2013. ACM.

[277] Sebnem Rusitschka, Christoph Doblander, Christoph Goebel, and Hans-Arno Jacobsen. Adaptive middleware for real-time prescriptive analytics in large scale power systems. In *Proceedings of the Industrial Track of the 13th ACM/IFIP/USENIX International Middleware Conference*, Middleware Industry '13, pages 5:1–5:6, New York, NY, USA, 2013. ACM.

[278] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[279] Semih Salihoglu and Jennifer Widom. Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, SSDBM, pages 22:1–22:12, New York, NY, USA, 2013. ACM.

[280] M. Satyanarayanan, P. Simoens, Y. Xiao, P. Pillai, Z. Chen, K. Ha, W. Hu, and B. Amos. Edge analytics in the internet of things. *IEEE Pervasive Computing*, 14(2):24–31, Apr 2015.

[281] Douglas C Schmidt. Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY-*, 39(2):25, 2006.

[282] E Schrödinger. The present status of quantum mechanics. *Die Naturwissenschaften*, 23(48):1–26, 1935.

[283] Arie Segev and Arie Shoshani. Logical modeling of temporal data. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, SIGMOD '87, New York, NY, USA, 1987.

[284] Arie Segev and Arie Shoshani. The representation of a temporal data model in the relational environment. In *Proceedings of the 4th International Conference on Statistical and Scientific Database Management*, SSDBM'1988, pages 39–61, London, UK, UK, 1988. Springer-Verlag.

[285] Ed Seidewitz. What models mean. *IEEE Softw.*, 20(5):26–32, September 2003.

[286] Bran Selic. The pragmatics of model-driven development. *IEEE Softw.*, 20(5):19–25, September 2003.

[287] Peter Pin shan Chen. The entity-relationship model: Toward a unified view of data. *ACM Trans. Database Syst.*, 1:9–36, 1976.

[288] Bin Shao, Haixun Wang, and Yatao Li. Trinity: A distributed graph engine on a memory cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 505–516, New York, NY, USA, 2013. ACM.

[289] A. Sheth, C. Thomas, and P. Mehra. Continuous semantics to analyze real-time data. *IEEE Internet Computing*, 14(6):84–89, Nov 2010.

[290] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, May 2010.

[291] Laurynas Siksnys. *Towards Prescriptive Analytics in Cyber-Physical Systems*. PhD thesis, Dresden University of Technology, 2015.

[292] Mandavilli Srinivas and Lalit M Patnaik. Genetic algorithms: A survey. *Computer*, 27(6):17–26, 1994.

[293] J. A. Stankovic. Research directions for the internet of things. *IEEE Internet of Things Journal*, 1(1):3–9, Feb 2014.

[294] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[295] Peter Stone and Manuela Veloso. Multiagent systems: A survey from a machine learning perspective. *Autonomous Robots*, 8(3):345–383, 2000.

[296] Thomas Strang and Claudia L. Popien. A context modeling survey. In *UbiComp 1st Int. Workshop on Advanced Context Modelling, Reasoning and Management*, pages 31–41, 2004.

[297] Philip Stutz, Abraham Bernstein, and William Cohen. Signal/collect: Graph algorithms for the (semantic) web. In *Proceedings of the 9th International Semantic Web Conference on The Semantic Web - Volume Part I*, ISWC'10, pages 764–780, Berlin, Heidelberg, 2010. Springer-Verlag.

[298] Alistair Sutcliffe and Pete Sawyer. Requirements elicitation: Towards the unknown unknowns. In *Requirements Engineering Conference (RE), 2013 21st IEEE International*, pages 92–104. IEEE, 2013.

[299] Gábor Szárnyas, Benedek Izsó, István Ráth, Dénes Harmath, Gábor Bergmann, and Dániel Varró. Incquery-d: A distributed incremental model query framework in the cloud. In *Model-Driven Engineering Languages and Systems*, pages 653–669. Springer International Publishing, 2014.

[300] Gabriele Taentzer, Claudia Ermel, Philip Langer, and Manuel Wimmer. A fundamental approach to model versioning based on graph modifications: From theory to implementation. *Softw. Syst. Model.*, 13(1):239–272, February 2014.

[301] Amir Taherkordi, Frederic Loiret, Romain Rouvoy, and Frank Eliassen. Optimizing sensor network reprogramming via in situ reconfigurable components. *ACM Trans. Sen. Netw.*, 9(2):14:1–14:33, April 2013.

[302] François Taïani. *Some Contributions to The Programming of Large-Scale Distributed Systems: Mechanisms, Abstractions, and Tools*. Accreditation to supervise research, Université Rennes 1, November 2011.

[303] J. Taneja, R. Katz, and D. Culler. Defining cps challenges in a sustainable electricity grid. In *Cyber-Physical Systems (ICCPS), 2012 IEEE/ACM Third International Conference on*, pages 119–128, April 2012.

[304] F. Taïani, S. Lin, and G. S. Blair. Gossipkit: A unified componentframework for gossip. *IEEE Transactions on Software Engineering*, 40(2):123–136, Feb 2014.

[305] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive: A warehousing solution over a map-reduce framework. *Proc. VLDB Endow.*, 2(2):1626–1629, August 2009.

[306] Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From "think like a vertex" to "think like a graph". *Proc. VLDB Endow.*, 7(3):193–204, November 2013.

[307] H. W. K. Tom, G. D. Aumiller, and C. H. Brito-Cruz. Time-resolved study of laser-induced disorder of si surfaces. *Phys. Rev. Lett.*, 60:1438–1441, Apr 1988.

[308] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, Nikunj Bhagat, Sailesh Mittal, and Dmitriy Ryaboy. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 147–156, New York, NY, USA, 2014. ACM.

[309] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[310] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[311] Luís Veiga, Rodrigo Bruno, and Paulo Ferreira. Asynchronous complete garbage collection for graph data stores. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, pages 112–124, New York, NY, USA, 2015. ACM.

[312] Shivaram Venkataraman, Erik Bodzsar, Indrajit Roy, Alvin AuYoung, and Robert S. Schreiber. Presto: Distributed machine learning and graph processing with sparse matrices. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 197–210, New York, NY, USA, 2013. ACM.

[313] Michael Vierhauser, Rick Rabiser, Paul Grunbacher, and Alexander Egyed. Developing a dsl-based approach for event-based monitoring of systems of systems: Experiences and lessons learned (e). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 715–725. IEEE, 2015.

[314] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, January 2009.

[315] World Wide Web Consortium W3C. Owl 2 web ontology language. structural specification and functional-style syntax, 2009.

[316] Matt P Wand and M Chris Jones. *Kernel smoothing*. Crc Press, 1994.

[317] Guozhang Wang, Wenlei Xie, Alan J. Demers, and Johannes Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR 2013, Sixth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 6-9, 2013, Online Proceedings*, 2013.

[318] M. N. Wernick, Y. Yang, J. G. Brankov, G. Yourganov, and S. C. Strother. Machine learning in medical imaging. *IEEE Signal Processing Magazine*, 27(4):25–38, July 2010.

[319] Tom White. *Hadoop: The definitive guide.* " O'Reilly Media, Inc.", 2012.

[320] Jon Whittle, John Hutchinson, Mark Rouncefield, Håkan Burden, and Rogardt Heldal. ustrial adoption of model-driven engineering: Are the tools. In Ana Moreira, Bernhard Schätz, Jeff Gray, Antonio Vallecillo, and Peter J. Clarke, editors, *MoDELS*, volume 8107 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2013.

[321] Jie Wu, ZhiHui Lu, BiSheng Liu, and Shiyong Zhang. Peercdn: A novel p2p network assisted streaming content delivery network scheme. In *Computer and Information Technology, 2008. CIT 2008. 8th IEEE International Conference on*, pages 601–606. IEEE, 2008.

[322] Ming Wu, Fan Yang, Jilong Xue, Wencong Xiao, Youshan Miao, Lan Wei, Haoxiang Lin, Yafei Dai, and Lidong Zhou. Gram: Scaling graph computation to the trillions. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 408–421, New York, NY, USA, 2015. ACM.

[323] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 2:1–2:6, New York, NY, USA, 2013. ACM.

[324] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *CoRR*, abs/1205.6233, 2012.

[325] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 1–14, Berkeley, CA, USA, 2008. USENIX Association.

[326] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[327] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[328] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM.

[329] Stanley B. Zdonik, Michael Stonebraker, Mitch Cherniack, Ugur Çetintemel, Magdalena Balazinska, and Hari Balakrishnan. The aurora and medusa projects. *IEEE Data Eng. Bull.*, 26(1):3–10, 2003.

[330] Bo Zhang and Ling Zhang. Multi-granular representation-the key to machine intelligence. In *Intelligent System and Knowledge Engineering, 2008. ISKE 2008. 3rd International Conference on*, volume 1, pages 7–7, Nov 2008.

[331] Bo Zhang and Ling Zhang. Multi-granular representation-the key to machine intelligence. In *Intelligent System and Knowledge Engineering, 2008. ISKE 2008. 3rd International Conference on*, volume 1, pages 7–7. IEEE, 2008.

[332] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. Priter: A distributed framework for prioritized iterative computations. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 13:1–13:14, New York, NY, USA, 2011. ACM.

[333] Hong Zhu, Lijun Shan, Ian Bayley, and Richard Amphlett. Formal descriptive semantics of uml and its applications. *UML 2 Semantics and Applications*, page 95, 2009.