# Execution-Based Model Profiling

Alexandra Mazak, Manuel Wimmer, Polina Patsuk-Bösch



# Execution-based Model Profiling

Alexandra Mazak, Manuel Wimmer, and Polina Patsuk-Bösch

Christian Doppler Laboratory for Model-Integrated Smart Production (CDL-MINT)
Institute of Software Technology and Interactive Systems,
TU Wien, Favoritenstrasse 9-11, 1040 Vienna, Austria
{mazak,wimmer,patsuk}@big.tuwien.ac.at
https://cdl-mint.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 (re)design phases. To tackle this limitation, we propose execution-based model profiling as a continuous process to improve prescriptive models at design-time through runtime information. This approach incorporates knowledge in terms of model profiles from execution logs of the running system. To accomplish this, we combine techniques of process mining with runtime models of MDE. In the course of a case study, we make use of a traffic light system example to demonstrate the feasibility and benefits of the introduced execution-based model profiling approach.

## 1 Introduction

In *model-driven engineering (MDE)*, models are put in the center and used as a driver throughout the software development process, finally leading to an automated generation of the software systems [14]. In the current state-of-practice in MDE [3], 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 [3]. Thus, models are used as *prescriptive models* for creating a software system [11]. 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 [23].

According to Brambilla et al. [3] 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 improving design models. As theoretically outlined in [16], 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 aim 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) [23]. 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 of execution-based model profiling conducted on a traffic light system example and present the results. 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 a 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 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 initial model [3]. These transformation engineering aspects are based on the metamodels of the used modeling language, which provide the abstract syntax of that language. This syntax guarantees that models follow a clearly defined structure. In addition, it forms the basis for applying operations on models (e.g., storing, querying, transforming, checking, etc.).

As described in [3], the semantics of a modeling language can be formalized by giving (*i*) *denotational semantics* by defining a mapping from the modeling language to a formal language, (*ii*) *operational semantics* by defining a model simulator (i.e., 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 that a model is executable when its *operational semantics* is fully specified [3]. 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 [17]. 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).** PM combines techniques from data mining and model-driven Business Process Management (BPM) [23]. 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 [8]. This means that PM works on the basis of event data instead of prescriptive models. The main challenge of PM is to capture behavioral aspects. Thereby, specialized algorithms (e.g., the $\alpha$-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 these logs. Generally in PM, event logs are analyzed from a process-oriented perspective using general modeling languages (e.g., UML, Petri nets) [24].

There are three main techniques in PM: (*i*) the *discovery technique* by which a process model can be automatically extracted from log data [23], (*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 [18], and (*iii*) the *enhancement technique* which is used to change or extend a process model by modifying it, or by adding a new perspective to this model [23].

Orthogonal to the dimension of these techniques, there exists a dimension of different perspectives [23]: (*i*) the *control-flow perspective* reflects the ordering of activities, (*ii*) the *organizational perspective* focuses on resources, organisational

units and their interrelations, (*iii*) the *case perspective* deals with properties of individual cases, or process instances, and (*iv*) the *time perspective* focuses on execution time analysis and the frequency of events. These perspectives give a complete picture of the aspects that process mining intends to analyze. In [19], van der Aalst suggests to combine perspectives in order to create simulation models of business processes based on runtime information.

In recent work, van der Aalst already brings together PM with the domain of software engineering. For instance in [25], 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, as well as to monitor and improve processes by performance analysis and conformance checking. In the context of this paper we focus on the control-flow and time perspectives of PM.

### 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 if the mapping feature of design models is really fulfilled, or if important information generated at runtime is actually missing in the design (i.e., prescriptive) model.

Figure 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 (i.e., executed models). In the following, we describe Figure 1 from left to right.

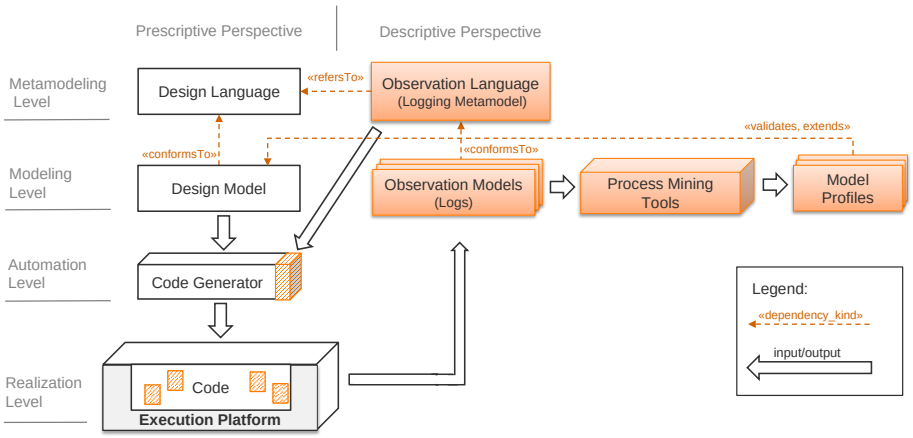


**Fig. 1.** Unifying conceptual architecture for MDE and PM.

The starting point is the *design language specification* at the metamodeling level which defines the syntax as well as semantics of a language like UML, SysML, or a certain domain specific language (DSML). The *design model* at the modeling level describes a certain system for a specific purpose and has to conform to the chosen design language (see Figure 1, «conformsTo»). In our approach, such a model describes two different aspects of the system: (*i*) the *static aspect* which describes the main ingredients of the domain to be modeled, i.e., its entities and their relationships, and (*ii*) the *dynamic aspect* which describes the behavior of these ingredients in terms of events and interactions that may occur among them. For the vertical transition from the modeling level to the realization level (i.e., the process of transforming models into source code), we use *code generation* at the automation level as introduced in [3]. Finally, at the realization level, the running software relies on a specific platform for its execution (e.g., a Raspberry Pi as presented in our case study in Section 3).

At the right-hand side of Figure 1 (at the top right), we present a logging metamodel—the so-called *observation language*. This metamodel defines the syntax and semantics of the 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 the observation metamodel can be derived from any modeling language that can be equipped with operational semantics. Figure 1 indicates this dependency at the metamodel level by the dashed arrow and the keyword «refersTo». The observation language has an influence on the code generator, which produces not only the code for the system to run, but also logging information (see Figure 1, arrow from the observation language (input) to the code generator (output)). This means that the observation language determines which runtime changes should be logged and the code generator provides the appropriate logging code after every change (e.g., state change, attribute value change). Finally, these execution logs are stored as so-called *observation models* (see Figure 1, arrow from the execution platform to the observation models). These observation models, which conform to the observation language, thumb the logs at runtime and provide these logs as input for any kind of tools used for checking purposes, e.g., for checking non-functional properties like performance, correctness, appropriateness. For instance, we transform the design language-specific observation model to a workflow representation which can be read by PM analysis tool as presented in our case study.

## 3 Case Study: Execution-based Model Profiling

In this section, we perform an exploratory case study based on the guidelines introduced in [20]. 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 explanatory research questions (RQ) composed of two requirement satisfaction questions (Transformability, Interoperability), an effect question (Usefulness), and a trade-off question (Timeliness):

1. *RQ1—Transformability:* Is the operational semantics of the modeling language rich enough to automatically derive observation metamodels?
2. *RQ2—Interoperability:* Do observation metamodels satisfy interoperability by fulfilling the requirements of existing process mining formats?
3. *RQ3—Verifiability:* Are the generated model profiles resulting from the observation model sufficient for runtime verification?
4. *RQ4—Timeliness:* Are there significant differences between timing of transitions on the specification level and the implementation level?

### 3.2 Case Study Design

**Requirements.** As an appropriate input to this case study, we require a system which is generated by a MDE approach and equipped with an executable modeling language. This means that 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 these case study requirements, we selected an existing MDE project concerning the automation controller of a traffic light system. We modeled this example by using a small sub-set of UML which we named *Class/State Charts (CSC)* language. CSC stands for UML class diagram and UML state machine diagram, both shown in Figure 2. The class diagram represents the static aspect of the system, whereas the state machine diagram describes the dynamic one. Generally, UML class diagrams consist of *classes* with *attributes*, and state charts containing *state machines* with *states* and *transitions* between them [21]. In a state chart diagram *transitions* can be triggered by different types of *events* like signal event, time event, call event, or change event [21]. Both, states and transitions can call *actions*.

Figure 2 presents the class diagram and state machine diagram of the traffic light system modeled in CSC. This system consists of several components such as

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

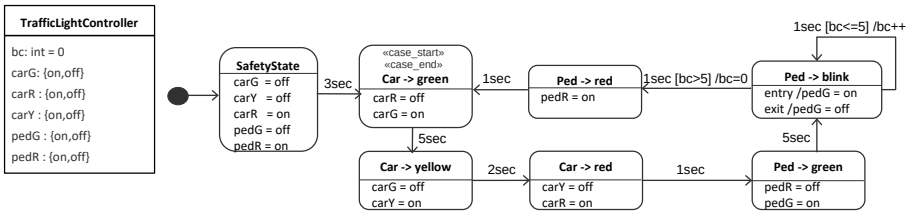


**Fig. 2.** CSC class diagram and state machine diagram of the traffic light system.

lights (green, yellow, red) for cars and pedestrians, a controller as well as a blink counter for the pedestrian light. While the CSC state machine diagram (see Figure 2, on the right-hand side) shows all possible and valid transitions/states within this example, the CSC class `TrafficLightController` (see Figure 2, on the left-hand side) specifies the blink counter `bc:int=0` and the different lights which can be on or off.

We employed the Enterprise Architect[2] (EA) tool to model the CSC class and state machine diagram. Additionally, we used and extended the Vanilla Source plug-in of EA to generate Python code from the executed CSC (design) models. The code can be executed on a single-board computer. For this purpose we used Raspberry Pi (see Figure 3, at the bottom left) as specific execution platform. 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. Firstly, we describe the technical realization of the example. Subsequently, we present the appropriate observation metamodel referring to the CSC design language and its conforming observation model. Finally, we generate different model profiles on the basis of PM techniques for checking purposes.

**Technical Realization at a Glance.** The execution logs of the running code on the Raspberry Pi form the basis for the experimental frame of our approach. Figure 3 gives an overview of its implementation. We extend the code generator to produce Python code (`CSC2Python`) which enables us 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 used JSON. This means that the JSON data transferred to the `MicroService` is parsed into log entry elements in the repository. We

[2] `http://www.lieberlieber.com`

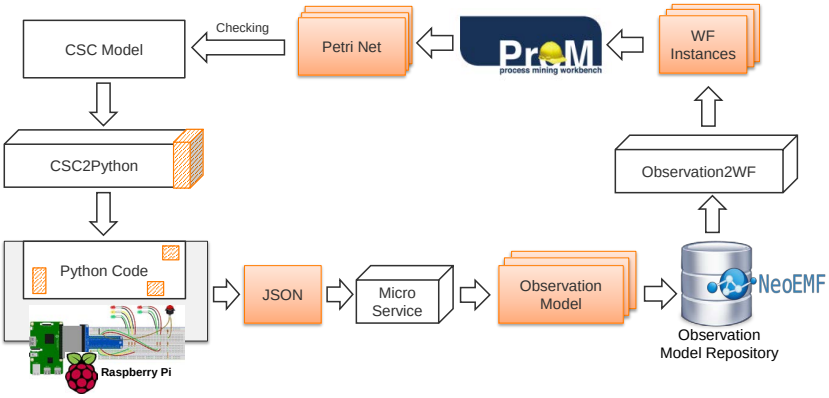


**Fig. 3.** Technical realization of the traffic light system example.

used the NoSQL database Neo4EMF[3] to store the execution logs for further analysis. To be able to use established PM tools, we generated XML files from the recorded execution logs (i.e., the observation models).

For the case study of our approach we used ProM Lite 1.1[4] which is an open source PM tool. Files that this tool takes as input have to correspond to the XSD-schema of the workflow log language MXML[5]. To accomplish this we used the *ATLAS transformation language (ATL)* [12] for transforming the observation models to MXML-conform XML files (`Observation2WF`). In particular, we reverse-engineered the XML Schema of the MXML language into a metamodel. This step enabled us to translate the language-specific observation model into workflow instances (`WF Instances`) to directly import these instances in ProM Lite. For our case study example the used MXML format was sufficient. Nevertheless XES is the current standard, therefore, we will build on the XES format in future work.

**The CSC 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 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 embedded system. Figure 4 shows the *observation metamodel* derived from the operational semantics of the CSC design language used in the context of this case study. The figure illustrates that changes at runtime are basically value updates for attributes of the CSC class diagram as well as updates concerning the current active state and current fired transition of the CSC state machine diagram.

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

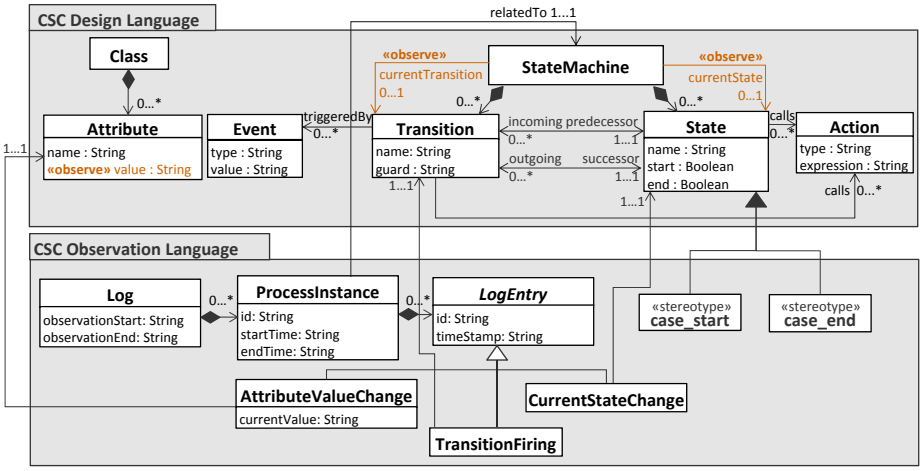


**Fig. 4.** Observation language for the CSC class diagram and CSC state machine diagram of the traffic light example.

As shown in the upper section of Figure 4, these elements are marked with the «observe» stereotype. The CSC dependent observation metamodel is shown in the lower section of Figure 4. 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 process instances related to the CSC `StateMachine`. Every `ProcessInstance` has a unique `id`, `startTime`, and `endTime` attributes and consists of log entries with the attributes `id` and `timeStamp` for ordering purpose (i.e., indicating when the entry was recorded).

Additionally, we defined a subset of a state machine by indicating the stereotypes «case_start» and «case_end». These stereotypes have to be annotated in the design model whenever objects may execute more than one case. The reason for such a stereotype annotation is that, in contrast to business processes, state machines do not necessarily have a clearly defined start- and end point, like in the case of our traffic light system example. This is due to the fact that state machines are often defined for long-life (persistent) objects. This means that only values of objects change over time, but not the objects themselves. Therefore, we defined these stereotypes in our metamodel which enables us to capture single cycles (like cases in PM) of the state machine to be profiled. In our case study example, the start point and end point coincide. When the example starts, their is a safety state only entered once. Each further cycle starts and ends with the state `Car→green` (see Figure 2).

The `LogEntry` either registers an `AttributeValueChange`, a `CurrentStateChange`, or a `TransitionFiring`. `CurrentStateChange` and `TransitionFiring` are associated with the state and the transition of the CSC design language. `AttributeValueChange` has an association with the changing attribute of a class and includes its `currentValue`.

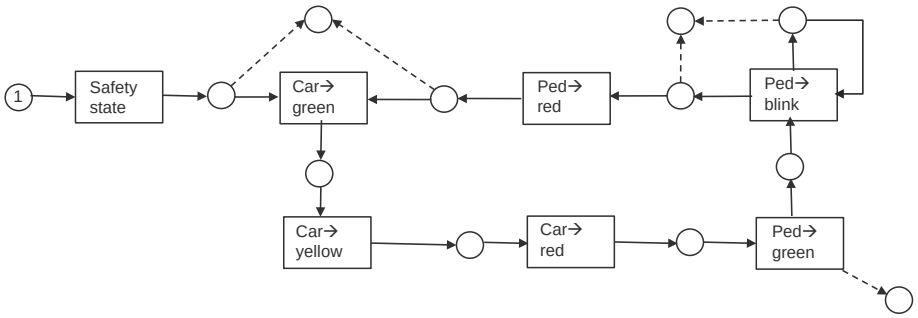


**Fig. 5.** Model profile of state changes.

**Generated Model Profiles.** We used ProM Lite for generating different *model profiles* from the observation model of the running code. For this purpose we employed ATL model transformations to import the CSC language-specific observation model as input into ProM Lite. By doing so, we focused on two PM-perspectives, (*i*) the control-flow perspective and (*ii*) the time perspective (cf. Section 2), as well as a (*iii*) data manipulation one. In the control-flow perspective, we employed the *α++-algorithm* of ProM Lite to generate Petri nets for reflecting all attribute value changes as well as state changes and their structure. For profiling the time perspective, we mined the sequence of fired transitions among all states with the *inductive miner* of ProM Lite and replayed the logs on the discovered Petri net by using a special performance plug-in of this tool.

In a first step of our case study, we implemented a model transformation in ATL which considered the state occurrences (`CurrentStateChange`) of the running system. By this, we checked on the one hand if the CSC state machine diagram is realized by the code generator as intended (see Figure 5), and on the other hand, if the state machine executes the specified control-flow on the realization level. This enables, both, a semantically as well as syntactically "equivalence" checking of the prescriptive (design) model and the descriptive (operational) model. In particular, for semantically checking we compared the state space of the state machine with the state space of the profiled Petri net. As shown in Figure 5 (see the dashed arrows) places with the same targets were merged. The dashed arrow at the bottom right symbolizes a manually interruption of a case. The figure shows that the places and transitions of the Petri net are equivalent to the states and transitions of the CSC state machine diagram presented in Figure 2. For syntactically checking purpose we may define bi-directional transformation rules to check the consistency [5].

In a second step, we implemented a Python component in order to simulate random system failures which were not reflected in the initial design model presented in Figure 2. We observed the control-flow perspective of this extended system and found out that the randomly simulated failure states were correctly detected by ProM Lite (compare the Petri net shown in Figure 6 with that one

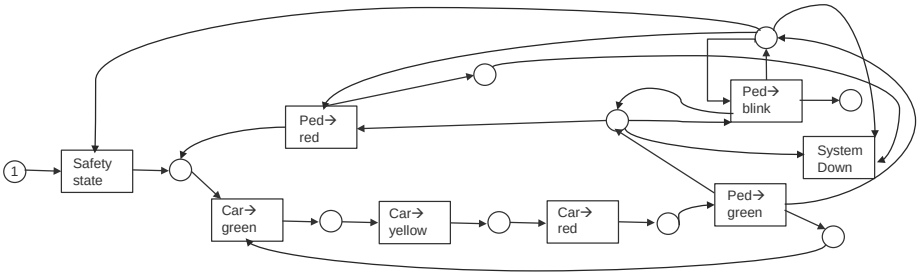


**Fig. 6.** Model profile of state changes including a failure state.

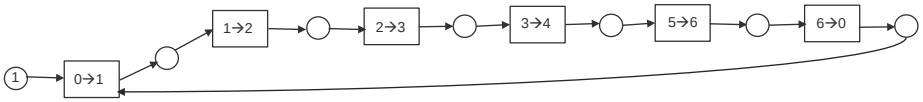


**Fig. 7.** Model profile of the attribute value changes for the blink counter (`bc`).

of Figure 5). Thereby, we proof the usefulness of the approach for runtime verification. It shows that failures which may happened in the implementation phase would be correctly detected and visualized. For instance, this provides useful insights in the running system for validating the code generator and manual code changes.

In a next step, we developed another ATL transformation to extract for each attribute a workflow instance that contains the sequence of `AttributeValueChanges`s. By this, we extracted the shape of the values stored in the attribute to enrich the model with this kind of information and to check if certain value constraints were fulfilled during execution. For instance for the blink counter attribute, we derived a profile which explicitly shows a loop counting from zero to six as depicted in Figure 7. These logged value changes conform to the attribute (`bc`) of the class `TrafficLightController` as shown at the left hand sight of Figure 2.

In the CSC state machine diagram the timing component is explicitly assigned to transitions (see Figure 2, «case_start» and «case_end»). In a last step of our case study, we observe the time perspective. Therefore, we needed an additional ATL transformation for filtering the sequence of `TransitionFiring`s (see Figure 4 from the upper section to the lower section). This sequence includes several iterations of the traffic light system and is used as an input for the performance plug-in of ProM Lite. Our simulation covered 78 cycles, which took 22, 26 minutes, and computed descriptive statistical values for performance evaluation like minimum, maximum and average transition time and sojourn time (i.e., waiting time), as well as the throughput which is the maximum rate

| Selected elements: **Car→yellow to Car→red** | | | | | |
|---|---|---|---|---|---|
| **Timing_property** | **Min** | **Max** | **Avg** | **Std.Dev** | **Freq** |
| Throughput_time | 0.00 ms | 0.00 ms | 0.00 ms | 0.00 ms | 78 |
| Waiting_time | 2.02 sec | 2.12 sec | 2.04sec | 19.24 ms | 78 |
| Sojourn_time | 2.02 sec | 2.12 sec | 2.04 sec | 19.24 ms | 78 |
| Observation_period | 22.26 min | | | | |

**Table 1.** Outcome of the performance evaluation based on transition firings.

at which a system can be processed. Table 3.3 presents the outcome of this descriptive analysis. To count several cycles (i.e., cases), we annotated the state `Car→green` with the stereotypes «case_start» and «case_end» as introduced in the CSC metamodel. On average the transition from car yellow to car red is 2,04 seconds, which is very close to the timing of transition (2sec) of the CSC state machine presented in Figure 2.

### 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 CSC design metamodel which is annotated with the ≪`observe`≫ 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 ATL 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 may be achieved such as computing value shapes for the given attributes of the CSC 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 $\alpha$++-algorithm of ProM Lite to derive a Petri net. This generated Petri net, as shown in Figure 5, exactly corresponds to the state machine, as shown in Figure 2 on the right hand side. We are therefore convinced that the state machine is realized by the code generator as intended. Similarly, we have done

this for attribute value changes. As output we extracted a value shape [0..6] stored in the attribute blink counter (see Figure 7). Thus, we are also able to enrich the initial CSC class diagram presented in Figure 2 with runtime information in terms of model profiles. Finally, we manually implemented random failure states in the Python code (not in the design model) in order to show that these system down states are reflected in the generated Petri net. By applying bi-directional transformations, these additional states may be also propagated to the initial CSC state machine diagram (i.e., prescriptive model) for completing the specification for error-handling states that are often neglected in design models [6].

*Answering RQ4.* For the detection of timing inconsistencies we filtered the sequence of transitions using an ATL transformation and analyzed it with the performance plug-in of ProM Lite. The inconsistencies between the specification and implementation levels are within the range of milliseconds. The average values of the delays can be propagated back to the design model in order to make the timing more precise during the system execution. The information about timing inconsistencies is especially relevant for time critical and safety critical systems, since this information may mitigate potential consequences of delays. However, it is important to observe a system for a sufficiently long period of time to have enough runtime information for reliable statistical values.

### 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 a single modeling language and a single 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 [1], process mining [23], complex event processing [15], specification mining [6], finite state automata learning [2], 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. [4] 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. [10] 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. [7] 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 [22] 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. Van der Aalst et al. [19] show the possibilty to use runtime information and automatically construct simulation models based on event logs. These simulation models can be used, e.g., to evaluate performance of different alternative designs prior to roll-out. Heldal et al. [11] 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 [13] 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 Future Work

In this paper, we pointed to the gap between design time and runtime in 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 the approach upon traditional activities of MDE such as design modeling, code generation, and code execution. In the conducted case study, we demonstrated and evaluated this approach on the basis of a traffic light system example. While the first results seem promising, there are still several open challenges, which we discussed in the threats to validity in the case study section. As next steps, we

will focus on the observation of further PM perspectives (e.g., the organisational perspective) that can be used for software component communication discovery and on the reproduction of our current results by conduction additional case studies, in this respect, domain-specific modeling languages (DSMLs) would be of special interest.

## Acknowledgment

The authors are affiliated with the Christian Doppler Laboratory for Model-Integrated Smart Production (CDL-MINT) at TU Wien, funded by the Austrian Federal Ministry of Science, Research, and Economy (BMWFW) and the National Foundation of Research, Technology and Development (CDG). Furthermore, the authors would thank LieberLieber Software GmbH for the provisioning of the traffic light example.

## References

1. Abedjan, Z., Golab, L., Naumann, F.: Profiling relational data: a survey. VLDB, vol. 24, pp. 557–584, (2015)
2. 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)
3. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice. Morgan & Claypool (2012)
4. Blair, G., Bencomo, N., France, R.B.: Models@run.time. IEEE Computer, vol. 42, pp. 22–27, (2009)
5. Czarnecki, K., Foster, J.N., Hu, Z., Lämmel, R., Schürr A., Terwilliger, J.F.: Bidirectional Transformations: A Cross-Discipline Perspective. In: ICMT (2009)
6. 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)
7. 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)
8. 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)
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. 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)
11. Heldal, R., Pelliccione, P., Eliasson, U., Lantz, J., Derehag, J., Whittle, J.: Descriptive vs Prescriptive Models in Industry. In: MoDELS (2016)
12. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Sci. Comput. Program. vol. 72, pp. 31–39, (2008)

13. Kühne, T.: Unifying Explanatory and Constructive Modeling. In: MoDELS (2016)
14. de Lara, J., Guerra, E., Cuadrado, J.S.: Model-driven engineering with domain-specific meta-modelling languages. SoSyM, vol. 14, pp. 429–459, (2015)
15. Luckham, D.: The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems. Addison-Wesley (2005)
16. Mazak, A., Wimmer, M.: Towards Liquid Models: An Evolutionary Modeling Approach. In: CBI (2016)
17. Meyers, B., Deshayes, R., Lucio, L., Syriani, E., Vangheluwe, H., Wimmer, M.: ProMoBox: A Framework for Generating Domain-Specific Property Languages. In: SLE (2014)
18. 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)
19. Rozinat, A. and Mans, R. S. and Song, M. and van der Aalst, W. M. P.: Discovering Simulation Models. In: Inf. Syst. (2009)
20. Runeson, P., Höst, M., Sjoberg, D.: Guidelines for conducting and reporting case study research in software engineering. Empirical Software Engineering, pp. 131–164, (2009)
21. Seidl, M. and Scholz, M. and Huemer, Ch. and Kappel, G.: UML classroom. An Introduction to Object-Oriented Modeling. Springer (2012)
22. Szvetits, M., Zdun, U.: Controlled Experiment on the Comprehension of Runtime Phenomena Using Models Created at Design Time. In: MoDELS (2016)
23. van der Aalst, W.M.P.: Process Mining: Discovery, Conformance and Enhancement of Business Processes. Springer (2011)
24. van der Aalst, W.M.P.: Process Mining. Commun. ACM, vol. 55, pp. 76–83. (2012)
25. 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)