International Journal of Conceptual Modeling
February 2018. DOI:10.18417/emisa.si.hcm.19

Special Issue on Conceptual Modelling in Honour of Heinrich C. Mayr

Model-Driven Time-Series Analytics

Sabine Wolny[*,a], Alexandra Mazak[a], Manuel Wimmer[a], Rafael Konlechner[a], Gerti Kappel[a]

[a] CDL-MINT, TU Wien, Austria

Abstract. *Tackling the challenge of managing the full life-cycle of systems requires a well-defined mix of approaches. While in the early phases model-driven approaches are frequently used to design systems, in the later phases data-driven approaches are used to reason on different key performance indicators of systems under operation. This immediately poses the question how operational data can be mapped back to design models to evaluate existing designs and to reason about future re-designs. In this paper, we present a novel approach for harmonizing model-driven and data-driven approaches. In particular, we introduce an architecture for time-series data management to analyse runtime properties of systems which is derived from design models. Having this systematic generation of time-series data management opens the door to analyse data through design models. We show how such data analytics is specified for modelling languages using standard metamodelling techniques and technologies.*

Keywords. Model-Driven Engineering • Time-Series • Data Analytics • Language Engineering

1 Introduction

In model-driven engineering (MDE), models are the central artefact and used as a main driver throughout the software development process, finally leading to an automated generation of software systems (Lara et al. 2015). In the current state-of-practice in MDE (Brambilla et al. 2017; Karagiannis et al. 2016), models are used as an abstraction and generalization of a system to be developed. By definition, a model never describes reality in its entirety, rather it describes a scope of reality for a certain purpose in a given context (Brambilla et al. 2017). Thus, models are mostly used as *prescriptive models* for creating a software system (Heldal et al. 2016). Such design models determine the scope and details of a domain of interest to be studied. For this purpose, different types of general modelling languages (e. g., state charts, class diagrams, etc.) may be used or domain-specific languages (DSLs) (Karagiannis et al. 2016) may be employed. It has to be emphasized that engineers typically have the desirable behaviour in mind when creating a system, since they are not aware in these early phases of many deviations that may take place at runtime (van der Aalst 2016).

According to Brambilla et al. (2017) the implementation phase deals with the mapping of prescriptive models to some executable systems and consists of three levels: (*i*) the *modelling level* where the models are defined, (*ii*) the *realization level* where the solutions are implemented through artefacts that are used in the running system, and (*iii*) the *automation level* where mappings from the modelling to the realization phase are made. However, these levels are currently only used for down-stream processes. The possibility of up-stream processes is mostly neglected in MDE (Mazak and Wimmer 2016). Especially, for later phases of the system lifecycle *descriptive*

* Corresponding author.
E-mail. wolny@big.tuwien.ac.at
This work has been supported by the National Foundation for Research, Technology and Development (CDG), the Austrian Federal Ministry of Science, Research, and Economy (BMWFW Austria), and the TU Wien research funds.

models may be employed to better understand how the system is actually realized and how it is operating in a certain environment (Mazak and Wimmer 2016). Compared to prescriptive models, those descriptive models are only marginal explored in the field of MDE, and if used at all, they are built manually.

In this paper, we move towards a well-defined mix of approaches to better manage the full life-cycle of systems by combining prescriptive and descriptive model types. In particular, we introduce a model-driven time-series data analytics architecture for harmonizing model-driven and data-driven approaches. Based on this architecture, we show how data analytics can be specified for modelling languages using standard metamodelling techniques. This means, design-oriented languages are equipped with extensions for representing runtime states as well as runtime histories, which in turn allow the formulation and computation of runtime properties with the Object Constraint Language (OCL). This approach has the advantage to directly interpret measurements and events within the design models without introducing an impedance mismatch.

The remainder of this paper is structured as follows. Section 2 provides the background for this paper by introducing a motivating example which is subsequently used as running example. Section 3 gives an overview of our architecture for unifying model-driven and data-driven approaches. In Section 4, we present in detail how time-series analytics can be integrated in metamodels. Section 5 discusses the related work. Finally, in Section 6, we conclude with an outlook on future work.

2 Motivating Example

In this section, we introduce a motivating example, which will subsequently become the running example of this paper. We first describe the example from the modelling perspective, then from the realization perspective with a focus on runtime data collection, and finally conclude with the challenges we aim to address with this paper.

Model-Driven Perspective

As our motivating example, we consider a grip-arm robot (gripper) with different position properties of axis angles: `BasePosition (BP)`, `MainArmPosition (MAP)`, and `GripperPosition (GP)`. From a device point of view (cf. Figure 1(a)), the structure of the gripper component and its behaviour are modelled at design time by a subset of a SysML-like language, i.e., blocks with associated state machines. The top of Figure 1(a) shows the specific properties (`BP`,`MAP`,`GP`) of the block, whereas the actual property values depend on the different states (e.g., `Idle`, `Pick Up`). The states are given at the bottom of Figure 1(a).

By the given state machine, property value changes are modelled. The gripper has certain positions at initialization, in state `Idle` and in state `Pick Up`. The assumption of the modelled state machine is that as soon these states are reached, the position values are set. However, such state machines are a kind of black box, where only the discrete values before entering the state and after leaving the state are known (cf. Figure 1(b)). While this may be sufficient for several design tasks and discrete systems, for continuous systems more information may be required. This is in particular true for our example case. The gripper represents a continuous system, since it does not immediately realize the next position, but needs time to move to the given place. Usually, such information is not directly given in a design model, but it may be important for several tasks such as optimization, validation, and verification. The ability to observe property value changes over time within states may contribute to capture the current capabilities and shortcomings of systems. Thus, the presented approach of this paper aims to transform the black box into a so-called “grey box” to make the effects of value changes visible (see Figure 1(c)). For instance, observation sequences of property value changes are an important base information of a system’s operation to compute operating figures to check if the behaviour of each

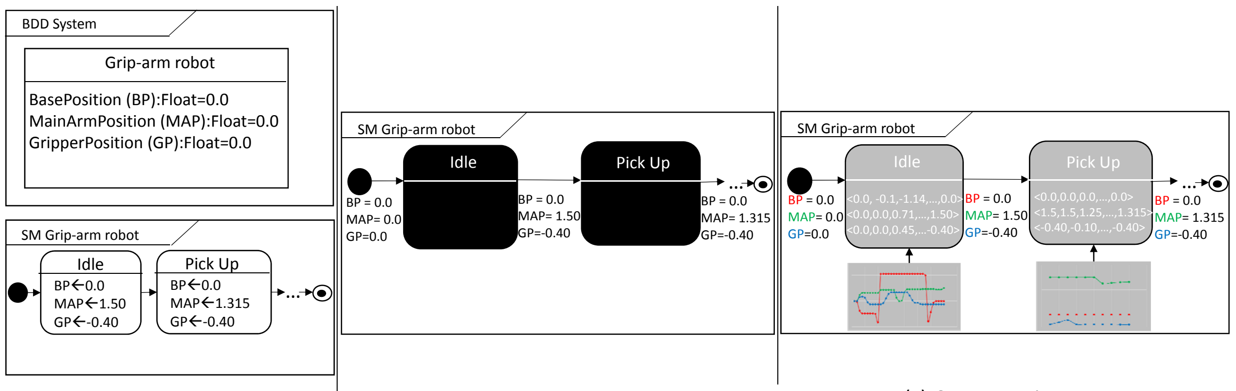

Figure 1: Different model-based views on a grip-arm robot.

gripper's axis corresponds to the defined one in the design model.

Data-Driven Perspective

For the technical realization of our example, we developed a simulation model of the gripper consisting of three angle sensors, which we executed by the open source tool Blender[1] . We deploy the scenario of a pick-and-place unit, where the gripper picks up different color-coded work pieces, place them on a test rig, picks the items up again and puts them down, depending on their red or green color, in two different storage boxes. The simulation environment receives its commands via Message Queue Telemetry Transport (MQTT) from a server controller implemented with Kotlin[2] . The simulation enables to acquire transient data streams in real-time from the angle axes of the gripper (BP, MAP, GP, unit is radian), which are equipped with sensors.

To react on events of interest provided by these data streams, we employ the publish/subscribe pattern. In our example, we subscribe to the sensor topic to receive in a temporal distance of 15 milliseconds the filtered data streams of the sensors of the gripper during simulation. Thereby, we are interested in property value changes (i. e., positions of the axes) in the simulation at given points in time. Messages from the sensor topic are defined in JSON[3] specifying the sending unit as well as the measured data. The following example shows such a message from the angle sensors of the gripper to the controller.

```
{"entity": "GripperArm",
"basePosition": 0.0,
"mainArmPosition": 0.0,
"gripperPosition": 0.0}
```

This example shows the positions of the angle axes at system initialization (see Figure 1). The angle position of each axis has the value 0.0. The default range of the angle values is $[-\pi,\pi]$. To analyze our scenario, it is important to save the measured data over time. For this purpose, we use the time series database InfluxDB[4]. InfluxDB allows us to store a large amount of time-stamped data. In addition, by the tool Grafana[5] we can visualize our stored sensor values.

Challenges

Our motivating example is discussed from two angles: (*i*) from the model-driven, i. e., how the intended system should work, and (*ii*) from the data-driven, i. e., how measurements can be taken from the running system to reason about the actual realization. While the first perspective is lacking concepts to define runtime data such as

[1] https://www.blender.org
[2] https://kotlinlang.org
[3] http://json.org/example.html
[4] https://www.influxdata.com
[5] https://grafana.com

time-series, the second perspective has to correctly interpret the collected measurements. The challenge is how to overcome the gap between those two perspectives (*i*) to monitor important data from operation, (*ii*) to align the measurements with the design model in order to provide a semantic anchoring of the data, and (*iii*) to provide meaningful analytics whereas the results of the analytics are interpretable for the given design models to reason about improvements or fulfilments of given requirements.

3 Unifying Architecture for Model-Driven and Data-Driven Approaches

In order to allow a smooth integration of model-driven and data-driven approaches, we present in this section an architecture, which builds on the classical model to system downstream in terms of code generators, but at the same time, supports an upstream in terms of mapping data back to design models. Figure 2 gives an overview of this architecture. In the following section, a more detailed description of the different parts will be presented based on our running example.

The proposed architecture consists of four main parts. First, the left hand side of Figure 2 captures the classical downstream MDE approach (cf. (a) in Figure 2). At the metamodel layer, the `design language` is defined with the help of a metamodelling language (in our setting Ecore). Conforming to the design language, the `design models` are defined at the model level describing the static (i. e., structure) and dynamic aspects (i. e., behaviour) of a system to be developed. For the vertical transition from the modelling to the realization level we assume the existence of model-to-text transformations for code generation. Thus, this part of our architecture describes how we can derive the executable system from the design model as is the state-of-the-art in MDE.

Second, we continue with defining the first part of the up-stream process of runtime data to the design model (cf. (b) in Figure 2). In addition to the actual systems, the `runtime observer` is generated out of the design model. The runtime observer collects important information from the running system to represent the current state of the system. Those observations should not only be recorded by observing the running system, but should be also representable at the model level. Thus, we extend the `design language` with a dedicated `runtime language`. This metamodel defines the syntax to represent snapshots of the running system connected to the design model elements. Those snapshots are represented in the `runtime state models` which extend the `design models` and may be directly updated by the `runtime observer` during runtime. In summary, this part of our architecture maps runtime data at the model level for one single point in time and may be used to monitor a system on the model level.

Third, we define the runtime history of a system (cf. (c) in Figure 2). For reasoning about, e.g., property value distributions, it is important to have the complete history of value changes as starting point as one snapshot is definitely not sufficient for such computations. Thus, in the `time series database` the observations of the running system are stored. Based on these collected observations, the `runtime history models` may be directly updated. These models conform to the `runtime history language`, which is an extension of the `runtime language`. In the `runtime history language`, the syntax is defined for representing histories of runtime phenomena of interest, e.g., property values, events, etc.

Finally, after defining those concepts for storing observation histories at the model level, it is also possible to analyse the stored observations (cf. (d) in Figure 2). For this purpose, we define `runtime properties` based on the Object Constraint Language (OCL) by introducing derived properties for the metamodel elements. These derived properties enable us, e.g., (*i*) to compute descriptive statistics, (*ii*) to evaluate monotony behaviour of value changes, or (*iii*) to compute lower and upper bounds of properties to mention just a few examples. Based on the `runtime properties`, the `runtime property values` are computed by analysing the collected time-series. Thus, runtime

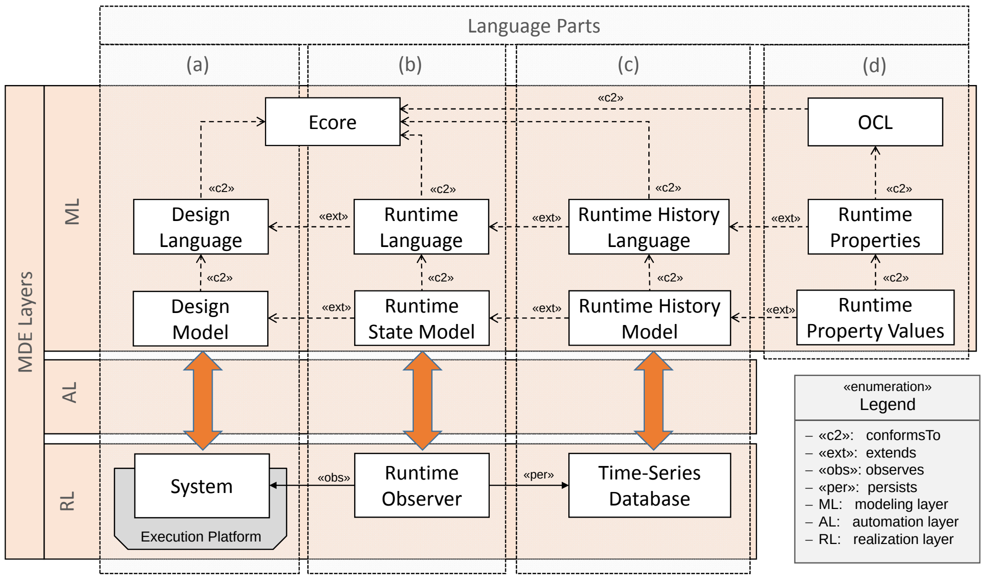

Figure 2: Unifying architecture for model-driven and data-driven perspectives.

data is back propagated to the design models and this mapping allows to interpret the data through the design model elements as there is a clear traceability guaranteed from design elements, runtime states, and runtime histories.

By this architecture, we are able to harmonize model-driven and data-driven approaches, where time-series data management of systems at runtime can be derived from initial design models and be used again at the model layer by importing the time-series to model structures. How this model structures are defined is the topic of the next section.

4 Metamodelling Blueprint for Enhancing Models with Time-Series Analysis

Based on our running example, we further detail in this section how the afore presented architecture can be realized for a given language. In particular, we show for the introduced design modelling language, how the extensions for runtime states, runtime histories, and runtime properties are defined as reusable metamodelling blueprints. The time-series analysis we are focusing on for demonstration purposes is about property value changes of the axis angles (i. e., BP, MAP, GP) of the gripper in our running example.

Design Elements

As already mentioned before, our starting point is the availability of a design modelling language expressed in Ecore. For our running example, we model the structure of the gripper with its properties as a kind of block diagram similar to what is known from SysML. A block has an associated state machine, where different states and transitions are defined. For states, assignments can be defined, which are executed when a state is activated. The assignments in our exemplary language are simple value assignments for the properties of a block. The resulting metamodel for the described design language is shown in Figure 3.

Runtime States

In order to express concrete runtime states on the model level, the metamodel has to be extended with runtime concepts. For this task, there are several existing approaches available, e. g., (Engels et al. 2000; Mayerhofer et al. 2013; Meyers et al. 2014). Most of them add additional metamodel elements to the design language to describe what

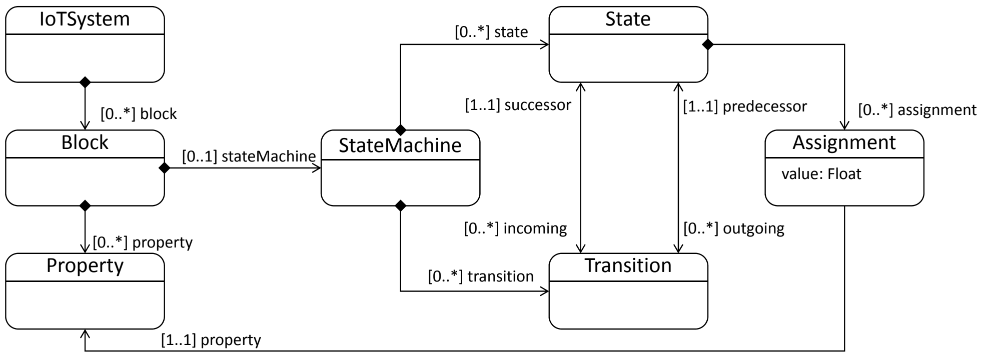

Figure 3: Design metamodel for the running example.

runtime phenomena are of interest and how they are connected to design concepts. For our running example, the `runtime language` is considered as an extension of the design metamodel to allow representing property values for a given point in time (i. e., for a snapshot of the running system). In addition, transitions may fire during runtime. Thus, the concept of transition firing is introduced. While values are considered by measurements during the operation phase, the firing of transitions are categorized as events. Please note that the relation to the design concepts has to be clearly stated by the runtime concepts, e. g., the value concept is related to the property concept. Figure 4 captures the concrete realization of the runtime extension for our design language.

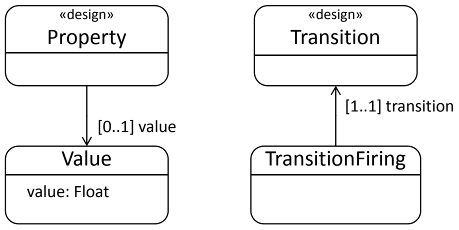

Figure 4: Runtime metamodel for the running example.

Runtime Histories

To reason about operation figures going beyond one snapshot in time such as distributions, upper and lower bounds, histories of property values and event sequences are necessary. Therefore, we need another extension which allows to represent the runtime history of a system. For this, we introduce a novel metamodelling blueprint which introduces the concept of history by providing a sequence of steps having a particular timestamp associated. Figure 5 illustrates the separation of steps into measurement snapshots and event snapshots. These specific steps are forming the event histories and measurement histories. The measurement history contains all measurement snapshots, which comprise values for given time steps. Event histories do the same for events. In our running example, the measurement snapshots refer to the value runtime concept introduced by the runtime extension and the event snapshots are referring to the transition firing concept.

Having this base structure introduced allows us to represent time-series data in design models by using runtime concepts as glue between models and data.

Runtime Properties

For analysing the time-series data represented in the aforementioned runtime history models, we introduce derived properties which actually represent runtime properties. Derived properties have been already used heavily in the past for deriving additional information from given structures and values. As we explicitly represent runtime histories as model structures, we can make use of derived properties to derive runtime information from the base time-series recorded during operation.

In the following we state three runtime properties for the given design language, namely for

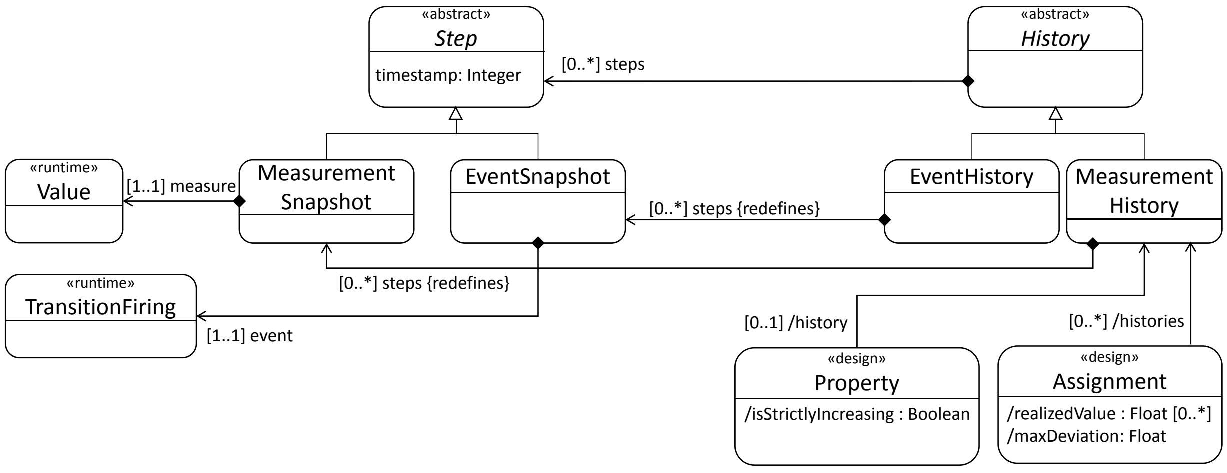

Figure 5: Runtime history metamodel for the running example.

the `Property` metaclass and the `Assignment` metaclass. We use standard OCL to derive the runtime properties.

For properties defined in blocks, it may be of interest if their values are strictly increasing over time or not. This can be expressed in OCL by providing a derived history reference for properties from the complete measurement history. The reference only contains the slice of the full history which concerns the given property. Using this reference, we can simply collect all values as a sequence (the ordering expresses the occurrence of the values). If the sorted sequence corresponds to the base sequence, then the property is strictly increasing.

```
context Property::isStrictlyIncreasing:
    ↪Boolean
derive: self.history.steps.measure.value->
    ↪flatten()->sortedBy(x|x) = self.
    ↪history.steps.measure.value->flatten()
```

Concerning the assignments within states, one may be interested if the stated value is actually realized by the system. For this, the realized values may be collected by taking the last snapshots of all assignment executions for a given assignment.

```
context Assignment::realizedValues:Set(Float
    ↪)
derive: self.histories -> collect(x|x.steps
    ↪-> last()) -> collect(x|x.measure.
    ↪value) -> asSet()
```

Having the set of realized values, the maximum deviation is calculated by introducing another derived property which builds on the previous one.

```
context Assignment::maxDeviation:Float
derive: self.realizedValues -> collect(x|(
    ↪self.value-x).abs()) -> sortedBy(x|x)
    ↪-> last()
```

5 Related Work

In this section, we discuss existing work with respect to the contribution of this paper, namely the combination of model-driven and data-driven approaches with a focus on time-series analytics. Therefore, we first discuss data-driven approaches for enhancing existing domain specific languages (DSLs), and subsequently, we enumerate existing work which proposes dedicated DSLs for time-series analytics.

Data-driven approaches for DSLs
An emerging field for data-enhanced modelling languages is Web engineering. For instance, Bernaschina et al. (2017) point to the fact that there is the need for merging Web site navigation statistics of user behaviour with the structure of

the Web application models. The authors show the advantages of combining user interaction models with user tracking information in form of user navigation logs, and details about the visualized content in the pages. Their approach interweaves design time information and runtime execution data of Web sites in order to significantly improve the analysis of user behaviour. In (Artner et al. 2017), we combined navigation models with Markov chains for representing navigation path probabilities, which are derived from execution logs. While these existing approaches for Web applications follow the general idea of combining data-driven and model-driven approaches, the approach of this paper is independent from the actual domain and may be also used in the future to reproduce these existing specific approaches.

Another very active research field is process mining (van der Aalst 2016) which aims to discover process models from workflow execution logs. A variety of process mining algorithms exists that allows the discovery of different process models in different formalisms. In (Wolny et al. 2017), we present an initial architecture how process mining may be related with time-series mining. By this, not only the dependencies between different process steps may be uncovered, but also dependencies between data and process steps are approachable.

Finally, in (Hartmann et al. 2017) the authors present a DSL which allows not only the specification of structural aspects of a systems, but also the definition of so-called learned properties. Such properties are computed from runtime data by using some kind of machine learning algorithms. Our approach directly allows to encode such properties as derived properties based on time-series data computed with OCL as we model the runtime history explicitly. In future work, it will be interesting to combine our time-series analysis with machine learning algorithms as proposed by Hartmann et al. (2017).

DSLs for Time-Series Analytics
The OMS3 modelling framework[6] introduces an extensible and lightweight layer for a simulation description expressed as Simulation DSL by using Groovy[7] as a framework for providing the code generator implementation. In (David et al. 2012), the authors present DSL variants in OMS3, e. g., the Ensemble Streamflow Prediction (ESP) DSL. This DSL uses time-series of historic meteorological data as model input to predict future conditions. In their approach, DSLs are employed for time-series unlike in our approach, where we use time-series for domain-specific modelling.

Gekko[8] is an open-source modelling approach for time-series data management and for solving and analysing large-scale time-series models. Gekko may be considered as a kind of DSL with a time-series domain focus. It provides interfaces to statistic packages such as R. In our approach, we use an open-source time-series database which offers besides high-availability storage and monitoring of time-series, application metrics and real-time analytics in addition. Nevertheless, in future work it is of interest to evaluate different possibilities to perform time-series analytics in addition to our current approach.

6 Conclusion and Future Work

In this paper, we have introduced an unifying architecture for combining model-driven and data-driven approaches for system engineering. By this architecture, we allow for specifying and computing runtime properties based on time-series data through design models. The extensions needed on the metamodel level are non-intrusive and connected to existing approaches for specifying the operational semantics of languages. The presented runtime history metamodel fragments are applicable for any design modelling language comprising features to be measured and events to be tracked as the current metamodelling languages Ecore and OCL are reused. We demonstrated our

[6] https://alm.engr.colostate.edu/cb/project/oms
[7] http://groovy-lang.org
[8] http://t-t.dk/gekko

approach for a cyber-physical production system case. We have also realized a prototype in Eclipse supporting our approach which is available on our project website[9] .

While the presented approach opens the door for using time-series analytics in a model-driven engineering toolbox, there are still several challenges to be tackled in the future. In particular, we consider the following points on our roadmap: *scalability* (e. g., should the analysis be performed on the model level or directly in the time-series database?), *expressivity* (e. g., which extensions of OCL are necessary for statistical reasoning?), *understandability* (e. g., how to visualize time-series oriented information in diagrams?), and *predictability* (e. g., how to derive and use operations from time-series for predicting future runtime states?).

References

Artner J., Mazak A., Wimmer M. (2017) Towards Stochastic Performance Models for Web 2.0 Applications. In: Proceedings of the 17th International Conference on Web Engineering (ICWE). Springer, pp. 360–369

Bernaschina C., Brambilla M., Mauri A., Umuhoza E. (2017) A Big Data Analysis Framework for Model-Based Web User Behavior Analytics. In: Proceedings of the 17th International Conference on Web Engineering (ICWE). Springer, pp. 98–114

Brambilla M., Cabot J., Wimmer M. (2017) Model-Driven Software Engineering in Practice. Morgan & Claypool

David O., Lloyd W., II J. C. A., Green T. R., Olson K., Leavesley G. H., Carlson J. (2012) Domain Specific Languages for Modeling and Simulation: Use Case OMS3. In: Proceedings of the International Congress on Environmental Modelling and Software Managing Resources of a Limited Planet, pp. 1201–1207

[9] https://cdl-mint.big.tuwien.ac.at/case-study-artefacts-for-emisa-2017

Engels G., Hausmann J. H., Heckel R., Sauer S. (2000) Dynamic Meta Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML. In: Proceedings of the 3rd International Conference on the Unified Modeling Language (UML). Springer, pp. 323–337

Hartmann T., Moawad A., Fouquet F., Le Traon Y. (2017) The next evolution of MDE: a seamless integration of machine learning into domain modeling. In: Software & Systems Modeling

Heldal R., Pelliccione P., Eliasson U., Lantz J., Derehag J., Whittle J. (2016) Descriptive vs prescriptive models in industry. In: Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS). ACM, pp. 216–226

Karagiannis D., Mayr H. C., Mylopoulos J. (2016) Domain-Specific Conceptual Modeling, Concepts, Methods and Tools. Springer

de Lara J., Guerra E., Cuadrado J. S. (2015) Model-driven engineering with domain-specific metamodelling languages. In: Software and System Modeling 14(1), pp. 429–459

Mayerhofer T., Langer P., Wimmer M., Kappel G. (2013) xMOF: Executable DSMLs Based on fUML. In: Proceedings of the 6th International Conference on Software Language Engineering (SLE), pp. 56–75

Mazak A., Wimmer M. (2016) Towards Liquid Models: An Evolutionary Modeling Approach. In: Proceedings of the 18th IEEE Conference on Business Informatics (CBI). IEEE, pp. 104–112

Meyers B., Deshayes R., Lucio L., Syriani E., Vangheluwe H., Wimmer M. (2014) ProMoBox: A Framework for Generating Domain-Specific Property Languages. In: Proceedings of the 7th International Conference on Software Language Engineering (SLE). Springer, pp. 1–20

van der Aalst W. M. P. (2016) Process Mining - Data Science in Action. Springer

Wolny S., Mazak A., Konlechner R., Wimmer M. (2017) Towards Continuous Behavior Mining. In: Proceedings of the 7th International Symposium on Data-driven Process Discovery and Analysis (SIMPDA), pp. 149–150