Information and Software Technology 152 (2022) 107039

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Undulate: A framework for data-driven software engineering enabling soft computing

Timo Asikainen [*], Tomi Männistö

University of Helsinki, Department of Computer Science, Finland

ARTICLE INFO

Keywords:
Soft computing
Multilevel modelling
Dimensional database
Continuous experimentation
Data-driven software engineering

ABSTRACT

Context. Especially web-facing software systems enable the collection of usage data at a massive scale. At the same time, the scale and scope of software processes have grown substantively. Automated tools are needed to increase the speed and quality of controlling software processes. The usage data has great potential as a driver for software processes. However, research still lacks constructs for collecting, refining and utilising usage data in controlling software processes.
Objective. The objective of this paper is to introduce a framework for data-driven software engineering. The Undulate framework covers generating, collecting and utilising usage data from software processes and business processes supported by the software produced. In addition, we define the concepts and process of extreme continuous experimentation as an exemplar of a software engineering process.
Method. We derive requirements for the framework from the research literature, with a focus on papers inspired by practical problems. In addition, we apply a multilevel modelling language to describe the concepts related to extreme continuous experimentation.
Results. We introduce the Undulate framework and give requirements and provide an overview of the processes of collecting usage data, augmenting it with additional dimensional data, aggregating the data along the dimensions and computing different metrics based on the data and other metrics.
Conclusions. The paper represents significant steps inspired by previous research and practical insight towards standardised processes for data-driven software engineering, enabling the application of soft computing and other methods based on artificial intelligence.

1. Introduction

Managing software processes has become increasingly complex. New features and versions of software-based services are introduced continuously. Consequently, the number of decisions concerning the evolution and content of such services to be made has grown. Ideally, the decisions should be made based on all the available data, including the data from both software and business processes. After all, any value created by the software is typically created in the business processes. Therefore, the data generated in the business processes can be assumed to contain more information about the value of software.

Given the large number of decisions that are needed and the required pace of decisions, there is a dire demand for various forms of automated decisions and decision support: In cases where the best course of action is clear based on data, we argue that a decision should be made and executed automatically. However, if the data is ambiguous, the relevant data should be presented to a human, and a ticket or other entity prompting a decision should be created. The approach is characteristically an exemplar of *soft computing*: depending on the availability and information content of data, different kinds of decisions can be made. When new data allowing more inferences arrives or new business rules are encoded, more decisions can be automated flexibly.

Data-driven software engineering has gained some research interest during the last couple of years [1–3]. In particular related to continuous experimentation, some models such as the RIGHT model [4] and the HYPEX model [5] have been suggested. However, these models address the experimentation process and strategic planning specifically but fall short of providing generic solutions for driving software processes with data. Similarly, many authors have applied the data-driven approach to requirements engineering, another key software engineering process [6, 7].

In recent years, machine learning and other artificial intelligence techniques have been studied intensively. A typical pattern in artificial intelligence research is to find the best algorithms and models, often based on neural networks, to well-defined problems with data sets

* Corresponding author.
E-mail addresses: timo.o.asikainen@helsinki.fi (T. Asikainen), tomi.mannisto@helsinki.fi (T. Männistö).

https://doi.org/10.1016/j.infsof.2022.107039
Received 30 November 2021; Received in revised form 26 May 2022; Accepted 4 August 2022
Available online 8 August 2022

in a fixed format. Often there are benchmark data sets available to solve the problem. However, this setting is not as such applicable to software processes nor the business processes powered by software: there is too much variety in those processes, primarily due to the fact that the business process can be almost any conceivable process. This condition severely hinders the straightforward application of artificial intelligence to software engineering and driving processes with data.

To better support the application of soft computing and other artificial intelligence methods in software engineering, we introduce the UNDULATE framework. The framework covers (1) the software processes and their control interfaces, (2) business processes powered by the software produced in the software engineering processing, (3) collecting and processing data from the above-mentioned processes, resulting in dimensional data, (4) applying computational methods to the dimensional data, thus enabling the control of software processes and generating stimuli for making decisions.

The framework is extensible: a software process may have a control interface (or many of them) or not, and its data may be captured or not. Similarly, the data from the business processes may be available or not (or something in between). Furthermore, new computational methods can be added as needed. Also, the dimensional database can be augmented with new metrics that may be defined in terms of other metrics or (augmented) raw data.

The remainder of this paper is structured as follows. Next, in Section 2, we will discuss previous work, followed by a description of the research method applied in this paper (Section 3). After that, we provide an overview of NIVEL^2 (Section 4), an advanced multilevel modelling language based on our earlier work that will be employed later in the paper. After that, we proceed to discuss the UNDULATE framework, starting with an overview of the related processes in Section 5. We proceed by giving a declarative description of experimentation concepts in NIVEL^2 and describing the experimentation process in the context of UNDULATE in Section 6. The part concerning UNDULATE is concluded by a discussion of soft computing and decision support enabled by the data. Discussion and comparison with previous work follow in Section 7. Conclusions and an outlook for further work round up the paper in Section 8.

2. Previous work

2.1. Software processes as a mapping from data to data

Table 1 contains a summary of various software processes from the data processing point of view. The table contains our abstraction of a number of software engineering processes as they are commonly defined in textbooks, see e.g. [8]. The table illustrates that even though two software processes can have independent traditions and identities, they may resemble each other from the data processing point of view: processes are controlled using some (meta) data and produce some form of data as output. The resemblance suggests that the same theoretical and pragmatic tools can be used to manage the data produced by software processes, and by the same token, by business processes. Also, as the processes are controlled by data, the similarities also enable driving the processes with data.

It is worth noting that although software testing has been largely based on tests that either pass/fail, information will be lost if the test outcome is reduced to this dichotomy. For example, in mechanical engineering, a prominent example of the application of artificial intelligence is related to maintenance: the failure of a device is often preceded by signals such as extra vibrations, different use of power or similar. In a similar vein, an increasing trend in the run time of an operation may imply that failures are about to occur, e.g. due to an impending timeout, and corrective action is needed.

2.2. Data-driven development

Data-driven development has been defined as the ability of a company to acquire, process, and leverage data in order to create efficiencies, iterate and develop new products, and navigate the competitive landscape [9]. To be useful, the ability should naturally be accompanied by action.

The setting resembles various forms of process automation in traditional engineering domains, such as chemical processing and power generation. In such settings, a human operator is often too slow to control the process efficiently; hence, automation is needed. Also, efficiency requires that the control is centralised to a control room instead of operators staffing various control panels distributed over the facility.[1] In many cases, the data generated within the process itself is enough to make decisions about continuing the process. This is the case in, e.g. a customary pipeline. On the other hand, all the available data should be leveraged when making critical decisions. Also, the data can be leveraged in making daily decisions once it has been made easily accessible to all processes.

2.3. Continuous software engineering and experimentation

During recent years, various continuous practices have become a significant trend both in the practice of and research on software engineering [10]. Simultaneously, *experimentation* has gained popularity as a way of acquiring knowledge from real users [11,12]. The continuous principles have been applied to experimentation as well, leading to *continuous experimentation* [3,13,14]. Continuous experimentation has been studied from various points of view, including conceptual [3,11,15], adoption [16], deployment [15] and metrics [17,18].

3. Research method

In this paper, we apply the constructive research method. In more detail, we generalise from previous research on software processes, data-driven approaches and automating software processes to come up with a generally applicable framework for data-driven software engineering.

Although empirical data is essential for keeping software engineering research aligned with the software engineering practice in the industry, we believe that constructive research is likewise necessary. Constructive research enables academia to introduce and elaborate on new ideas that would be too risky to study in the industry. It may also be the case that the most innovative ideas are only published as research or otherwise openly discussed only several years after their conception.

We base our understanding of the relevant fields, i.e. software processes, data-driven approaches and automating software processes, on previous research literature. We have collected no empirical data as a part of the research being reported. Also, for the research literature, we apply the *classical literature review* approach, with the aim to focus on the most representative papers in each field. In addition, we run a number of literature searches on relevant keywords in order to identify additional relevant papers.

The fact that we do not restrict ourselves to specific application domains or software processes implies that the kind of data we consider in the UNDULATE framework is likewise highly general. This, in turn, implies that the kind of data or even the situations in which decisions need to be made are unknown. Consequently, it is not possible to elaborate on specific computing methods.

We use experimentation as an example of a software process that may benefit from data and be driven by it. Several reasons motivate this choice. First, experimentation has not yet been thoroughly studied

[1] https://en.wikipedia.org/wiki/Process_control.

Table 1
Comparison of the inputs and outputs of different software processes.

Process	Input	Environment	Output	Utilisation
Experimentation	Minimum viable feature or product	Production	Data on selected metrics	Feature backlog (further development), Decide whether to deploy in production
Testing	Test cases (with variable degree of specification on the results)	Test (unit, integration)	(test case × execution) $\mapsto$ {pass, fail}	Alerts, tickets for bug fixes
Feature roll-out	Schedule of features to be rolled out	Production	Control metrics conferred with hypothesis	Perform rollouts in a systematic manner, roll back if necessary
Monitoring	Production	Metrics on system health etc.	React and repair	
Simulation	Agents (with specified behaviour), software under study	Production, Offline copy of product	Execution traces, user graphs	Comments on code changes, traces, etc.
Business processes	Software	Production	Usage data, User behaviour	Analytics, tactical and strategic decision making

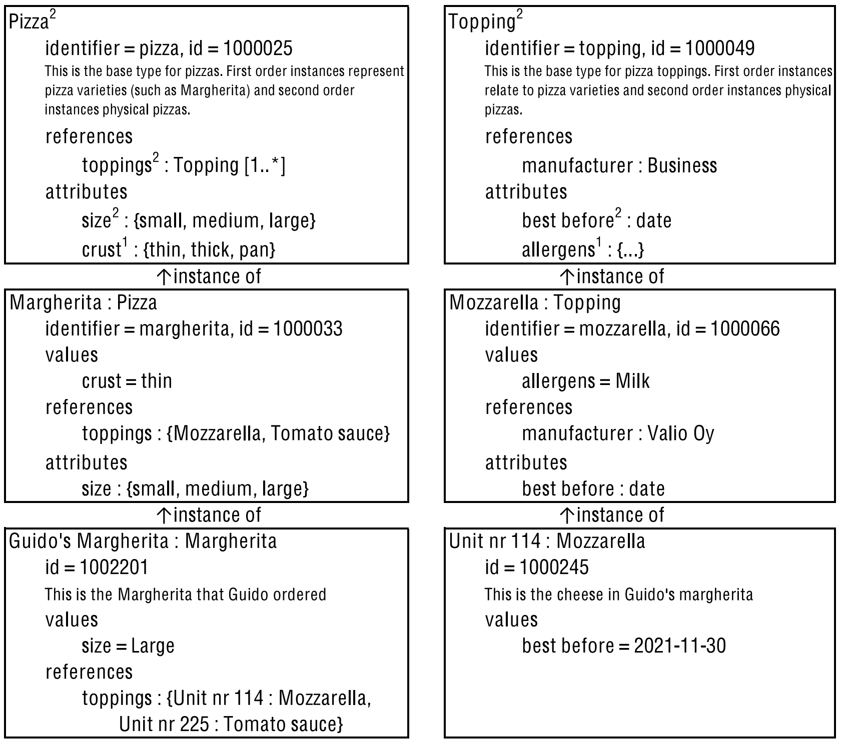
Pizza²
identifier = pizza, id = 1000025
This is the base type for pizzas. First order instances represent pizza varieties (such as Margherita) and second order instances physical pizzas.
references
toppings² : Topping [1..*]
attributes
size² : {small, medium, large}
crust¹ : {thin, thick, pan}
Topping²
identifier = topping, id = 1000049
This is the base type for pizza toppings. First order instances relate to pizza varieties and second order instances physical pizzas.
references
manufacturer : Business
attributes
best before² : date
allergens¹ : {...}
instance of
Margherita : Pizza
identifier = margherita, id = 1000033
values
crust = thin
references
toppings : {Mozzarella, Tomato sauce}
attributes
size : {small, medium, large}
Mozzarella : Topping
identifier = mozzarella, id = 1000066
values
allergens = Milk
references
manufacturer : Valio Oy
attributes
best before : date
instance of
Guido's Margherita : Margherita
id = 1002201
This is the Margherita that Guido ordered
values
size = Large
references
toppings : {Unit nr 114 : Mozzarella, Unit nr 225 : Tomato sauce}
Unit nr 114 : Mozzarella
id = 1000245
This is the cheese in Guido's margherita
values
best before = 2021-11-30

Fig. 1. An example of Nivel^2 modelling concepts.

in research. Instead, there is room for new conceptions of the process and data involved in it. Second, experimentation is essentially linked to data: experimentation is about obtaining and analysing data from two or more variants. Third, the data is typically generated in a production environment, i.e. outside the context of the software processes themselves, which makes the set-up interesting.

The research questions we set out to answer are as follows:

RQ1 What constituents are necessary to support data-driven software processes?
RQ2 What concepts and processes are required for extreme continuous experimentation for web-facing software services?
RQ3 How should the data generated by the software and business processes be processed and analysed to support software development?
RQ4 How can soft computing be used to support extreme continuous experimentation?

4. Nivel^2 — a short introduction

We use a variant/extension of a metamodelling language Nivel that we have created in our previous work [19]. The idea that a model may span an arbitrary number of levels spun by an *instance of relationship* is not new; see, e.g. [20], where the authors argue that such modelling concepts can be used to reduce the accidental complexity due to restricting to two levels in domain models. Nivel is based on a core set of conceptual modelling concepts: class, generalisation, instantiation, attribute, value and association. Nivel can be used to express models spanning an arbitrary number of levels defined by the *instance of* relationship: this is in contrast with conventional conceptual modelling methods, such as object-oriented modelling or entity-relationship modelling, in which there are only two levels available: one for types (classes and tables, respectively) and instances (objects and rows/tuples, respectively). However, in Nivel^2 an entity[2] can be both an instance of another entity and a type of an entity at the same time.

The version we use is named Nivel^2 and remains still unpublished. The most important aspect in which Nivel^2 is different from Nivel is that instead of associations, Nivel^2 uses *references* for representing

[2] In Nivel, the term *class* was used instead of *entity* used in Nivel^2: the term *class* implies the role of being a classifier or other entities, which is undesired as an entity need not be a type of another entity.

relationships between entities. This is justified by the predominance of references in modern knowledge representation. E.g. the JSON data format/language uses a form of references in that an object can include another object as a named value. The use of references will be discussed extensively below.

4.1. Concepts

Nivel^2 modelling concepts are illustrated in Fig. 1, such as the concepts of pizza and its toppings at various levels of abstraction. At the top, the entities Pizza and Topping are defined. The first order instances of Pizza represent pizza recipes or varieties, such as Margherita shown in the figure. Further, a second-order instance of Pizza represents physical pizzas, each an instance of a first-order instance, e.g. the Guido's Margherita. On the other hand, first-order instances of Topping are toppings appearing in pizza recipes, such as mozzarella or tomato sauce, whereas second-order instances of toppings are physical ingredients in pizzas.

Entities can be characterised using *attributes* and their manifestations, *values*. An attribute has the usual characteristics of *identifier*, *name*, and *description* and a value type, and in addition a *potency*, a positive integer with the semantics as follows: an instance of an entity containing an attribute of potency p has an attribute of potency $p-1$, for $p > 1$, and may have *values* with the identifier, name and type defined by the attribute for $p = 1$. As an example, Pizza defines an attribute size of potency = 2, which is manifested as values in second order instances, e.g. physical pizzas: size is a choice that a customer makes when ordering a pizza. On the other hand, the attribute crust represents the type of crust in a pizza recipe. As an example, Margherita has crust type thin.

The relationship between pizzas and toppings is represented by Pizza having a *reference* to Topping. The reference has *potency* = 2: this implies that the first and second-order instances of Pizza may have a reference to one or more toppings. In the example of Fig. 1, the reference *toppings* of Margherita has Mozzarella and Tomato sauce as its targets. More generally, each reference target in an instance of the source must be an instance of one of the targets of the source.

4.2. Implementation

We have implemented Nivel^2 as follows. The data is stored in a relational database (Microsoft SQL Server Azure) using tables for each basic concept (entity, instance of, reference, reference target, attribute, value, generalisation). A stored procedure can be used to query the data related to an entity. The database can be accessed using an API component *datapi* implemented in Python.[3] A generic interface component, implemented in React,[4] can be used to edit and view Nivel^2 entities.

5. Overview of the Undulate framework

In this section, we provide an overview of the Undulate framework supporting data-driven software engineering. Next, we will describe the concepts and process of the experimentation process, a stereotypical example of a software process founded on data that would greatly benefit from the application of soft computing techniques. Thereafter, we outline how the data produced in experiments and other executions of software can be refined and made accessible to the experimentation and other software processes.

As far as information systems, supported at least partially by software, implement a significant part of a company's business processes, the framework can also be applied to the business processes themselves and support the transition towards a data-driven business.

[3] https://www.python.org/.

[4] https://reactjs.org/.

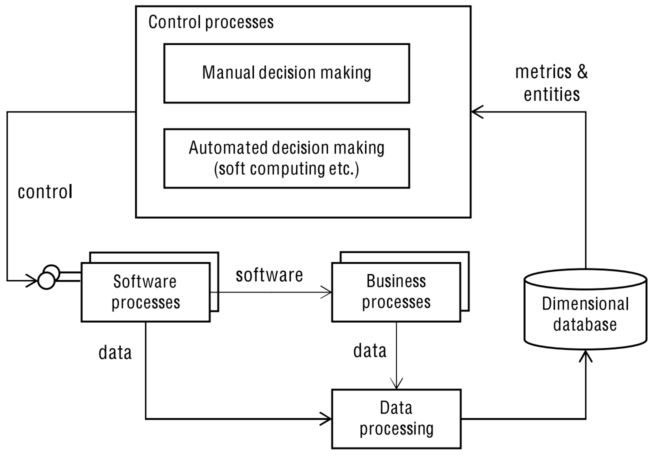

Fig. 2. The main processes underlying Undulate. **Legend** (entity — graphical symbol): Process — box, arrow — data flow, cylinder — database, grouping of processes — nested box and overlapping boxes.

Fig. 2 illustrates the general processes underlying the Undulate framework. A more detailed view of data processing can be found in Fig. 3. As can readily be seen from the figure, the processes linked by data flows form a cycle: this cycle demonstrates the fact that data drives the processes. In the following subsections, we will discuss various parts of the processes.

5.1. Data generation

Let us start from *software (engineering) processes* and the information they produced. First and foremost, software processes produce *software* that is later run as a part of *business processes*. Second, software processes themselves produce data, such as test reports, ticket data, activity data from version control systems etc.

It is assumed that the software produced by the software (engineering) processes is used in one or more business processes: as an example, the software could be used to implement a web-facing service, such as an online store running in a browser or a mobile application, and its backend running on a server and connected to various databases. Depending on to what extent the software is *instrumented*, details of user interactions with the software as well as its inner working can be collected and sent for further processing and analysis.

While the data produced by the software processes themselves is undoubtedly valuable for many purposes, *usage data* from the business processes should, in general, better reflect the behaviour of the users of the software and, consequently, the value that the software is producing. For example, in an online store, the software should enable a smooth payment process. However, transaction data from the online store could reveal that users have difficulties completing their purchases.

In the software engineering context, particularly related to online experiments, data is often assumed to be sent one record at a time using, e.g. an HTTP API endpoint or small batches. This is in contrast with traditional data warehouses, in which approaches such as ETL (extract, transform, load) based on files and scheduled (e.g. daily) updates have been the predominating approach. If the data is received with significant delay, it cannot be used for real-time analytics or process control. On the other hand, once the data has been received and appropriately incorporated into the database, the majority of analytics should remain the same, notwithstanding how the data was received.

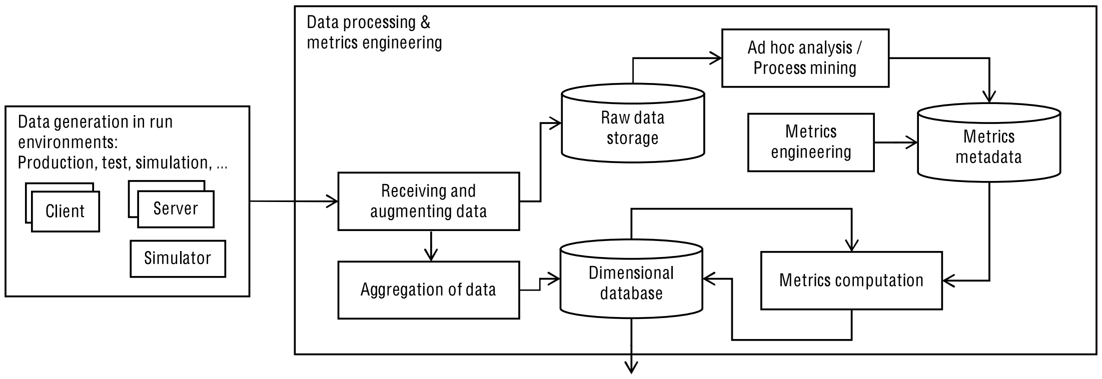
Data processing & metrics engineering
Ad hoc analysis / Process mining
Raw data storage
Metrics engineering
Metrics metadata
Data generation in run environments: Production, test, simulation, ...
Client
Server
Simulator
Receiving and augmenting data
Aggregation of data
Dimensional database
Metrics computation

Fig. 3. Data processing in Undulate. **Legend** (entity — graphical symbol): Process — box, arrow — data flow, cylinder — database, grouping of processes — nested box and overlapping boxes.

5.2. Data processing and dimensional database

Both the data produced by software processes and business processes enter the *dimensional database* through the *data processing* process. Data processing is described in more detail in Fig. 3.

The first stage of data processing is *receiving and augmenting the data*, also called *preprocessing*. Data may be received by an API service or similar, either one record at a time or, especially where a data connection is not continuously available, in batches. Upon receiving the data, the data may be augmented with additional metadata: for example, a user ID may be used to supplement the record with other dimensional data, such as geographical location, age, gender or similar information that can legally and technically be used in further analysis.

After preprocessing, the data is forked into two distinct storages: *raw data storage* and *dimensional database*. As the name suggests, the raw data storage contains the data in the form in which it was received, with no loss of information. The purpose of storing raw data is that it enables *ad hoc analysis*, i.e. computing aspects of data that were not preconceived when implementing the software. As an example, social media platforms enable different forms of malicious behaviour, some of which may not have been considered by the designers of the platform at the design time. However, when such behaviour emerges, it is useful to be able to go back to the data can analyse it with respect to the behaviour.

Privacy As the raw data has not been aggregated or otherwise anonymised, it is possible or even likely, depending on the application, that the raw data is *personal data*: names, user IDs, IP addresses, email addresses etc. but also free text may be individually, or in combination with other data, sufficient to link the data to a person with certainty or significant likelihood. Therefore, special legislation concerning personal data may apply to the raw data. The general data protection regulation (GDPR) of the European Union is a prominent example of such legislation. In essence, such legislation may, in effect, prevent the processing of data beyond prescribed purposes and tasks. Further, safeguards are needed to keep track of who has accessed the data etc. Therefore, raw data or other data that has not been anonymised cannot be freely used for explanatory analyses or making decisions in innovative ways.

Privacy legislation, such as GDPR, usually makes no general exemption to its provisions concerning data analysis. In other words, the restrictions on processing data apply to analysis as well, and so do the obligations of providing information on the processing of the data subjects. This makes developing analytical methods more difficult. To alleviate this situation, simulation can be used to produce sample data that can be used as a basis for implementing various forms of analysis. While simulated data may be different from real user data in many respects, it is still likely to be syntactically and structurally, and it can be used to validate that the computations that are to be applied to real data later work as intended. We will discuss simulating user behaviour below; see Section 6.2.2.

The main body of analysis and further computations are ideally based on a *dimensional database*. The dimensional database resembles functionally a *data warehouse* or a *data mart*[5]: The data is stored as *facts* in *fact tables*. Fact tables are linked to *dimensions*, such as date or geographic location, and facts correspondingly with *dimension values*. Each dimension includes only a manageable number of values, also termed *levels*. For example, instead of storing timestamp with the millisecond accuracy or higher, the dimension would only include the date or have the precision of at most seconds. The purpose of reducing the number of possible levels is to be able to *aggregate* data based on dimensions.

Aggregating data serves several purposes. First, aggregation may, if the characteristics of data are adequately considered and implemented, enables the anonymisation of data, making it possible to utilise the data in the application and freely analyse it.

Second, aggregating data also helps reduce the size of the data, resulting in faster analysis and lower storage requirements.

Third, aggregation enables joining data from multiple fact tables based on dimensions for analysis and other purposes. For example, it is possible to study the interrelation between application failures and server load levels by joining the two fact tables according to the time and service instance dimensions. Note that the more fine-grained level of dimension is used, the more direct the link between the variables should be; on the other hand, there will be fewer units available for analysis, and random factors will play a larger role. Taken to an extreme, there may be no matching values available at all if the level of granularity is too high.

In addition to joining the fact tables along the matching dimensions, i.e. dimensions that are the same, it may also be relevant to study lagged time dimensions which help to answer questions related to delayed behaviour and reason about *leading* and *lagging* indicators. As an example, a campaign, other event or an update may have long term effects which can be analysed by means of time series analysis of the related time series.

Also, retention is defined as the proportion of customers a business is able to keep over a period of time and can be defined as the number of customers left from the initial set at the end of the period, i.e. the start of the next period.

[5] Although dimensional databases have been applied in practice for decades [see e.g. 21], there is somewhat surprisingly only a little research on the topic.

5.3. Control processes

The loop in Fig. 2 is closed when data from the dimensional database is used in *control processes* to drive the software processes. For example, the data generated in business and software processes can be used to decide which tests to run, which features to develop further or roll out or how much server capacity should be made available in the future. In general, control processes can be either manual or automated in part or entirely. We will discuss continuous experimentation as an example of a software engineering process that is essentially data-driven and may benefit from automation in Section 6.

5.4. Metrics engineering

In addition to the processing of data itself, we believe that it is crucial to pay effort to the definition and content of metrics themselves. We term this activity *metrics engineering* and the related, organised storage of data pertaining to *metrics database*, see Fig. 3. Recognising metrics engineering as an activity of its own is based on the observation that significant statistical and application-specific expertise enters the way how metrics are defined and computed. We believe that identifying this as an activity of its own will improve the quality of the collected data and, maybe more importantly, the quality of the processed data that is eventually used to drive software engineering and other processes. Also, the metrics database supports reuse at the level of metrics and may also serve as the platform for publishing information related to metrics. The metrics database should contain data similar to a quality description in statistics, including, e.g. how often the data is updated, possible sources of error, if and how corrections are made etc.

It may also be reasonable to compute known high-level metrics already in the client or server depending on the availability of computing and other resources, the possibility of changes in the definition of high-level events etc.

Finally, it is also possible to define new metrics based on metrics. As an example, the standard Z score related to the null hypothesis $\mu_0 = \mu$ is $Z = \frac{\hat{X}-\mu_0}{s}$, where $\hat{X}$ is the sample mean and s is the standard deviation of a sample.

5.5. Adopting data-driven software engineering

For a framework such as Undulate to be practically useful, it is of paramount importance that the framework can be implemented in a company within a reasonable time frame and preferably in an incremental manner. This is since it is typically economically not feasible to commit to multi-year projects that provide no early paybacks. Also, the framework should not require extensive changes to a large number of processes in order to be useful. In this section, we discuss how the design Undulate addresses these issues.

The adoption of the Undulate framework is illustrated in Fig. 4. In the figure, the curves in the time-level of automation space represent different software processes controlled using data.

The dimensional database and the related data processing are the key architectural components (not shown in Fig. 4) that must be in place for the framework to work. In addition, driving processes using data requires two components: First, the data used to drive a process must be made available in the dimensional database. This is illustrated using dependency arrows from the processes to the parts of the dimensional database (cylindric shapes).

Second, automation is only possible if there is a machine-operable interface to the process, e.g. a REST API or a command-line interface, which can typically, with ease, be turned into an API. The diamond symbols represent the control mechanisms enabling automated control over the processes.

The illustration of the adoption process is, necessarily, an idealisation. The degree of automation does not need to grow smoothly nor approach 100% as time passes. Also, new features and subsystems are

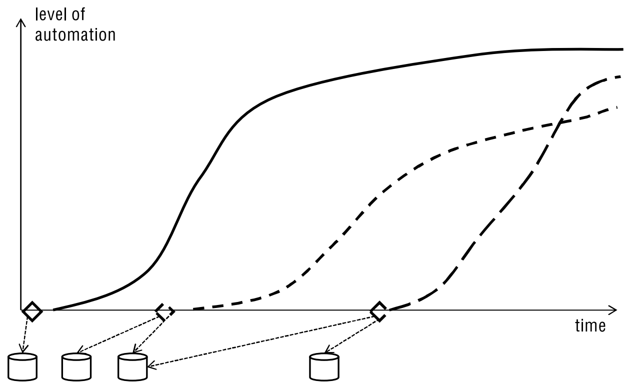

Fig. 4. An example of the transition towards data-driven software engineering. In the course of time, new data sources and control mechanisms can be added incrementally instead of committing to fully data-driven processes at once. The level of automation (proportion of decisions made automatically) increases in each process as the data becomes better understood through learning from past data and decisions. **Legend** (entity — graphical symbol): curve — the level of automation of a software engineering process as a function of time; diamond — the introduction of a control mechanism (related to a process); cylinder — data source; dashed arrow — dependency of a process on a data source.

introduced, and the proportion of manual decisions may also decrease. It should also be noted that automation should not be a value as such but a means towards other goals, e.g. making decisions quicker and enabling the expert staff to focus on decisions with high risks or otherwise of importance.

6. Experimentation

In this section, we discuss the software engineering process *experimentation*. In more detail, we define the main concept related to experiments using Nivel^2. In addition, we will outline the experimentation process.

6.1. Concepts

Fig. 5 illustrates the Nivel^2 entities used to represent concepts related to experiments. We will use a sans serif typeface when referring to Nivel^2 concepts.

The main concept is naturally Experiment. The first order instances of Experiment represent experiments in the conventional sense: such an instance defines a number of hypotheses and test groups, where each group is treated with different software. Both hypotheses and groups are represented by Nivel^2 entities, Group and Hypothesis. In addition, an experiment instance defines a target duration, answering the question of how long the experiment should run.

An instance of Group defines aspects related to an experiment. Most importantly, an instance of Group defines an implementation (represented by the references implementation) used to produce the service for the users in that group. The implementation is represented as a reference to one or more containers: an instance of the group refers to one or more instances of Container, corresponding to source code or a container image, e.g. Docker[6] image: here, the first order instances of Container have a different name in common language than the second-order instances, i.e. containers created from an image. A group also defines the target size of its instances, i.e. how many users the group should have when the experiment is run.

Note that an instance of Experiment may be dynamic in that the active groups of the experiment may change over time as new groups are added. Old ones are completed, i.e. the timespan scheduled for

[6] https://www.docker.com/.

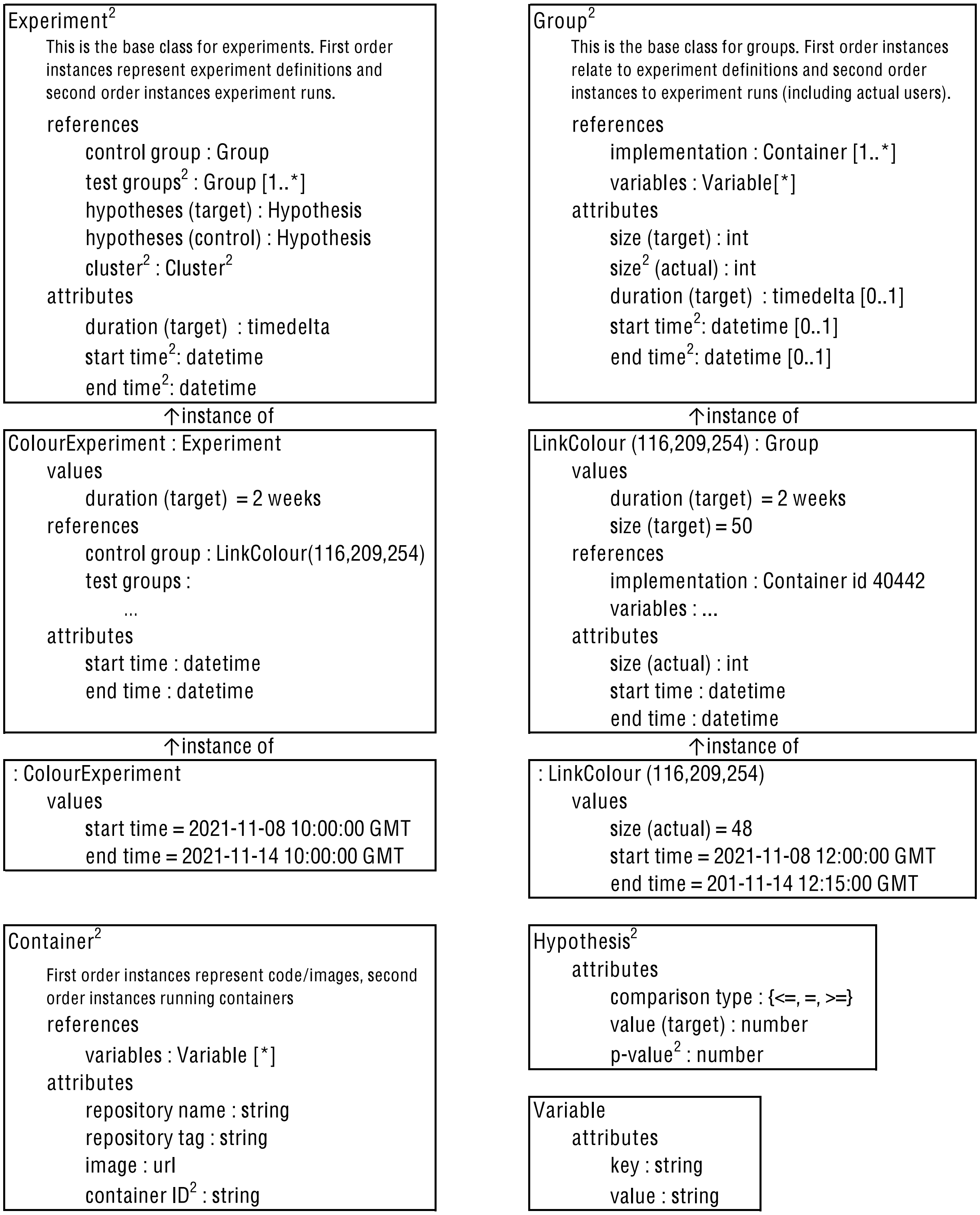
Experiment^2
This is the base class for experiments. First order instances represent experiment definitions and second order instances experiment runs.
references
control group : Group
test groups^2 : Group [1..*]
hypotheses (target) : Hypothesis
hypotheses (control) : Hypothesis
cluster^2 : Cluster^2
attributes
duration (target) : timedelta
start time^2: datetime
end time^2: datetime
instance of
ColourExperiment : Experiment
values
duration (target) = 2 weeks
references
control group : LinkColour(116,209,254)
test groups :
...
attributes
start time : datetime
end time : datetime
instance of
: ColourExperiment
values
start time = 2021-11-08 10:00:00 GMT
end time = 2021-11-14 10:00:00 GMT
Group^2
This is the base class for groups. First order instances relate to experiment definitions and second order instances to experiment runs (including actual users).
references
implementation : Container [1..*]
variables : Variable[*]
attributes
size (target) : int
size^2 (actual) : int
duration (target) : timedelta [0..1]
start time^2: datetime [0..1]
end time^2: datetime [0..1]
instance of
LinkColour (116,209,254) : Group
values
duration (target) = 2 weeks
size (target) = 50
references
implementation : Container id 40442
variables : ...
attributes
size (actual) : int
start time : datetime
end time : datetime
instance of
: LinkColour (116,209,254)
values
size (actual) = 48
start time = 2021-11-08 12:00:00 GMT
end time = 201-11-14 12:15:00 GMT
Container^2
First order instances represent code/images, second order instances running containers
references
variables : Variable [*]
attributes
repository name : string
repository tag : string
image : url
container ID^2 : string
Hypothesis^2
attributes
comparison type : {<=, =, >=}
value (target) : number
p-value^2 : number
Variable
attributes
key : string
value : string

Fig. 5. Concepts related to experiments and sample instances represented in Nivel^2.

the group has ended, or the group has been otherwise terminated, e.g. due to sufficient level of statistical significance has been reached. The changes in groups can be done manually, or they may be done automatically based on the *search strategy*, defined by a referenced Nivel^2 entity. The role and semantics of search strategy are discussed in more detail in the following subsection concerning the experimentation process. Due to its dynamic character, an experiment can also be termed an *experiment programme*, although *experiment* is preferred here in most cases for brevity.

A Hypothesis defines a condition related to a metric or a statistic computed based on the experimentation data. For example, the hypotheses could state that a target variable in an experiment, say click-through rate, should have a value that is statistically significantly greater than the baseline observed in production.

An experiment makes a distinction between *target* and *control hypothesis*, represented by references with respective names. The idea behind the distinction is that while an improvement, such as an increase in click-through rate, is expected in the primary hypotheses, there is simultaneously expected to be no change in the control hypotheses. For example, one would expect that a change in the user interface does not cause the number of application crashes (per page load) to increase. To automate the decisions based on experiment results, a reasonable strategy could be to introduce a sufficient number of control hypotheses covering all aspects of the correct behaviour of service and then check these hypotheses automatically, enabling the experts to concentrate on the primary hypotheses.

A second-order instance of Experiment represents a run of an experiment. The content of the experiment is defined naturally by its type, the first order Experiment instance. The second-order instance has values tied to the attributes and references defined at the top level. For example, the second-order instance has a start and end time, and the groups of the experiment have actual sizes and respective start and end times. Also, the hypotheses related to the experiment have values that enable reasoning on them.

6.2. Experimentation process

The experimentation process is illustrated using a flow chart in Fig. 6. The experimentation process begins with the experiment design that will result in a Nivel[2] experiment entity along with the entities it refers to. Designing an experiment is typically an iterative and cross-disciplinary process, the details of which will depend on the characteristics of the experiment and organisation and are outside the scope of this paper. For the software experimentation process in general, see, e.g. [22]. In addition to manual experiment design, previously acquired data can be used to generate new experiments also even automatically or at least extend existing experiment programmes with new test groups.

Using the design of the experiment as input, the software that will be used to implement the experimental services must be prepared. This part of the process may include different activities based on the situation. Suppose the features being experimented on have already been *implemented* and *instrumented* to generate the data needed to compute the outcomes of the experiment, i.e. resolve the hypotheses. In that case, the required source code can be automatically *generated*. If, on the other hand, features have not yet been implemented or instrumented to produce the data, this must be done using the software processes of the company running the experiment.

Overall, instrumentation plays a significant role in experimentation. Depending on the application domain and devices used, it may not be feasible to collect all data related to all features but rather seek a balance between performance, bandwidth usage etc. and the availability of data. In any case, the instrumentation should include the information about the experiment group it relates to: this will enable linking the usage data to the experiment in a reliable way.

Once the source code is ready, it can be *built* using the processes and tools that are generally used in the company. The build pipeline may, for example, include automated tests at different stages of the pipeline. In Fig. 6, it is assumed that Docker images are built, but the process itself does not commit to a particular technology. However, some assumptions are made about *deploying* the images (or other encapsulated units of software). First, it is assumed that such images can be deployed independently of each other. Second, it is assumed that there are available mechanisms for *routing* some requests to certain implementations of service at runtime dynamically based on, e.g. cookies. An example of such mechanisms is Kubernetes[7] (for managing containerised workloads) and Istio[8] (for routing the traffic to the correct implementation of a service based on experiment group membership, encoded in a cookie or similar part of a request). The above-mentioned technologies are mainly practical tools, but they have been discussed in the research [see e.g. 23–25].

Once the containers providing the experimental software are running, and the routings have been updated, the cluster is *ready for executing the experiment* and the experiment is ready to run. The users in experiment groups are routed to the experimental version of the software, and the usage data is sent for processing as described in Section 5.2.

6.2.1. User registry and management

In parallel with preparing the experimental software, the users participating in the experiment need to be selected and allocated to experiment groups. In this paper, we assume for simplicity that users need to register and log in to use the software under experimentation. In practice, this is not an assumption that would be true in all circumstances. Further, identifying users that have not registered may be difficult or even impossible between sessions. Users are allocated to groups randomly. The number of users allocated to each group should be larger than the desired size of the group: users inevitably drop out of the experiment during its course, and some allocated users may even never actually enter the experiment.

It may also be the case that a user may not be allocated to an experiment. Various reasons for this may exist: for instance, the user may have participated in an experiment previously and needs, therefore, to be excluded from further experiments until a predefined time, e.g. two weeks, has passed. In addition, judicial reasons may prevent users from participating in an experiment. The experiment or the group itself may have specific requirements for the users participating in an experiment. For example, the experiment may be restricted to a particular geographical area or a previously identified cluster of users. Overall, a large variety of different considerations must be taken into account and combined with various forms of data related to the users. The logical site for such computation is called the *user registry*.

Once the cluster is ready for execution and the users have been allocated, the experiment may be started. The start of an experiment may be specified in the second-order experiment instance (*start date* value). At that time, users allocated to groups should receive their group ID when authenticating to the service along with other data. This group ID is subsequently used to route the user requests to the software related to that group and sent back with usage data for processing.

6.2.2. Simulated users

Usage data is essential in any experiment. As we cannot rely on real user data in our work, we have implemented a simulation system to overcome the lack of actual user data.

In a simulation, each *user* belongs to a *group*. A group defines a set of possible *states* in which a user can be at a given time. In addition to its state, each user has a set of *variables* that affect its behaviour. The simulation happens in *rounds*. On each round, an *action* is selected for the user. The action to be performed is selected randomly based on *action weight formulas* defined for the group of the user: each action weight formula is evaluated, resulting in a weight value, used subsequently in the draw. The action may involve invoking the service related to an experiment. Through the calls to the service that may be external to the simulation, the users can also interact with each other; otherwise, each user is encapsulated and not affected by other users.

In the next step of the round, the result of the selected action enters the *state transition formulas*, defined for each group and state. The formulas may also involve the variables of the users. Similarly as,

[7] https://kubernetes.io/.

[8] https://istio.io/.

when selecting the action, the new state is drawn based on the weight values from the formulas.

After choosing the new state, the variables are updated based on *variable formulas* defined likewise for the group. The variable formulas may involve the current values of the variables as well as the state and the result of the action.

Finally, a formula is used to compute the *delay* (in seconds) until the next round begins.

Each formula may also involve a random component drawn from a predefined distribution, such as the normal distribution. This enables stochastic simulations. Another source of stochasticity is that each variable may be defined as an initial distribution. Due to the differences in initial values of the variables, each user belonging to the same group may have a different type.

The group as defined above can be modelled in Nivel^2; the corresponding entity is called a *group definition*. Similarly, a *simulation* is defined in Nivel^2. A simulation is characterised by a *set of groups*, each to referring a group definition and a number of users to be created for the group. A simulation also defines a *time scale* that defines how quickly the simulation should be run in comparison to real-time; a simulation can also be run as quickly as possible, i.e. without scheduled delay between rounds of simulation.

The simulations described above have been implemented in a component called *simulient* in Python. The *simulient* system sends the simulation results to an API called *results_api* after each round. The results involve the user ID (running number), group ID, initial and final state and the action taken. In addition, the values of each variable are sent as details. The simulation runs for a scheduled duration of simulation time. It can also be stopped using an API endpoint on request.

6.2.3. Updating and stopping the experiment

While the experiment is running, the accumulating results, as received from the dimensional database, may be used for various tasks. Soft computing and other methods may be used for this task. As an example, if one or more of the control hypotheses show that some group is performing badly, alerts can be sent for human intervention or the group or entire experiment may be aborted altogether. Similarly, based on the results, new groups can be added. For example, in a search-based approach, if a group G corresponding to a certain point in an n dimensional parameter space performs significantly better than other groups currently running, the search may be extended to points closest to G (in a grid). If, on the other hand, no currently running group is better than the others and each group has been running for the scheduled time and is ready for analysis, a ticket may be added suggesting human intervention to the experiment.

6.2.4. Implementation

Checking the experiments. Periodically, the Nivel^2 database is queried for active experiments, i.e. experiments that are running. Each running experiment is checked, and groups may be terminated (moved under completed groups) or added based on the results so far, the attribute values of the experiment (e.g. the maximum number of groups running in parallel). A search strategy for linear search has been implemented as a Transact-SQL stored procedure.

If and when the experiment is run as a simulation instead of using real users, the corresponding *simulient* configuration is updated with the newly created groups so that data will be generated corresponding to the new groups as well.

Updating the cluster. Another process, likewise invoked periodically, goes through each combination of service and cluster to check which experiment groups should run in each cluster. Based on this information, JSON files containing Kubernetes specifications (deployment, destination rule, virtual service) as generated and applied to the respective clusters.

Simulated users. In parallel with the experiment, a *simulient* session is running. The *simulient* component checks the simulient configuration in Nivel^2 database using data API provided by *datapi* for new or completed groups and updates its internal configuration accordingly: users in the terminated groups are removed, and new users are added for the new groups.

Viewing the experiment results. We have implemented a web interface for viewing the experiment results. The interface is implemented using the Shiny[9] package of the R[10] programming language. The interface allows selecting a run of the experiment for further analysis. The groups and metrics related to the run are shown, and any subset can be selected. In the current implementation, the results are shown as curves, one for each combination of selected metric and group; other visualisation can be added with relative ease.

7. Discussion

7.1. Answering the research questions

In the following subsections, we will answer each of the research questions RQ1–RQ4, respectively.

What components are necessary to support data-driven software processes?

The question is answered by the construction of the Undulate framework in Section 5. Its key components are:

- Control interfaces of software processes. If automation is required, the interfaces should be programmable. Otherwise, manual interfaces will do as well. New interfaces can be added and old ones automated as new data-driven practices are introduced.
- Data collection from both software and business processes. Initially, it is not required to collect all data from all processes.
- The dimensional database, which is a key and mandatory component of the framework. A centralised database organised using the same dimensions, where applicable, enables joining data stemming from different processes, thereby vastly expanding the possibilities of utilising the data.
- Controlling processes based on the data is the very essence of data-driven software engineering. Similarly, as in the items above, not all decisions need to be based on quantitative data and or automated.

What are the concepts and processes required for extreme continuous experimentation for web-facing software services?

The concepts related to experimentation were discussed extensively in Section 6, see esp. Figs. 5 and 6. In addition to the basic concepts, some technical concepts may be necessary depending on the deployment technology used.

dee

How should the data generated by the software and business processes be processed and analysed to support software development?

Data processing as a part of the Undulate framework is discussed in Section 5.2. The storage of data is centralised, which allows joining data from different sources for various analytical tasks. The same goal is supported by the use of dimensions to characterise data. The use of dimensions, as well as metrics computed based on augmented raw data and other metrics, works towards the same goal.

Further, the Undulate framework is mainly based on using data that is anonymised by aggregating it to large enough units. This approach makes the data more useable from a privacy point of view. On the other hand, raw data is saved and made accessible on a more restrictive basis for ad hoc analysis, process mining etc.

[9] https://shiny.rstudio.com/.

[10] https://www.r-project.org/.

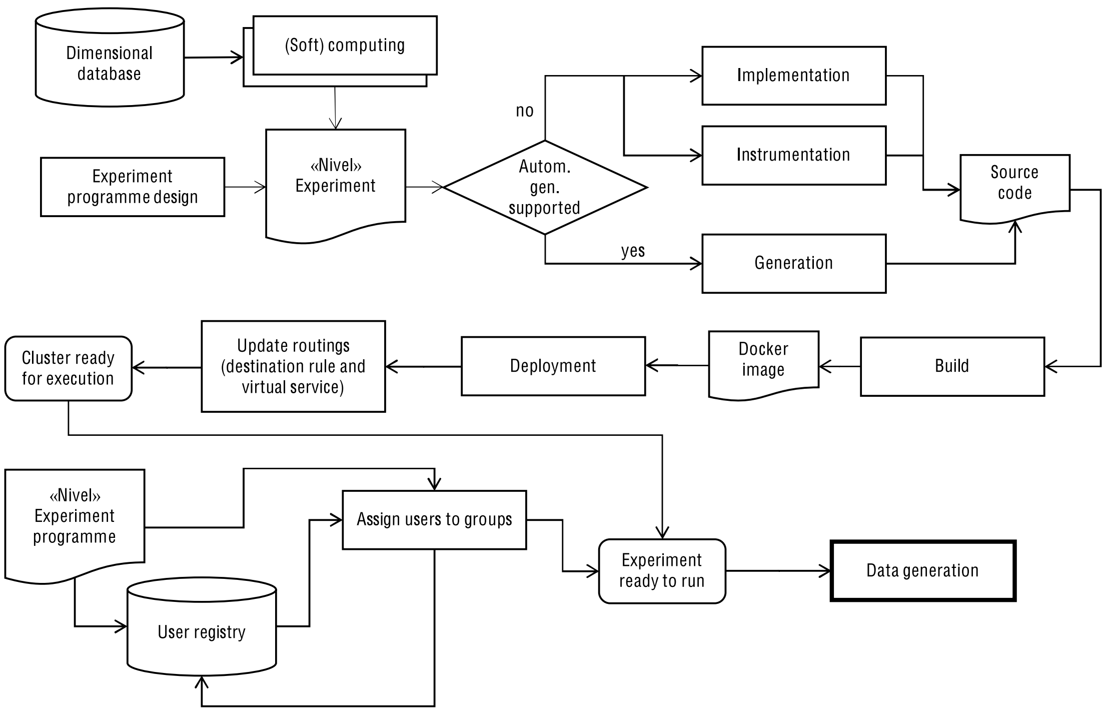

Fig. 6. Flow chart of the experimentation process.

How can soft computing be used to support extreme continuous experimentation?

The data made available in the dimensional database can be used as such as an input for soft computing methods. In addition, in Section 7.5, we will argue that aspects of softness are also present in various familiar software and statistical processes.

7.2. Experimentation

In the following subsections, we compare the work presented in Section 6 with previous work.

7.2.1. Knowledge representation

In this paper, we are using Nivel[2], a declarative multilevel modelling language for representing experimentation knowledge. This is in contrast with most previous approaches to experimentation: the predominant style of knowledge representation seems to be various configuration files in JSON or similar languages and even procedural knowledge in the form of computer programs written in general-purpose programming languages.

Although configuration files and programming languages have the benefit of familiarity to software professionals, we believe that applying declarative knowledge representation techniques provides a number of benefits. First, languages such as Nivel[2] provide tool support for creating, viewing and maintaining the knowledge base. Second, the languages provide schema-like support for creating entities and ensuring their consistency. Third, the tool support resembles standard form editing functionalities, which makes the knowledge more accessible to non-technical experts.

7.2.2. Process

The overall experimentation process described in this paper resembles the processes discussed in existing research [see e.g. 6,7]. This has also been the intent. However, unlike in previous research, we allow the experiment to take the form of an experiment program, consisting of multiple related stages in which groups may be added and completed groups removed. The changes in groups are based on the results from previous phases and are an example of extreme continuous experimentation driven by data.

7.2.3. Metrics

Metrics are artefacts, and like any other artefact, large amounts of both explicit and implicit or tacit knowledge are related to metrics. Traditionally, statistics and analytics has been an expert task with a limited amount of automation. When automation has been available, it has been supported by a thorough understanding of the accumulating data, both from the content and statistical and analytical point of view.

In previous research, metrics have been classified as OEC (Overall Evaluation Criteria) metrics, data-quality metrics, guardrail metrics and local feature and diagnostic metrics [17,18]. The OEC metrics correspond to target hypotheses in the *experiment* entity of Fig. 5. More generally, in the approach presented in this paper, the classification does not apply to the metrics themselves and not even the hypotheses, but rather the hypotheses in the context of an experiment. Further, the other metrics roughly correspond to control metrics. The extent to which changes are allowed to each kind of the remaining metrics varies. In our approach, this would be addressed by defining the corresponding hypotheses, e.g. setting the reference value.

7.3. Dimensional databases

In previous research, the general characteristics of storing experiment data have not been studied extensively. As the experiment data is typically stemming from the production system, experiment data is a subset of usage data generated as the software is used.

In the case of data-driven software engineering, the characteristics of data will depend on a large number of factors, including the software under study, the instrumentation in particular, how collected data is sent for analysis, possible with delay or omitted altogether; these conditions are relevant, especially in the context of mobile apps. Also, the users have control over the cookies that can be set, which is bound to affect the amount and quality of data that is available and will show up as omitted values for some users/session variables. In addition, the data is affected by the application itself, including possible errors.

The complexity of managing data is also emphasised by the emergence of specialised roles, such as data engineer, data scientist and business analyst [see e.g. 26].

Although dimensional databases seem well suited for many analytical tasks related to software engineering, not all kinds of data fit their scope. As an example, *clusterings*, or more generally, *embeddings* of users, products, items of news etc., would best be represented as attributes of the dimensional values rather than facts.

Although dimensional databases are not a topic of intense scientific work, they are practically still relevant. As an example, Google Analytics, a part of Google Marketing Platform[11] seems to be essentially a dimensional database, although not explicitly phrased as such. Google Analytics also provides an example of computations that are made based on the raw data and not the (aggregated) dimensional data: an *attribution* is a calculated assignment of a quantity (such as the value of a sale transaction expressed in currency) to one or more contributed elements, such as landing page or other pages visited before the sale was completed. The attribution is more likely to be useful if it is based on individual page views and sales transactions instead of aggregated data from even a short period. The attributions themselves can be aggregated.

The raw data store and analysis based on it illustrated in Fig. 3 provide the opportunity for trying out different advanced computations. Once completed, such computations can be incorporated into the data augmentation phase.

In addition to the abundant technical challenges related to collecting, processing and utilising data, also legal challenges are increasing. Most importantly, GDPR sets significant requirements on any controller or processor of personal data situated in the European Union or processing the data of citizens of the union. Although there is some initial research on the effects of GDPR on software development [see e.g. 27], the GDPR, along with other privacy concerns, remain largely unaddressed in research. The aggregating performed as a part of the data processing in Undulate is, in principle, an efficient form of anonymisation and therefore resolves the privacy concerns.

7.4. Simulation

As a part of validating the implementation of the experiment process in Section 6, we introduced the *simulient* component that generates a simulated database based on a configuration declaratively represented using Nivel[2]. The simulation shows that the experimental software is correctly built and deployed and that the routing to the software using cookies works as intended. Finally, the data thus generated is received by the API component *result_data_api* and available for analysis.

However, simulation has other purposes beyond the kind of validation described above. During recent years, *digital twins* of different physical entities, such as factories, have become the topic of intense discussions. The underlying idea or assumption is that the data describing the entity is more or less readily available for simulations, which enables running different scenarios. This, in turn, allows analysing of the effects of specific alternatives and different optimisations to be made.

Digital twins may also be useful in the digital world, as has been shown by Facebook [28–30]. The authors report the use of simulations for various purposes. The simulations may be run against the actual production platform, an offline copy of the platform (in which the responses to specific events have been recorded), and a model-based version of the platform in which data from production has been used to train a statistical model of the platform. Simulations provide a quick and low-cost way to study various aspects of the platform, such as the possibility of malicious behaviour. From the software engineering point of view, simulations also provide an interesting alternative for testing and experimentation. As argued in Section 2, software processes that are different at first sight may turn out to be similar in many respects, and these similarities can be exploited when implementing and automating these processes.

[11] https://marketingplatform.google.com/about/analytics/.

7.5. On the notion of softness

In this subsection, we discuss how softness in various forms is manifested in different software engineering and other applications.

CI/CD (continuous integration/continuous deployment) pipelines and other practices have become increasingly popular in software engineering. However, CI/CD pipelines lack a commonly agreed definition. However, the common understanding is that the pipelines automate a number of steps, each using as input the output from the previous stage or stages. There are few, if any, alternative paths: if the pipeline fails at any point, the execution stops and some kind of error is raised. Therefore, there is little softness as such in these pipelines. Therefore, the Undulate framework as such and in particular applied to continuous experimentation would significantly increase the potential for applying data-driven soft computing techniques in continuous software engineering processes.

Examples of data-based optimisations in software processes include:

- Different kinds of resources can already be elastic in clusters in that their capacity changes based on the usage data in real-time. Also, usage forecasts based on usage data can be used to alter the capacity before actual changes in usage occur
- Also, other forms of resource optimisation in a cloud environment can be implemented
- Relational databases use previous execution data to optimise queries as well

The above implies that the idea of leveraging usage data to drive software engineering processes is not new.

Aspects of softness can also be seen in *risk-based methods*. Examples of such methods include inspecting only certain passengers entering a country, tax returns or components in complex machinery. What is common to these scenarios is that exhaustive inspection of all the entities is not possible due to the costs to the inspector and subjects. The cases selected for inspection are based on various sources of data, for example, the personal data of a passenger or some model of how components age and break does, or which components would cause the largest damage should they fail. In addition, some cases may be selected for inspection randomly: this is important if the inspection is related to detecting fraud or similar malicious human behaviour.

On the other hand, in software engineering, especially software testing, the emphasis has been on achieving full coverage: all the cases or combinations of values should ideally be tested, and testing should cover the entire code base. This may be due to the fact that software can, in many cases, be tested with relatively little cost nowadays with the aid of test automation. However, in some cases, software testing may be costly. As an example, this may happen in the case of embedded software when specialised, high-cost equipment is needed for testing. That equipment may then become the bottleneck in the production process. To optimise the process, one would like to select the tests that have the highest potential to reveal broken functionality in the devices and even to predict future problems; tests that are not useful for this purpose could be omitted. The Undulate approach is well suited for such optimisation.

7.6. Transition towards data-driven software engineering

The Undulate framework introduced in this paper is unequivocally based on centralised data storage and control based on its data. In practical settings, implementing such data storage may be prohibitively expensive in time-consuming. Indeed, it has been reported that the application of artificial intelligence methods can be difficult to achieve due to various reasons, such as management scepticism, pressure to develop new features and challenges of combining data from multiple sources [26]. The dimensional database in Undulate should, however, help to resolve the challenge of combining data from multiple sources.

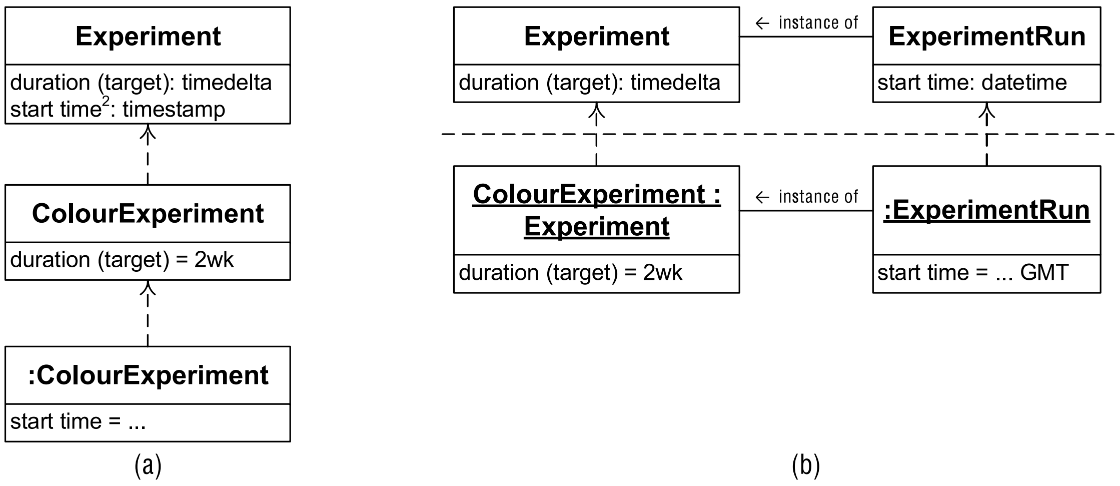

Fig. 7. Comparing the representation of the experiment concept using (a) Nivel² and (b) standard UML class diagrams.

The acceptability may be improved when the same infrastructure is created with the aim of supporting both software engineering and business processes, all the way to the strategic level. Also, the fact that not all data sources need to be included at once should reduce the initial investment required. The same applies to the fact that not all software processes need to be included in the initial applications, as well as the fact that the target need not be full automation at once, if at all. Setting up such an infrastructure is still likely to require high-level management commitment from both the business and technical sides.

A possibly overlooked aspect is related to making the data useable; it may not be obvious what is available. Even when considering the software processes alone, not all stakeholders are likely to find and make use of the data directly using technical interfaces. Also, especially from the privacy point of view, access rights and logging access may be a problem. Towards this end, the dimensional database should contain a declarative model of the data, including descriptions in natural language, and have constructs for providing access to different users in a fine-grained manner. Also, the database should provide an interface that allows exploring both the metadata (descriptions of data) and the data itself. Making the data collected visible and useable more widely may also improve the acceptability of the investment.

Indeed, the focus on managing data may have been too much on technology. Tackling challenges such as ingesting vast amounts of data and running queries involving vast amounts of data have been extensively studied. Unquestionably, these issues must be resolved but eventually, data must be made available for people in order for them to run the processes.

7.7. Nivel² vs. standard UML class diagrams

In this paper, we have applied the Nivel², a multi-level modelling language under development, to represent both the concepts used to model experiments as well as the experiments and their instances, i.e. runs themselves. Given that standard UML[12] class diagrams enjoy extensive tool support and are well understood in the software engineering community, one may justifiably ask what are the benefits of using a modelling language still under development.

Fig. 7 contains a model of the experimentation concept at various levels of abstraction represented using Nivel², in panel (a), and UML class diagrams with objects, panel (b). The levels of abstraction are: (1) the concept of experiment itself, represented by the entity and class Experiment in Nivel² and UML, respectively; (2) the experiment definitions (ColourExperiment) containing at hypotheses etc. that can be run at various times or in various areas, resulting in (3) experiment runs (an instance of ColourExperiment ExperimentRun). The benefits of Nivel² become apparent especially when considering the second level: in Nivel², entities on that level are both instances of Experiment and types of experiment runs, and the Nivel² instantiation semantics covers the instantiation between both levels (1) and (2), and (2) and (3). In UML, on the other hand, the object of type ExperimentRun is both linked to the ColourExperiment and in class ExperimentRun: hence, two model elements are needed to characterise the object. Moreover, the instance of link bears no semantics as such. Instead, the semantics should be defined using some auxiliary construct.

The benefits of Nivel² over UML extend even beyond the conceptual difficulties described above. Whereas Nivel² tool support is largely based on the ability to create instances of entities in a straightforwards and guided manner, much of the UML tool support is still, to the best of our knowledge organised around creating class diagrams and mapping the classes in a programming language. Instantiation takes place only when the software is run, and objects of the classes are created.

7.8. Threats to validity

Validity can be considered as the approximate truth of an inference or knowledge claim [31, p. 34]. In the following, we address the potential threats to validity in the proposed artefact or the rationale of the design decisions behind it, and the answers we have provided to the research questions. In more detail, we address the threats to validity by addressing internal, construct and external validity [31,32].

Internal validity is essentially about causal claims. We do not make any strong causal claims in the paper. However, some aspects of this kind are implied by the necessary support and conceptualisation addressed in RQ1 and RQ2, which in constructive research turns into a question of whether they can be feasibly conceptualised and implemented. A straightforward way of addressing that is by creating artefacts that demonstrate the feasibility. This was done in the form of the Undulate framework.

Construct validity concerns the constructs used or created in the research, e.g. whether they are conceptually clearly explicated and coherent with respect to each other and to the more general usage of the same concepts. The concepts used in the Undulate framework are rigorously defined, so the threat to their explication or coherence in relation to each other can be considered minimal. Furthermore, the implementation of the framework described in Section 4.2 further helps to ensure the non-existence of any major issues with the conceptualisation. The conceptual basis of Nivel² has a solid basis in existing work in multi-level modelling which lessens the potential for any confusion with the usage of the concepts in more general.

[12] Unified Modeling Language, https://www.uml.org/.

There are, however, central concepts that may have different interpretations in different contexts. For example, the terms *experiment* and *hypothesis* have established meanings in research methodology, and their usage in continuous experimentation in software engineering reflects those but is not always exactly the same. Experimentation in software engineering practice is not based on scientifically solid theories, but the hypotheses may have a more practical purpose in explicating what is being tested. This may cause confusion and misinterpretation, thus posing a potential threat to construct validity, particularly if the reader considers experimentation from a scientific perspective rather than as a practical means in soft computing.

External validity concerns how the knowledge claims are valid, or in more general terms the results, applicable beyond the context of the current research. From yet another point of view, the external validity pertains to the *limitations* of the applicability of the results. Our scope in the paper was the development of web-facing software services, as it gives a clear and simple setting for experimentation. There are several characteristics that make web-based software highly suitable for continuous experimentation, such as the ability to easily route users to different software variants, flexible means to dynamically deploy software, vast possibilities for collecting usage data, etc. In another application domain or development setting, variations in these may affect the feasibility of the approach. As an example, physical devices without a network connection are considerably more difficult to experiment with, although, in principle, it would be possible to alter the software or its configuration in different devices and collect data from the devices in offline mode. When considering applying the results in other contexts, these should be considered. The external validity is thus highly dependent on the context and assumes the responsibility from the one aiming to transfer any of the results to another context.

7.9. Implications

The Undulate framework introduced in this paper has a number of implications both for the research and practice of data-driven software engineering in general and continuous experimentation in particular. So far, both data-driven software engineering and continuous experimentation have not become widely applied but are to a large extent only practised by large companies. This is most likely due to the significant investments in various forms of infrastructure and process development required to implement such processes.

The Undulate framework facilitates the implementation of data-driven practices in a number of ways. First, software engineering and business processes are given characterisations by the data they use as input and produce as output. Second, the characterisations enable utilising the process output for controlling the processes themselves, thus creating a feedback loop. The processes can be controlled in part or whole automatically, and the automatic control may involve soft computing technologies.

From the experimentation point of view, a representation of experimentation knowledge using Nivel^2 was defined. Such a representation and the tool support for Nivel^2 make managing experiment data at various levels of abstraction easier, thus reducing the investment required to introduce more ambitious forms of continuous experimentation.

In summary, the contributions in this paper make advanced software engineering techniques available to a larger portion of software companies. From the scientific point of view, key implications include demonstrating the usability of multilevel modelling languages for representing engineering knowledge as well as creating a more solid conceptual basis of experimentation knowledge.

8. Conclusions and further work

In this paper, we have contributed towards the theory and practice of data-driven software engineering by introducing the Undulate framework. Further, in the context of the framework, we have elaborated on the experimentation process as an example of a continuous software engineering process that can be driven by and automated using data.

The work presented in this paper can be extended in multiple ways. An obvious avenue would be to study other software processes to see if they indeed fit the framework as expected or should the framework be extended.

Another perspective would be to study the literature and available implementations, both open source and proprietary, of dimensional databases. As noted in the paper, there is relatively little research on the topic available. Such a database would need to ingest and contain large amounts of data while still providing near real-time query performance for usability. Any existing implementation or a prototype implemented as a part of research would need to match these stringent requirements.

Case studies in real companies would be needed to evaluate the applicability of the framework in practice. Ideally, the case studies should cover companies producing different kinds of software (online, on-premise, embedded, etc.) and a scale of processes as wide as possible.

CRediT authorship contribution statement

Timo Asikainen: Conceptualization, Methodology, Software, Writing – original draft, Writing – review & editing. **Tomi Männistö:** Conceptualization, Methodology, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

The work was supported by the Academy of Finland (project 317657).

References

[1] H. Holmstöm Olsson, J. Bosch, Data driven development : Challenges in online, embedded and on-premise software, in: X. Franch, T. Männistö, S. Martínez-Fernández (Eds.), Proceedings of Product-Focused Software Process Improvement (PROFES) 2019, Lecture Notes in Computer Science 11915, 2019, pp. 515–527.

[2] F. Auer, M. Felderer, An infrastructure for platform-independent experimentation of software changes, in: T. Bureš, R. Dondi, J. Gamper, G. Guerrini, T. Jurdziński, C. Pahl, F. Sikora, P.W.H. Wong (Eds.), SOFSEM 2021: Theory and Practice of Computer Science, Springer International Publishing, Cham, 2021, pp. 445–457.

[3] F. Auer, C.S. Lee, M. Felderer, Continuous experiment definition characteristics, in: Proceedings — 46th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2020, 2020, pp. 186–190, http://dx.doi.org/10.1109/SEAA51224.2020.00041.

[4] F. Fagerholm, A. Sanchez Guinea, H. Mäenpää, J. Münch, The RIGHT model for continuous experimentation, J. Syst. Softw. (2017) http://dx.doi.org/10.1016/j.jss.2016.03.034.

[5] H. Holmström Olsson, J. Bosch, The HYPEX model: From opinions to data-driven software development, in: J. Bosch (Ed.), Continuous Software Engineering, Springer International Publishing, Cham, 2014, pp. 155–164, http://dx.doi.org/10.1007/978-3-319-11283-1_13.

[6] X. Franch, C. Gómez, A. Jedlitschka, L. López, S. Martínez-Fernández, M. Oriol, J. Partanen, Data-driven elicitation, assessment and documentation of quality requirements in agile software development, in: J. Krogstie, H.A. Reijers (Eds.), Proceedings of CAiSE 2018: Advanced Information Systems Engineering, Lecture Notes in Computer Science 10816, Springer, 2018, pp. 587–602, http://dx.doi.org/10.1007/978-3-319-91563-0_36.

[7] A. Henriksson, J. Zdravkovic, Holistic data-driven requirements elicitation in the big data era, Softw. Syst. Model. online fir (2021) http://dx.doi.org/10.1007/s10270-021-00926-6.
[8] I. Sommerville, Software Engineering, 9th ed., Addison-Wesley, Harlow, England, 2010.
[9] D. Patil, Building Data Science Teams, O'Reilly Media, 2011.
[10] B. Fitzgerald, K.J. Stol, Continuous software engineering: A roadmap and agenda, J. Syst. Softw. 123 (2017) 176–189, http://dx.doi.org/10.1016/j.jss.2015.06.063.
[11] H. Holmström Olsson, J. Bosch, Towards continuous customer validation: A conceptual model for combining qualitative customer feedback with quantitative customer observation, in: J.M. Fernandes, R.J. Machado, K. Wnuk (Eds.), Software Business, Springer International Publishing, Cham, 2015, pp. 154–166.
[12] D.I. Mattos, J. Bosch, H. Holmström Olsson, Your system gets better every day you use it: Towards automated continuous experimentation, in: Proceedings — 43rd Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2017, 2017, pp. 256–265, http://dx.doi.org/10.1109/SEAA.2017.15.
[13] F. Fagerholm, A. Sanchez Guinea, H. Mäenpää, J. Münch, Building blocks for continuous experimentation, in: 1st International Workshop on Rapid Continuous Software Engineering, RCoSE 2014 — Proceedings, 2014, pp. 26–35, http://dx.doi.org/10.1145/2593812.2593816.
[14] G. Schermann, J. Cito, P. Leitner, Continuous experimentation: Challenges, implementation techniques, and current research, IEEE Softw. 35 (2) (2018) 26–31, http://dx.doi.org/10.1109/MS.2018.111094748.
[15] G. Schermann, D. Schöni, P. Leitner, H.C. Gall, Bifrost — Supporting continuous deployment with automated enactment of multi-phase live testing strategies, in: Proceedings of the 17th International Middleware Conference (Middleware'16), ACM, Trento, Italy, 2016, pp. 1–14, http://dx.doi.org/10.1145/2988336.2988348.
[16] S.G. Yaman, M. Munezero, J. Münch, F. Fagerholm, O. Syd, M. Aaltola, C. Palmu, T. Männistö, Introducing continuous experimentation in large software-intensive product and service organisations, J. Syst. Softw. 133 (2017) 195–211, http://dx.doi.org/10.1016/j.jss.2017.07.009.
[17] D. Issa Mattos, P. Dmitriev, A. Fabijan, J. Bosch, H. Holmström Olsson, An activity and metric model for online controlled experiments, in: M. Kuhrmann, K. Schneider, D. Pfahl, S. Amasaki, M. Ciolkowski, R. Hebig, P. Tell, J. Klünder, S. Küpper (Eds.), Product-Focused Software Process Improvement, Springer International Publishing, Cham, 2018, pp. 182–198.
[18] P. Dmitriev, S. Gupta, D.W. Kim, G. Vaz, A Dirty Dozen: Twelve common metric interpretation pitfalls in online controlled experiments, in: Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Vol. Part F1296, 2017, pp. 1427–1436, http://dx.doi.org/10.1145/3097983.3098024.
[19] T. Asikainen, T. Männistö, Nivel: A metamodelling language with a formal semantics, Softw. Syst. Model. 8 (4) (2009) 521–549, http://dx.doi.org/10.1007/s10270-008-0103-2.
[20] C. Atkinson, T. Kühne, Reducing accidental complexity in domain models, Softw. Syst. Model. 7 (3) (2008) 345–359, http://dx.doi.org/10.1007/s10270-007-0061-0.
[21] W. Inmon, Building the Data Warehouse, Vol. 4th ed, Wiley, 2005.
[22] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in Software Engineering, Springer Science & Business Media, 2012.
[23] W. Li, Y. Lemieux, J. Gao, Z. Zhao, Y. Han, Service mesh : Challenges, state of the art, and future research opportunities, in: 2019 IEEE International Conference on Service-Oriented System Engineering (SOSE), IEEE, 2019, pp. 122–1225, http://dx.doi.org/10.1109/SOSE.2019.00026.
[24] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, J. Wilkes, Borg, omega, and kubernetes, Commun. ACM 59 (5) (2016) 50–57, http://dx.doi.org/10.1145/2890784.
[25] D. Bernstein, Containers and cloud: From LXC to docker to kubernetes, IEEE Cloud Comput. 1 (3) (2014) 81–84, http://dx.doi.org/10.1109/MCC.2014.51.
[26] I. Figalist, C. Elsner, J. Bosch, H. Holmström Olsson, Breaking the vicious circle: A case study on why AI for software analytics and business intelligence does not take off in practice iris, J. Syst. Softw. (2021) 111135, http://dx.doi.org/10.1016/j.jss.2021.111135.
[27] D. Torre, M. Alferez, G. Soltana, M. Sabetzadeh, L. Briand, Model driven engineering for data protection and privacy: Application and experience with GDPR, Softw. Syst. Model. online fir (2020) http://dx.doi.org/10.1007/s10270-021-00935-5, URL http://arxiv.org/abs/2007.12046.
[28] J. Ahlgren, M.E. Berezin, K. Bojarczuk, E. Dulskyte, I. Dvortsova, J. George, N. Gucevska, M. Harman, R. Lämmel, E. Meijer, S. Sapora, J. Spahr-Summers, WES: Agent-based user interaction simulation on real infrastructure, in: Proceedings — 2020 IEEE/ACM 42nd International Conference on Software Engineering Workshops, ICSEW 2020, 2020, pp. 276–284, http://dx.doi.org/10.1145/3387940.3392089.
[29] J. Ahlgren, K. Bojarczuk, S. Drossopoulou, I. Dvortsova, J. George, N. Gucevska, M. Harman, M. Lomeli, S.M. Lucas, E. Meijer, S. Omohundro, R. Rojas, S. Sapora, N. Zhou, Facebook's cyber-cyber and cyber-physical digital twins, in: ACM International Conference Proceeding Series, Association for Computing Machinery, 2021, pp. 1–9, http://dx.doi.org/10.1145/3463274.3463275.
[30] K. Ahlgren, J. Bojarczuk, M. Dvortsova, I. Harman, R. Hatout, M. Lomeli, E. Meijer, S. Sapora, Behavioural and structural imitation models in Facebook's ww simulation system, in: Proceedings of International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering (RAISE), 2021, p. n/a.
[31] W. Shadish, T.D. Cook, D.T. Campbell, Experimental and Quasi-Experimental Designs for Generalized Causal Inference, Houghton Mifflin, Boston, 2002.
[32] P. Runeson, M. Höst, Guidelines for conducting and reporting case study research in software engineering, Empir. Softw. Eng. 14 (2) (2009) 131–164.