On Marrying Model-Driven Engineering and Process Mining: A Case Study in Execution-based Model Profiling

Alexandra Mazak and Manuel Wimmer

Business Informatics Group (BIG),
Institute of Software Technology and Interactive Systems,
TU Wien, Favoritenstrasse 9-11, 1040 Vienna, Austria
{mazak,wimmer}@big.tuwien.ac.at
http://www.big.tuwien.ac.at

Abstract. In model-driven engineering (MDE), models are mostly used in prescriptive ways for system engineering. While prescriptive models are indeed an important ingredient to realize a system, for later phases in the systems' lifecycles additional model types are beneficial to use. Unfortunately, current MDE approaches mostly neglect the information upstream in terms of descriptive models from operations to design, which would be highly needed to improve systems continuously. To tackle this limitation, we propose execution-based model profiling as a continuous process to improve prescriptive models at design-time through runtime information by incorporating knowledge in form of profiled metadata from event logs generated during the execution of a code model. For this purpose we combine techniques of process mining (PM) with runtime models of MDE. In the course of a case study, we implement a preliminary prototype of our framework based on a traffic light system example which shows the feasibility and benefits of our execution-based model profiling approach.

1 Introduction

In model-driven engineering (MDE), models are put in the center and used throughout the software development process, finally leading to an automated generation of the software systems [13]. In the current state-of-practice in MDE [2], 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. Thus, models are used as *prescriptive models* for creating a software system [18]. Such *models@design.time* determine the scope and details of a domain of interest to be studied. Thereby, different aspects of the domain or of its solution can be taken into account. For this purpose different types of modeling languages (e.g., state charts, class diagrams, etc.) may be used. It has to be emphasized that engineers typically have the desirable behavior in mind when creating a system, since they are not aware in these early phases of the many deviations that may take place at runtime [3].

According to Brambilla et al. [2] the implementation phase deals with the mapping of prescriptive models to some executable systems and consists of three levels: (i) the

modeling level where the models are defined, (ii) the *realization level* where the solutions are implemented through artifacts that are used in the running system, and (iii) the *automation level* where mappings from the modeling to the realization phase are made. Thus, the flow is from models down to the running realization through model transformations.

While prescriptive or *design models* are indeed a very important ingredient to realize a system, for later phases in the system's lifecycle additional model types are needed. Therefore, *descriptive models* may be employed to better understand how the system is actually realized and how it is operating in a certain environment. Compared to prescriptive models, these other mentioned types of models are only marginal explored in the field of MDE, and if used at all, they are built manually. Unfortunately, MDE approaches have mostly neglected the possibility to describe an existing and operating system which may act as feedback for design models. As theoretically outlined in [12], we propose *model profiling* as a continuous process (i) to improve the quality of design models through runtime information by incorporating knowledge in form of *profiled metadata* from the system's operation, (ii) to deal with the evolution of these models, and (iii) to better anticipate the unforeseen. However, our aim is not to "re-invent the wheel" when we try to close the loop between downstream information derived from prescriptive models and upstream information in terms of descriptive models. There exist already promising techniques to focus on runtime phenomena, especially in the research field of Process Mining (PM) [3]. Thus, our model profiling approach in its first version follows the main idea of combining MDE and PM. The contribution of this paper is to present a unifying architecture for a combined but loosely-coupled usage of MDE approaches and PM techniques.

The remainder of this paper is structured as follows. In the next section, we present a unified conceptual architecture for combining MDE with PM frameworks. In Section 3, we present a case study in execution-based model profiling based on a traffic light system example. In Section 4, we present recent work related to our approach and discuss its differences. Finally, we conclude this paper by an outlook on our next steps in Section 5.

2 Marrying Model-Driven Engineering and Process Mining

In this section, we briefly describe the main building blocks of both, MDE as well as PM, necessary for the context of this paper, before we present a unifying architecture for their combined but loosely-coupled usage.

2.1 Prerequisites

Model-driven Engineering (MDE). In each phase of an MDE-based development process "models" (e.g., analysis models, design models) are (semi-)automatically generated by *model-to-model transformations (M2M)* that take as input those models that were obtained in one of the previous phases. In the last step of this process the final code is generated using *model-to-text transformation (M2T)* from the design model [2]. These transformation engineering aspects are based on the metamodels of a modeling

language, which provide the abstract syntax of that language. This syntax guarantees that models follow a clearly defined structure, and it forms the basis for applying operations on models (e.g., storing, querying, transforming, checking, etc.).

As described in [2], the semantics of a modeling language can be formalized by (i) giving *denotational semantics* by defining a mapping from the modeling language to a formal language, (ii) giving *operational semantics* by defining a model simulator (i.e., for implementing a model execution engine), or (iii) giving *translational semantics* by defining, e.g., a code generator for producing executable code. In order to generate a running system from models, they must be *executable*. This means, a model is executable when its *operational semantics* is fully specified [2]. However, executability depends more on the used execution engine than on the model itself. The main goal of MDE is to get running systems out of models.

In our approach, we consider executable modeling languages which explicitly state "what" the runtime state of a model is, as well as all possible events that can occur during execution [1]. These executable modeling languages not only provide operational semantics for interpreters, but also translational semantics in form of code generators to produce code for a concrete platform to realize the system.

Process Mining (PM). It combines techniques from data mining and model-driven Business Process Management (BPM) [3]. In PM, business processes are analyzed on the basis of *event logs*. Events are defined as process steps and event logs as sequential ordered events recorded by an information system [15]. This means that PM works on the basis of event data instead of designed models and the main challenge is to capture behavioral aspects. Thereby, specialized algorithms (e.g., the α-algorithm) produce a Petri net, which can be easily converted into a descriptive model in form of a process model. To put it in a nutshell, there is a concrete, running system which is producing logs and there are algorithms used to compute derived information from those logs.

Generally in PM, event logs are analyzed from a process-oriented perspective using GPLs (e.g., UML, Petri nets) [4]. There are three main techniques in PM: (i) the well-established *discovery technique* by which a process model can be automatically extracted from log data [3], (ii) the *conformance checking technique*, which is used to connect an existing process model with an event log containing data related to activities (e.g., business activities) of this process [11], and (iii) the *enhancement technique* which is used to change or extend a process model by modifying it (i.e., *repair*), or by adding a new perspective to this model (i.e., *extension*) [3]. In recent work, van der Aalst already brings together PM with the domain of software engineering. For instance in [10], the authors present a novel reverse engineering technique to obtain real-life event logs from distributed software systems. Thereby, PM techniques are applied to obtain precise and formal models and to monitor and improve processes by performance analysis and conformance checking.

2.2 Unifying Conceptual Architecture

In this section, we combine MDE with PM by presenting a unifying conceptual architecture. The alignment of these two different research fields may help us, e.g., to verify

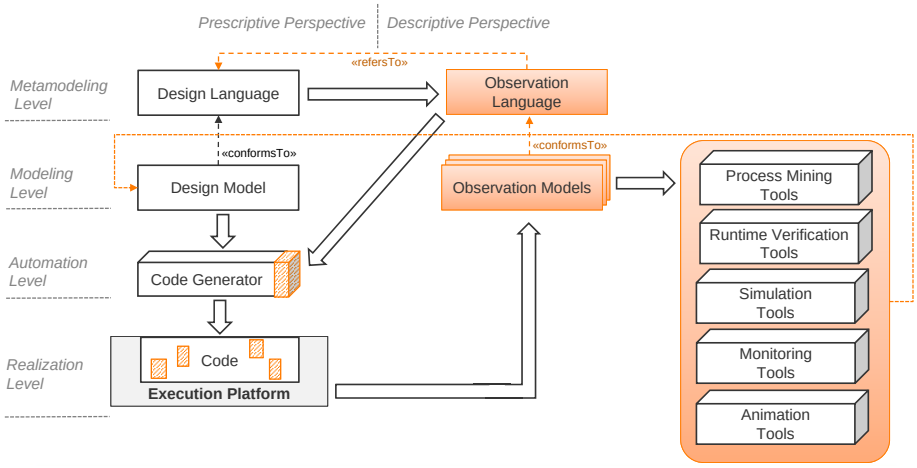

Fig. 1. Unifying conceptual architecture for MDE and PM.

if the mapping feature of design models is really fulfilled, or if important information generated at runtime is actually missing in the design model.

Fig. 1 presents an overview of this architecture. On the left-hand side there is the *prescriptive perspective*, where we use models for creating a system, whereas on the right-hand side there is the *descriptive perspective* where models are extracted from running systems. In the following, we describe Fig. 1 from left to right. The starting point is the *design language specification* at the metamodeling level which defines the syntax as well as the semantics of the language. The design model at the modeling level describes a certain system and conforms to the design language. In our approach, this design model describes two different aspects of the problem or the solution: (i) the *static aspect*, which is used to describe the main ingredients of the modeled entity and their relationships, and (ii) the *dynamic aspect*, which describes the behavior of these ingredients in terms of events and interactions that may occur between them [2]. For the vertical transition from the model level to the realization level (i.e., the process of transforming models into source code), we use *code generation* at the automation level. Finally, at the realization level the running software relies on a specific platform for its execution.

At the right-hand side of Fig. 1 (at the top right), we present a logging metamodel—the so-called *observation language*. This metamodel defines the syntax and semantics of the (event) logs we want to observe from the running system. In particular, we derive this metamodel from the operational semantics of the design language. This means that this observation metamodel can be derived from any modeling language that can be equipped with operational semantics. Fig. 1 indicates this transformation at the metamodel level. The *observation model*, which conforms to the observation language, thumbs the logs at runtime and provides these logs as input for any kind of tools used for checking purposes of non-functional properties (e.g. performance, correctness, ap-

propriateness, etc.). This means that we can transform a language-specific observation model to a workflow file and import it, e.g., in a PM tool as presented in our case study.

3 Case Study: Execution-based Model Profiling

In this section, we perform an empirical explanatory case study based on the guidelines introduced in [14]. The main goal is to evaluate if current approaches for MDE and PM may be combined in a loosely-coupled way, i.e., both can stay as they are initially developed, but provide interfaces to each other to exchange the necessary information to perform automated tasks. In particular, we report on our results concerning a fully model-driven engineered traffic light system which is enhanced with execution-based model profiling capabilities. All artifacts of the case study can be found on our project website.[1]

3.1 Research questions

As mentioned above, we performed this study to evaluate the feasibility and benefits of combining MDE and PM approaches. More specifically, we aimed to answer the following research questions (RQ):

1. *RQ1—Transformability:* Is the operational semantics of the modeling language rich enough to automatically derive observation metamodels?
2. *RQ2—Interoperability:* Are the observation metamodels amenable for PM tools by reusing existing process mining formats?
3. *RQ3—Usefulness:* Are the generated model profiles resulting from the observation model sufficient enough for runtime verification?

3.2 Case Study Design

Requirements. As an appropriate input to this case study, we require a system which is generated by an MDE approach equipped with an executable modeling language, i.e., its syntax and operational semantics are clearly defined and accessible. Furthermore, the approach has to provide translational semantics based on a code generator which may be extended by additional concerns such as logging. Finally, the execution platform hosting the generated code must provide some means to deal with execution logs.

Setup. To fulfill the stated requirements, we selected an existing MDE project concerning the automation controller of a traffic light system. This system consists of several components, e.g., lights (green, yellow, red) for cars and pedestrians, as well as a controller of the system. We modelled this example by using a UML-based design language named *Class/State Charts* (CSC) resulting in a class diagram and a timed state machine as prescriptive models (cf. Fig. 2). While the state machine shows all possible and valid transitions/states within this example, the class `TrafficLightController` specifies the blinkcounter (bc) and the different lights which can be on or off. The models of this example have been developed by using the Embedded Engineer.[2] We use the

[1] `http://www.sysml4industry.org/?page_id=722`
[2] `http://www.lieberlieber.com`

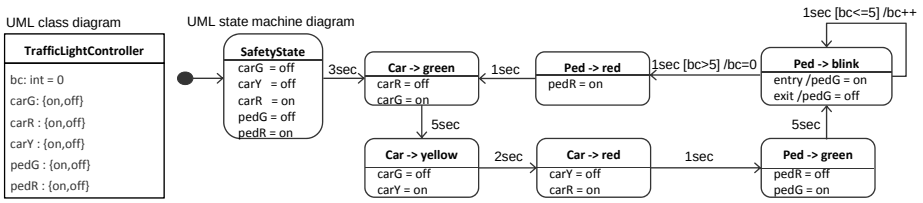

Fig. 2. UML class diagram and state machine of the traffic light system.

Embedded Engineer also to produce Python code from the traffic light model. The code can be executed on a single-board computer. We use as specific execution platform a Raspberry Pi (see Fig. 3, at the bottom left). It has to be noted that we aimed for full code generation by exploiting a model library which allows to directly delegate to the GPIO module (i.e., input/output module) of the Raspberry Pi.

3.3 Results

In this subsection, we present the results of applying the approach presented in Section 2.2 for the given case study setup. First, we show the general architecture to realize execution-based model profiling for the traffic light system example. Subsequently, several details of the realization are presented by focussing on the observation metamodel as well as the usage of an established PM tool.

Technical Realization at a Glance. Our prototypical realization of execution-based model profiling considers the execution logs of a running code model as the experimental frame. Fig. 3 gives an overview of its implementation. We extend the code generator to produce Python code which enables to report logs to a log recording service implemented as `MicroService` provided by an observation model repository. For data exchange between the running system and the log recording service, we use JSON, which means that the JSON data transferred to the `MicroService` is parsed into log entry elements in the repository. We use Neo4EMF[3] to store the execution logs in a NoSQL database for further analysis. In order to be able to use established PM tools, we generate XML files from the recorded logs (i.e., the observation model). For first evaluation purposes, we import this files in PromLite.[4] The use of this PM tool enables us to generate different Petri net (PN) models from the recorded logs. In more detail, we use ATL [20] as transformation tool to transform an observation model to a workflow instance model and import it in ProM to run the PN discoverer.

The Observation Metamodel. According to PM techniques, we consider an *observation model* as an event log with a start and end time registered as a *sequences of transactions* that having already taken place. However, we do not receive event logs

[3] `http://www.neoemf.com`
[4] `http://www.promtools.org/doku.php?id=promlite`

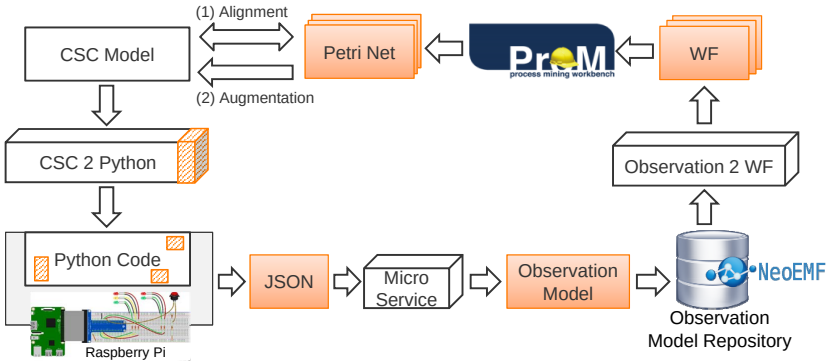

Fig. 3. Deployed traffic light system example.

from an executed process model (i.e., the activities of a business process in an ordered manner), rather we receive the traces from transformed log messages of an executed code model.

Fig. 4 shows the *observation metamodel* derived from the operational semantics of the excerpt of UML which is considered in the context of this case study. It illustrates that changes at runtime (rt) are basically value updates for attributes of the UML class diagram as well as updates concerning the current active state of the UML state machine (cf. Fig. 4, these elements are marked with the ≪rt≫ stereotype). The class `Log` represents a logging session of a certain running software system with a registered `observationStart` and an `observationEnd`. The class `Log` consists of log entries with a unique `id` and a `timeStamp` for ordering purpose (i.e., showing when the entry was recorded). The `LogEntry` either registers a `AttValueChange` or a `CurStateChange`. In case of a `CurStateChange` the `LogEntry` considers the predecessor (`pre`) and successor (`succ`) states. If an attribute has changed the `LogEntry` includes the additional information about the `preValue` and `postValue`.

Generated Model Profiles. For generating different model profiles from the observation model, we employ existing PM tools. For this purpose, we have reverse engineered the XML Schema of the workflow log language[5] into a metamodel, which allows to translate our language-specific observation model into workflow instances by defining different model transformations based on the information which should be discovered. For our case study, we have first implemented a model transformation in ATL which considers the state occurrences of system runs, i.e., it is a view on the observation model just considering the `CurStateChange` elements. By this, we can check if the state machine is realized as intended by the code generator and if it executes on the realization level as specified. In particular, their equivalence can be check semantically by comparing the state space of the state machine with the state space of the observed Petri net or also syntactically by defining bi-directional transformation rules which can be used to check the consistency of two heterogeneous models [23]. Second, we have

[5] `http://www.processmining.org/WorkflowLog.xsd`

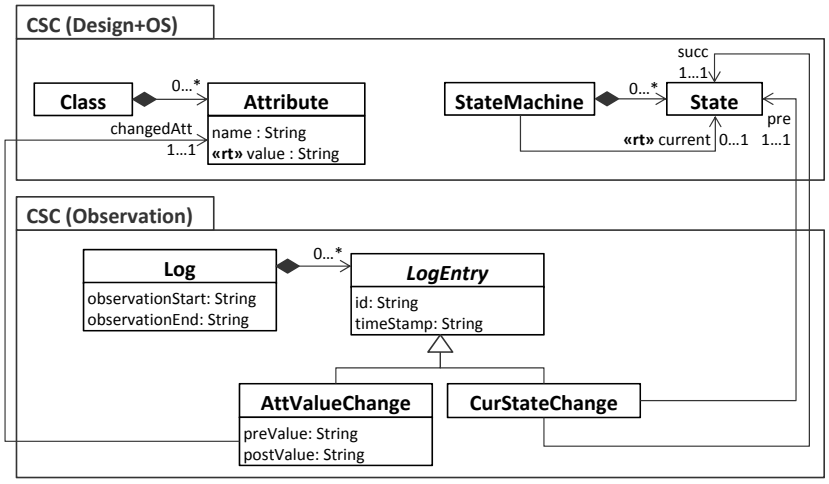

Fig. 4. Observation language for UML class diagram (CD) and state machine (SM).

developed another ATL transformation which extracts for each attribute a workflow instance which contains the sequence of `AttValueChanges`. By this, we can extract the shape of the values stored in the attribute and enrich the model with this kind of information, or check if certain value constraints are fulfilled during execution. For instance, for the blink counter (`bc`) attribute we have derived a PN which explicitly shows a loop counting from zero to six. All the generated artefacts can be found on our project website.

3.4 Interpretation of Results

Answering RQ1. The operational semantics could be transferred into an observational viewpoint. By generating a change class for every element in the design metamodel which is annotated with the $\ll$`rt`$\gg$ stereotype, we are able to provide a language to represent observations of the system execution. This language can be also employed to instrument the code generator in order to produce the necessary logging statements as well as to parse the logs into observation model elements.

Answering RQ2. By developing transformations from the language-specific observation metamodels to the general workflow-oriented formats of existing PM tools, we could reuse existing PM analysis methods for MDE approaches in a flexible manner. Not only the state/transition system resulting from the state machine can be checked between implementation and design, but also other mining tasks could be achieved such as computing value shapes for the given attributes of the class diagram. Thus, we conclude that it is possible to reuse existing formats for translating the observations, however, different transformations may be preferred based on the given scenario.

Answering RQ3. For runtime verification, we took as input transformed event logs (i.e., selected state changes as a workflow file) and employed the α++-algorithm of ProM-Lite 1.1 to derive a PN. This generated PN exactly corresponds to the state machine, as shown in Fig. 2 on the right hand side. We are therefore convinced that the state machine is realized as intended by the code generator. Similarly, we have done this for variable changes. As output we extracted a value shape $[0..6]$ stored in the attribute blink counter. By doing so we demonstrated, that we are also able to enrich the initial class diagram with runtime information in terms of profiled metadata. Finally, we manually implemented random error-handling states in the Python code model (not in the design model) to show that these errors are one-to-one reflected in the generated PN. By applying bi-directional transformations, these additional states may be also propagated to the state machine model in order to complete the specification for error-handling states which are often neglected in design models [5].

3.5 Threats to Validity

To critically reflect our results, we discuss several threats to validity of our study. First, in the current realization of our approach we do not consider the instrumentation overhead which may increase the execution time of the instrumented application. Of course, this may be critical for timed systems and has to be validated further in the future. Second, the current system is running as a single thread which means we are not dealing with concurrency. Extensions for supporting concurrency may result in transforming the strict sequences in partially ordered ones. Third, we assume to have a platform which has network access to send the logs to the micro service. This requirement may be critical in restricted environments and measurements of network traffic have to be done. Finally, concerning the generalizability of the results, we have to emphasize that we currently only investigated one modeling language and one execution platform. Therefore, more experiments are needed to verify if the results can be reproduced for a variety of modeling languages and execution platforms.

4 Related Work

We consider model profiling as a very promising field in MDE and as the natural continuation and unification of different already existing or emerging techniques, e.g., data profiling [7], process mining [3], complex event processing [6], specification mining [5], finite state automata learning [8], as well as knowledge discovery and data mining [9]. All these techniques aim at better understanding the concrete data and events used in or by a system and by focusing on particular aspects of it. For instance, data profiling and mining consider the information stored in databases, while process mining, FSA learning and specification mining focus on chronologically ordered events. Not to forget models@run.time, where runtime information is propagated back to engineering. There are several approaches for runtime monitoring. Blair et al. [22] show the importance of supporting runtime adaptations to extend the use of MDE. The authors propose models that provide abstractions of systems during runtime. Hartmann et al. [21] go one step further. The authors combine the ideas of runtime models with

reactive programming and peer-to-peer distribution. They define runtime models as a stream of model chunks, like it is common in reactive programming.

Currently, there is emerging research work focusing on runtime phenomena, runtime monitoring as well as discussing the differences between descriptive and prescriptive models. For instance, Das et al. [16] combine the use of MDE, run-time monitoring, and animation for the development and analysis of components in real-time embedded systems. The authors envision a unified infrastructure to address specific challenges of real-time embedded systems' design and development. Thereby, they focus on integrated debugging, monitoring, verification, and continuous development activities. Their approach is highly customizable through a context configuration model for supporting these different tasks. Szvetits and Zdun [17] discuss the question if information provided by models can also improve the analysis capabilities of human users. In this context, they conduct a controlled experiment. Heldal et al. [18] report lessons learned from collaborations with three large companies. The authors conclude that it is important to distinguish between descriptive models (used for documentation) and prescriptive models (used for development) to better understand the adoption of modeling in industry. Last but not least, Kühne [19] highlights the differences between explanatory and constructive modeling, which give rise to two almost disjoint modeling universes, each of it based on different, mutually incompatible assumptions, concepts, techniques, and tools.

5 Conclusion and Next Steps

In this paper, we pointed to the gap between design time and runtime in the current MDE approaches. We stressed that there are already well-established techniques considering runtime aspects in the area of PM and that it is beneficial to combine these approaches. Therefore, we presented a unifying conceptual architecture for execution-based model profiling, where we combined MDE and PM. We built this approach upon traditional activities of MDE such as design modeling, code generation, and execution and demonstrated it for the traffic light system case study. While the first results seem promising, there are still several open challenges, which we discussed in the threats to validity subsection of the case study. As next steps we will focus on reproducing our current results with other case studies, e.g., by considering a domain-specific modeling language for an elevator system [1].

Acknowledgment

This work has been supported by the Austrian Science Fund (FWF): P 28519-N31 and by the Austrian Research Promotion Agency (FFG) within the project "InteGra 4.0 - Horizontal and Vertical Interface Integration 4.0".

References

1. Meyers, B., Deshayes, R., Lucio, L., Syriani, E., Vangheluwe, H., Wimmer, M.: ProMoBox: A Framework for Generating Domain-Specific Property Languages. In: SLE (2014)

2. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice. Morgan & Claypool (2012)
3. van der Aalst, W.M.P.: Process Mining: Discovery, Conformance and Enhancement of Business Processes. Springer (2011)
4. van der Aalst, W.M.P.: Process Mining. Commun. ACM, vol. 55, pp. 76–83. (2012)
5. Dallmeier, V., Knopp, N., Mallon, Ch., Fraser, G., Hack, S., Zeller, A.: Automatically Generating Test Cases for Specification Mining. IEEE TSE, vol. 38, pp. 243–257, (2012)
6. Luckham, D.: The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley (2005)
7. Abedjan, Z., Golab, L., Naumann, F.: Profiling relational data: a survey. VLDB, vol. 24, pp. 557–584, (2015)
8. Giles, Lee C., , Miller, B.C., Dong, C., Hsing-Hen, C., Guo-Zeng, S., Yee-Chun, L.: Learning and extracting finite state automata with second-order recurrent neural networks. Neural Computation, vol. 4, pp. 393–405, (1992)
9. Fayyad, Usama M., Piatetsky-Shapiro, G., Smyth, P.: From Data Mining to Knowledge Discovery: An Overview. In: Advances in Knowledge Discovery and Data Mining, pp. 1–34, (1996)
10. van der Aalst, W.M.P., Leemans, M.: Process mining in software systems: Discovering real-life business transactions and process models from distributed systems. In: MoDELS (2014)
11. Rozinat, A., an der Aalst, W.M.P.: Conformance checking of processes based on monitoring real behavior. Inf. Syst., vol. 33, pp. 64–95, (2007)
12. Mazak, A., Wimmer, M.: Towards Liquid Models: An Evolutionary Modeling Approach. In: CBI (2016)
13. de Lara, J., Guerra, E., Cuadrado, J.S.: Model-driven engineering with domain-specific meta-modelling languages. SoSyM, vol. 14, pp. 429–459, (2015)
14. Runeson, P., Höst, M., Sjoberg, D.: Guidelines for conducting and reporting case study research in software engineering. Empirical Software Engineering, pp. 131–164, (2009)
15. Dumas, M., van der Aalst, W.M.P., ter Hofstede, A.H.M.: Process-Aware Information Systems: Bridging People and Software Through Process Technology. Wiley (2005)
16. Das, N., Ganesan, S., Bagherzadeh, J. M., Hili, N., Dingel, J.: Supporting the Model-Driven Development of Real-time Embedded Systems with Run-Time Monitoring and Animation via Highly Customizable Code Generation. In: MoDELS (2016)
17. Szvetits, M., Zdun, U.: Controlled Experiment on the Comprehension of Runtime Phenomena Using Models Created at Design Time. In: MoDELS (2016)
18. Heldal, R., Pelliccione, P., Eliasson, U., Lantz, J., Derehag, J., Whittle, J.: Descriptive vs Prescriptive Models in Industry. In: MoDELS (2016)
19. Kühne, T.: Unifying Explanatory and Constructive Modeling. In: MoDELS (2016)
20. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Sci. Comput. Program. vol. 72, pp. 31–39, (2008)
21. Hartmann, T., Moawad, A., Fouquet, F., Nain, G., Klein, J., Le Traon, Y.: Stream my Models: Reactive Peer-to-Peer Distributed Models@run.time. In: MoDELS (2015)
22. Blair, G., Bencomo, N., France, R.B.: Models@run.time. IEEE Computer, vol. 42, pp. 22–27, (2009)
23. Czarnecki, K., Foster, J.N., Hu, Z., Lämmel, R., Schürr A., Terwilliger, J.F.: Bidirectional Transformations: A Cross-Discipline Perspective. In: ICMT (2009)