TU WIEN Informatics

A Framework for Execution-based Model Profiling

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Business Informatics

eingereicht von

Polina Patsuk-Bösch, BSc.
Matrikelnummer 01029574

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.-Prof. Mag. Dr. Manuel Wimmer
Mitwirkung: Dipl.-Ing. Mag. Dr.techn. Alexandra Mazak-Huemer, Bakk.techn.

Wien, 10. Jänner 2020

Polina Patsuk-Bösch	Manuel Wimmer

TU WIEN Informatics

A Framework for Execution-based Model Profiling

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Business Informatics

by

Polina Patsuk-Bösch, BSc.
Registration Number 01029574

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.-Prof. Mag. Dr. Manuel Wimmer
Assistance: Dipl.-Ing. Mag. Dr.techn. Alexandra Mazak-Huemer, Bakk.techn.

Vienna, 10th January, 2020

Polina Patsuk-Bösch	Manuel Wimmer

Erklärung zur Verfassung der Arbeit

Polina Patsuk-Bösch, BSc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 10. Jänner 2020

Polina Patsuk-Bösch

Acknowledgements

I would like to thank my advisors Dr. Manuel Wimmer and Dr. Alexandra Mazak-Huemer for your guidance, inspiration, insights, and patience. I would also like to thank my husband Christoph, it would not be possible without your relentless support and encouragement.

Kurzfassung

Model-Driven-Engineering (MDE) fokusiert sich auf Modelle, die über den gesamten Software-Entwicklungsprozess auf präskriptive Art eingesetzt werden. Obwohl diese präskriptiven Modelle während der Systemimplementierung von Bedeutung sind, liefern deskriptive, aus Laufzeitdaten abgeleitete Modelle wertvolle Informationen in späteren Phasen des Systemlebenszyklus. Bisher wurden solche deskriptiven Modelle im Gebiet des MDE kaum erforscht. Aktuelle MDE-Ansätze vernachlässigen meist die Möglichkeit ein existierendes und laufendes System anhand des Informationsflusses von Betrieb zu Design zu beschreiben. Um eine Verbindung zwischen präskriptiven und deskriptiven Modellen herzustellen, schlagen wir ein vereinheitlichendes Framework vor, in dem MDE-Ansätze und Techniken aus Process-Mining (PM) lose gekoppelt zum Einsatz kommen. Dieses Framework nutzt ausführungsbasiertes Model-Profiling als kontinuierlichen Prozess zur Verbesserung präskriptiver Modelle zur Design-Zeit durch die Nutzung von Laufzeitinformation. Weiter legen wir eine Evaluierungsfallstudie vor, um die Umsetzbarkeit und die Vorteile des vorgeschlagenen Ansatzes zu demonstrieren. In dieser Fallstudie implementieren wir einen Prototypen unseres Frameworks um Logs eines laufenden Systems aufzuzeichnen. Der implementierte Prototyp transformiert die aufgezeichneten Logs in das XES-Format um eine Verarbeitung und Analyse durch PM-Algorithmen zu ermöglichen. Wir zeigen, dass die resultierenden Modelprofile für eine Laufzeitverifikation ausreichend sind. Darüber hinaus demonstrieren wir die Möglichkeit durch das vereinheitlichende Framework solche Modelprofile für verschiedene Perspektiven einzusetzen, darunter Funktionalität, Performanz und Wechselwirkungen zwischen Komponenten.

Schlagworte: Process Mining, Model-Driven Engineering, Model Profiling

Abstract

In Model-Driven Engineering (MDE) models are put in the center and used throughout the software development process in prescriptive ways. Although these prescriptive models are important during system implementation, descriptive models derived from runtime data offer valuable information in later phases of the system life cycle. Unfortunately, such descriptive models are only marginally explored in the field of MDE. Current MDE approaches mostly neglect the possibility to describe an existing and operating system using the information upstream from operations to design. To create a link between prescriptive and descriptive models, we propose a unifying framework for a combined but loosely-coupled usage of MDE approaches and process mining (PM) techniques. This framework embodies the execution-based model profiling as a continuous process to improve prescriptive models at design-time through runtime information. We provide an evaluation case study in order to demonstrate the feasibility and benefits of the introduced approach. In this case study we implement a prototype of our framework to register logs from a running system. The implemented prototype transforms the registered logs into XES-format for further processing and analysis via PM algorithms. We prove that the resulting model profiles are sufficient enough for runtime verification. Furthermore, we demonstrate the possibility to maintain model profiles for multiple concerns, such as functionality, performance and components interrelations, through the unifying framework.

Keywords: process mining, model-driven engineering, model profiling

Contents

Kurzfassung **ix**

Abstract **xi**

Contents **xiii**

1 Overview **1**
1.1 Introduction and Motivation 1
1.2 Problem Definition and Relevance 2
1.3 Aim of the Work 4
1.4 Methodology 5
1.5 Structure of the Thesis 5

2 State Of The Art **7**
2.1 Model-Driven Engineering 7
2.2 Process Mining 12
2.3 Models@run.time 24

3 Execution-based Model Profiling **29**
3.1 Overview 29
3.2 Approach 30
3.3 Unifying Framework 31

4 Technical Realization **35**
4.1 Overview 35
4.2 Modeling Tool and Code Generator 36
4.3 Logging Instrumentation 37
4.4 Execution Platform 38
4.5 MicroService 39
4.6 Transformations 40
4.7 Process Mining Tool 46

5 Evaluation: Case Study **49**
5.1 Overview 49

5.2 Design 49
5.3 Results 57
5.4 Interpretation of Results 72
5.5 Threats to Validity and Limitations 75

6 Related Work **77**

7 Summary and Future Work **79**

List of Figures **81**

Acronyms **85**

Bibliography **87**

Appendices **95**

CHAPTER 1

Overview

1.1 Introduction and Motivation

Short cycles of innovation, fast changes in customer needs, continuous adjustments in legislation are just a few reasons for ever increasing importance of flexibility in the industrial automation domain [LSVH14]. The recent emergence of the term Cyber-Physical System (CPS) emphasizes that both hardware and software are equally important to realize modern production systems [LBK14, VHLL15]. Apart from that several new challenges arise for production companies with the transition to the new industry paradigm, Industry 4.0. In this paradigm automated engineering solutions are produced by other automated solutions [Bau17].

In particular, two main demands can be specified. The first one is early validation of design decisions via simulation runs. The second one is the back-propagation of data from operations to design in order to enhance and adapt the initial design of a production system. A way to solve those issues is to combine various methods and techniques into an interoperable toolchain from engineering and design, over simulation, to operations and monitoring of different versions of productive systems in order to gather massive sets of runtime data for further analysis. These toolchains and data sets should finally lead to a systematic creation of digital twins - one of the enabler concepts of Industry 4.0 [Bau17].

Since industrial engineering is confronted with these emerging requirements, the software engineering domain is inevitably challenged with them as well. Model-Driven Engineering (MDE) is a promising approach to deal with the increasing complexity of modern software systems in the industrial automation domain [BGS+14, BCW12, KMR00]. In this paradigm software is created on a higher abstraction level. Thus, MDE facilitates industrial engineers to work with software models of embedded automation systems, modify them using modeling tools and single-handedly generate and deploy code for early design validation. However, code centric development still takes the leading role in

software engineering and the advantages of modeling are mostly utilized only for code generation [Vya13]. Nevertheless, MDE offers additional benefits, such as dynamical extraction of runtime models and cooperative simulation. In this thesis we focus on the first point, i.e., exploring runtime phenomena observed from MDE-based systems during their execution.

Modern complex automated and embedded systems produce significant amount of data as they run. This includes registration and logging of components communication within the system, as well as handling of external contexts and environmental inputs. As example of internal component communication to emphasize its possible volume we can point out, e.g., modern cars with one hundred electronic control units which regulate the car's braking systems, the vehicle speed, the speed of each wheel, the drive mode, etc. [DGJ+16]. Additionally, in the near future, the interconnection level of stand-alone systems is expected to increase rapidly multiplying the runtime data they produce. For instance, these already complex cars will dynamically connect to each other to implement functionalities like automated cross roads assistants [AGJ+14]. In the agricultural domain autonomously driving tractors are already today able to perform organised tasks and connect to each other [Fen11]. Furthermore, such interconnected systems have to deal with and react to various environmental inputs at runtime. In consequence, taking into account the level of complexity, both the structure and the behavior of the embedded automation systems have to be observed at runtime, since they can not be entirely predicted at design time. The runtime communication data that such systems produce, if being properly logged, offers plenty of unexplored but valuable information and experiences.

This thesis was written based on the research conducted in Christian Doppler Laboratory[1], modul "Reactive Model Repositories", in collaboration with the company LieberLieber[2] [MWPB18].

1.2 Problem Definition and Relevance

In the current state-of-practice in MDE [BCW12], 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, i.e., a model is never complete and only created as a blueprint of a system. Therefore, models at design time are early system snapshots, which are used in a prescriptive manner in order to automatically generate model-driven engineered software systems [BCW12, HPE+16].

While prescriptive or design models are indeed a very important ingredient to realize a system, for later phases in the system's life cycle additional model types are needed.

[1] Christian Doppler Labor for Model-Integrated Smart Production (CDL-MINT): https://cdl-mint.big.tuwien.ac.at/

[2] LieberLieber: https://www.lieberlieber.com/en/home-en/

In industrial automation engineers typically have the desirable behavior in mind when creating a system, but they are not aware in these early phases of the many deviations that may take place at runtime [vdA16]. 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 descriptive models are only marginally 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. Thus, both type of models should be included into the full system life cycle, since having only an a-priori description of a system is not enough. For fulfilling this requirement, models should not be considered as isolated static system prescriptions, but as evolutionary descriptions. This can be realized in a modeling approach combining model-driven design and runtime phenomena. Therefore, as theoretically outlined in [MW16], model profiling can be considered 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 document the evolution of these models, and (iii) to better anticipate the unforeseen.

There exist already promising techniques to focus on runtime phenomena, especially in the area of Process Mining (PM) [vdA16]. PM combines techniques from data mining and Business Process Management (BPM). 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 [DvdAtH05a]. This means that PM works on the basis of event data instead of designed models and the main challenge here is to capture behavioral aspects of a running systems. Thereby, specialized algorithms can convert runtime data into a Petri net, which forms 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. However, PM alone doesn't take the design models into consideration and doesn't have any semantic links to them.

In this thesis we focus on creating a link between downstream information derived from prescriptive models and upstream information in terms of descriptive models. The research objective of the thesis is to develop a unifying framework realizing a model profiling approach for a combined but loosely-coupled usage of MDE and PM techniques. In order to reach the research objective the following research questions should be answered:

1. What concepts, techniques, and tools can be used to to realize a model profiling approach?

2. Can the operational data be automatically stored as descriptive models derived from the operational semantics of the design language?

3. Are the resulting model profiles sufficient enough to verify system's behavior at runtime?

4. Is it possible to maintain model profiles for multiple concerns, such as functionality, performance, and components interrelations, through unifying framework?

1.3 Aim of the Work

The expected outcome of this thesis is a unifying framework for a combined but loosely-coupled usage of MDE approaches and PM techniques. The framework will be developed in the context of a case study with prototype implementation within an experimental frame. This case study is based on a traffic light system, engineered via a model-driven approach. The model for this example has been developed by using the modeling tool Enterprise Architect (EA)[3]. The extension *VanillaSource* for EA provided by the company LieberLieber[4] is used to automatically produce Python code from the design model. The code can be executed on a single-board computer, e.g., Raspberry Pi[5]. During the execution, the traffic light system produces logging output, which is used for further analysis by PM techniques.

The expected outcome is to be reached in several steps, as follows:

In the first step, the initial model-driven engineered traffic light system is analyzed to determine potential drawbacks and inconsistencies. The result of this step is a new model-driven engineered system improved according to the outcome of the analysis.

In the next step, both the initial and the improved systems are generated from their design models and then deployed on hardware to simulate real systems and collect event logs. The results of this step are data sets of the recorded logs ready to be processed by PM tools.

In the third step, PM techniques are applied to the data set of the recorded logs. The result of this step is a re-created design model in form of a Petri net which might differ from the initial design model. Therefore the recreated model is compared to the initial design model in order to find discrepancies and to locate their sources.

In the last step, the results of the previous steps are interpreted and evaluated according to the case study design. The architecture of the whole system including the MDE-part and the PM-part is described. The results of these steps are prototype artifacts and a documented unifying framework.

[3] http://www.sparxsystems.com/

[4] Embedded Engineer and Python with Enterprise Architect: https://blog.lieberlieber.com/2015/06/02/embedded-engineer-and-python/

[5] Raspberry Pi Foundation: https://www.raspberrypi.org/

1.4 Methodology

The general approach to the research is an explanatory case study based on the guidelines for conducting empirical explanatory case studies proposed by [RH09]. Such an observational study provides deeper understanding of the phenomena under study and their interrelations. This approach allows to evaluate the feasibility of combining MDE and PM techniques.

Applied methodologies:

- **Requirements Elicitation**
  - **Literature review**. State of the art should be summarized to outline the background of the related topics: MDE including the important concepts of modeling languages, metamodeling and model transformations, PM including an overview of existing algorithms, models@run.time (M@RT), etc.
- **Realization**
  - **Case Study Design**. Definition of objectives, requirements and setup: modeling language, tools used, interaction between the tools, data-formats, etc.
  - **Developing a prototype**. MDE-based realization of the prototype within an experimental frame including creation of an automated workflow from a running MDE-system to the PM-analyzed logs of this system. This method also includes refactoring of the initial model for a variety of experiments.
- **Evaluation: Case-Study-based**
  - **Qualitative data collection**. Deploying and execution of the prototype on the hardware and collection of the event logs for further analysis.
  - **Analysis of collected data**. Analysis of the recorded event logs using PM techniques: discovery of the underlying processes using existing algorithms.
  - **Interpretation of the results and hypothesis confirmation**. Summarizing the research results and answering the research questions.

1.5 Structure of the Thesis

Chapter 1 gives an introduction of the thesis, including problem definition, aim of the work and methodology. In Chapter 2 two central concepts are introduced, i.e., MDE and PM. Theoretical fundamentals and principles of both topics relevant for this thesis are given in the Sections 2.1 and 2.2. Section 2.3 addresses the emerging research work focusing on runtime phenomena, namely M@RT. Chapter 3 presents our approach combining models at design time and runtime. This approach integrates MDE and PM techniques in a unifying framework for execution-based model profiling at runtime. Chapter 4 describes

the technical realization of the approach within an experimental frame. The evaluation of the approach with a case study and the results of the conducted experiments are presented in Chapter 5. Chapter 6 gives an overview of the related work similar to our approach. Finally, Chapter 7 summarizes the results of the thesis, discusses limitations, future work and possible next steps.

CHAPTER 2

State Of The Art

2.1 Model-Driven Engineering

2.1.1 Overview

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 executable software systems [dLGC15]. The primary goal of MDE is to get running systems out of models.

The main aspects considered in MDE are shown in Figure 2.1. MDE is presented in two dimensions: *implementation* (rows) and *conceptualization* (columns). The dimension of implementation shows the way from model definition to a running system through mapping. There are three levels in this dimension, which are as follows [BCW12]:

- The *modeling level* considers definitions of models and corresponding modeling languages.
- The *realization level* considers implementation artifacts and platforms where those artifacts are used (e.g. executable code in case of software).
- The *automation level* allows mapping from the modeling level to the realization level.

The conceptualization dimension describes the process of defining conceptual models on different levels of abstraction. This dimension has three levels:

- The *application level* considers certain systems. Models for these systems are defined on this level. Code is generated by executing concrete transformations.

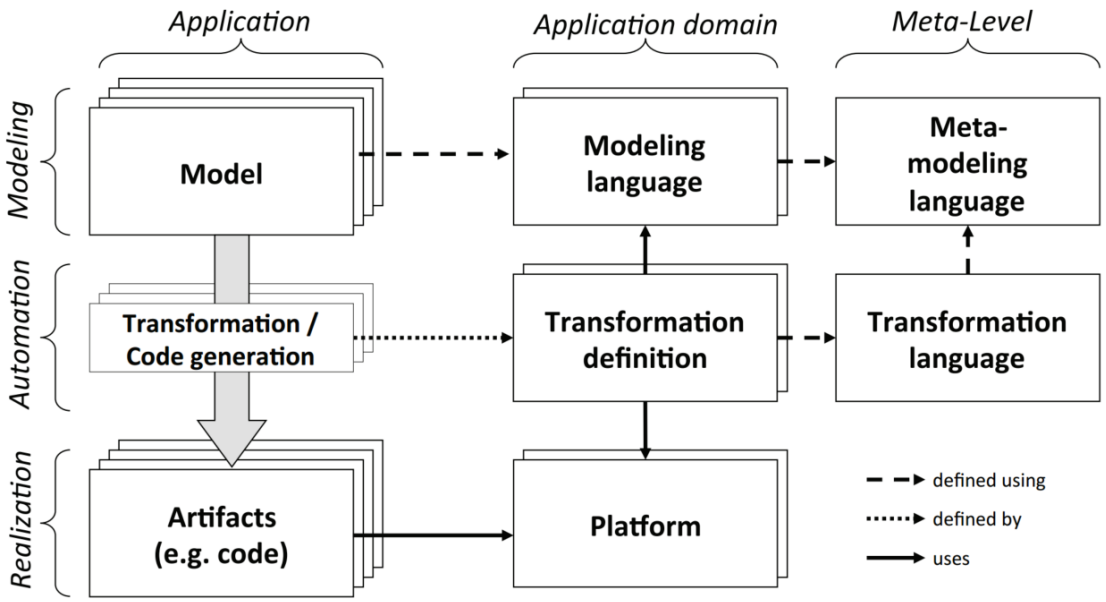


Figure 2.1: Overview of the MDE methodology (top-down process)[BCW12].

- The *application domain level* defines modeling languages and transformations and specifies implementation platforms.
- The *meta-level* provides the abstract syntax for modeling languages and defines the transformation language (see Section 2.1.3).

The core flow of MDE is shown in Figure 2.1 as a wide grey arrow from the models down to the code artifacts. This core flow represents the system construction as a top-down process. A model on top of this process is an abstraction and generalization of the future system. This model limits the scope of the system and defines its static and dynamic structure [BCW12].

A typical MDE-based software development process is shown in more detail in Figure 2.2. It includes *requirement elicitation*, *analysis*, *design*, and *implementation*. These activities underlie any software development process independent of its methodology, be it the traditional waterfall model or any of modern iterative methodologies. According to the mentioned activities there are several types of models in the development process, which are: *requirement models*, *analysis models*, and *design models*. An executable software system can be produced by means of one or more transformations of these models. The first two transitions between the models are executed by applying model-to-model (M2M) transformations. The implementation of the system (executable code) is produced by applying a model-to-text (M2T) transformation to the design model [BCW12]. For details on transformation types see Section 2.1.4.

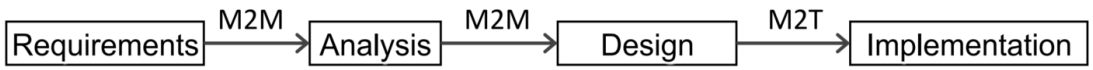


Figure 2.2: A typical MDE-based software development process [BCW12].

The possibility of model transformations is based on the concept of metamodel. The metamodel of a modeling language provides the abstract syntax of that language. This syntax ensures that models follow a defined structure. It serves as a basis for applying operations on models (e.g., storing, querying, transforming, checking, etc.) [BCW12]. More information about modeling languages and metamodeling can be found in the following Sections.

2.1.2 Modeling Languages

Modeling languages are an integral part of MDE. Modeling languages are conceptual tools that allow for formalization and conceptualization of the reality in an explicit textual or graphical form [BCW12]. Generally, a modeling language is defined by three core ingredients and is not complete if any of them is missing or deficient. These core ingredients are [BCW12]:

- *Abstract syntax:* description of the primitive modeling elements, their structure and relationships independent of any particular representation.
- *Concrete syntax:* description of specific graphical or textual representations of the modeling elements. The concrete syntax is used by an engineer during actual design process.
- *Semantics:* description of the meaning of the elements and their combinations. The semantics directly defines the meaning of the abstract syntax and indirectly of the concrete syntax.

According to [Kle09] semantics of a modeling language can be further divided into three types.

- *Denotational semantics* defines a mapping from the modeling language to a formal language by formulating the meaning of all the concepts, properties, relationships, and constraints by means of mathematical expressions.
- *Operational semantics* defines the meaning of the language implementing an interpreter that directly defines the model behavior.
- *Translational semantics* maps the language concepts to another language with clearly defined semantics.

Running systems can only be generated from executable models. A model is executable if its operational semantics is fully specified [BCW12]. To put it in a nutshell, operational semantics defines everything that is changing in a system at runtime, e.g., attribute values, or the current state of the system or its components.

As it follows from the previous paragraph, software systems have not only a static structure but also a behavior. They are dynamically changing during execution and as a result of interaction with users or other systems. Therefore, modeling languages and their models, i.e. concrete MDE-systems, also have two aspects: the *static (or structural) aspect* and the *dynamic (or behavioral) aspect* [BCW12]. The static aspect focuses on the structural shape and architecture of a system. In terms of the Unified Modeling Language (UML)[1] this aspect can be described with structural diagrams like class diagrams, component diagrams, package diagram, etc. The dynamic aspect shows the sequence of actions and algorithms, the communication between system components and the changes of their current internal states. The behavior diagrams in UML are activity diagrams, sequence diagrams, and state machine diagrams. This separation of aspects allows different views on the same system. Nevertheless, these two views should be interconnected to maintain a sound overall picture.

Modeling languages can be classified into two groups: *General-Purpose languages (GPLs)*, which model any domain, and *Domain-Specific languages (DSLs)*, which are designed specifically for a certain domain [BCW12]. The typical examples of GPLs are UML and Petri nets (see Section 2.2.2). The goal of DSLs is to simplify modeling or programming tasks for domain experts. Well-known examples of DSLs are HTML[2] for creating web pages, MATLAB [3] for natural sciences and economics, Simulink for engineering and automatic control [4], Structured Query Language (SQL) for managing data held in a relational database management system. A DSL can not only be developed for a domain, but also for a specific project, e.g., the language Solidity[5] was developed for the blockchain-based platform Ethereum[6]. A third group of customized GPLs for specific purposes can be defined as an overlap of the two big groups. An example for such an intermediate solution are UML-profiles, e.g., the Systems Modeling Language (SysML)[7] and Modeling and Analysis of Real Time and Embedded systems (MARTE)[8].

2.1.3 Metamodeling

Every model is an abstraction and generalization of a real world phenomenon. In order to build a model, an engineer needs a set of rules referring this type of models. Therefore another level of abstraction defining properties of the model itself is needed. This level

[1] UML specifications: http://www.omg.org/spec/UML/2.5/
[2] HyperText Markup Language (HTML) specifications: https://www.w3.org/html/
[3] MATLAB as the product of MathWorks: https://www.mathworks.com/products/matlab.html
[4] Simulink as the product of MathWorks: https://www.mathworks.com/products/simulink.html
[5] The Solidity Contract-Oriented Programming Language: http://solidity.readthedocs.io/en/v0.4.21/
[6] https://www.ethereum.org/
[7] SysML specifications: http://www.omg.org/spec/SysML/1.4/
[8] MARTE specifications: http://www.omg.org/spec/MARTE/1.1/

is called a *metamodel*. Typically a metamodel represents the definition of a modeling language. A metamodel defines all valid models of that modeling language [KÖ06].

In metamodeling practice, a model *conforms to* its metamodel in a way that a computer program conforms to the grammar of the programming language. If a model conforms to its metamodel, all the elements of the model can be *instantiated* from the corresponding classes of the metamodel. Figure 2.3 presents the notation of these two relationships.

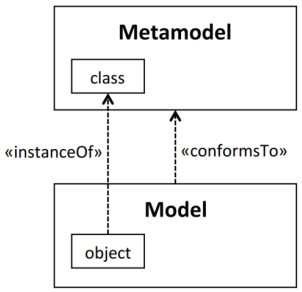


Figure 2.3: Relationships between metamodel and model [BCW12].

Metamodels can be used *constructively* as a set of rules for building models and *analytically* as a set of constraints a model must adhere to conform to its metamodel. The transformation aspects discussed in the next Section are also based on metamodels of modeling languages [BCW12].

2.1.4 Transformations

Model *transformations* are another integral part of MDE in addition to models and model languages. Transformations are defined on the level of metamodels (cf. Figure 2.4). To enable a transformation, it should be specified which elements of the source metamodel are to be transformed into which elements of the target metamodel. This specification is called mapping. It allows to automate transformations with an execution engine. The transformation itself is performed between two models which conform to the source and target metamodels [CH06, BCW12].

As mentioned in the Section 2.1.1, there are two types of model transformation: M2M transformations and M2T transformations. These two transformations are used in the context of this thesis and explained in detail in the Section 3.3. Concrete implementations are demonstrated in the Section 4.6.

M2M transformations take one or more models as input and generate one or more models as output. If transformation is performed between two models, defined in two different languages, it is referred to as *exogenous*. Such transformations are mostly *out-place* transformations, since the target model is created from scratch in the target modeling language. Transformations within the same language are called *endogenous*. These transformations are mostly happening *in-place* which means they are rewriting a source

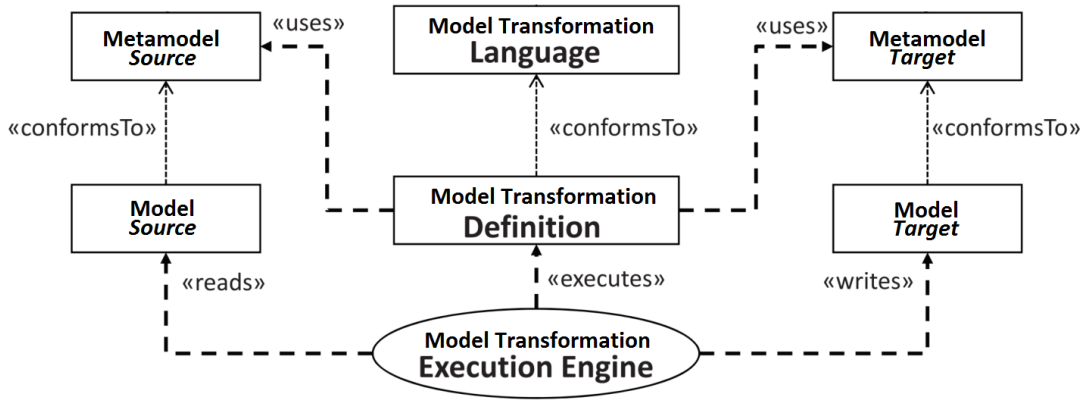


Figure 2.4: Definition of a transformation between models [BCW12].

model by creating, deleting, and updating elements. An example for an endogenous in-place transformation is model refactoring [BCW12].

M2T transformations are mostly used in code-generators to execute the transition from the modeling level to the code level. On the code level it is not only the functional code that can be generated, but also test code, logging code, deployment scripts, etc. A code-generator typically generates code in a certain programming language. Model elements are transformed into the corresponding code statements, e.g., if-statements and cycles, or language structures, e.g. classes. Thus, transforming the same model in different code generators means to get running code in different programming languages [BCW12].

2.2 Process Mining

2.2.1 Overview

In Process Mining (PM) the object of discovery are business processes or processes in general, whereas data mining aims to discover unsuspected data structures in big data sets [HSM01]. The goal in both cases is to make this new derived information useful for the data owner. PM-techniques analyze processes that are running in an information system on the basis of event logs and discover non-trivial and useful information from those logs. Such process-centric systems are called Process-Aware Information System (PAIS) [DvdAtH05b]. The process notion is an integral part of these information systems which distinguishes them from, e.g., databases or text editors. For example, Enterprise Resource Planning (ERP) systems, Business Process Management (BPM) systems and Customer Relationship Management (CRM) systems are aware of the processes they support.

Each *event* in PAIS is a well-defined and globally unique process step that refers to

an activity and a case. *Event logs* are sequentially ordered events registered by a PAIS [DvdAtH05a]. These event logs can grow into significant sets of data and can be recognized as Big Data especially in terms of their volume and the velocity they are produced with. The main challenge of PM is to capture behavioral aspects of PAIS on the basis of event data [vdA16].

PM can be considered as an interdisciplinary bridge between data science and process science. Further components of this bridge are shown in Figure 2.5. Data science approaches are too data-centric to provide a complete understanding of end-to-end processes. Process science approaches tend to concentrate on process models without considering that event data can be used to improve processes. PM is a relatively young research discipline that fills the gap between traditional model-based process analysis and data-centric analysis such as machine learning and data mining [vdA16].

Figure 2.5: Process mining as the bridge between data science and process science [vdA16].

Three main types of PM in terms of input and output [VDA12]:

- *Discovery.* This technique doesn't use any a-priori information about the system. Event logs are the only input that discovery techniques take for explaining system behavior. For example, the α-algorithm (see Section 2.2.3) constructs a process model in form of a Petri net using exclusively an event log without any additional information.

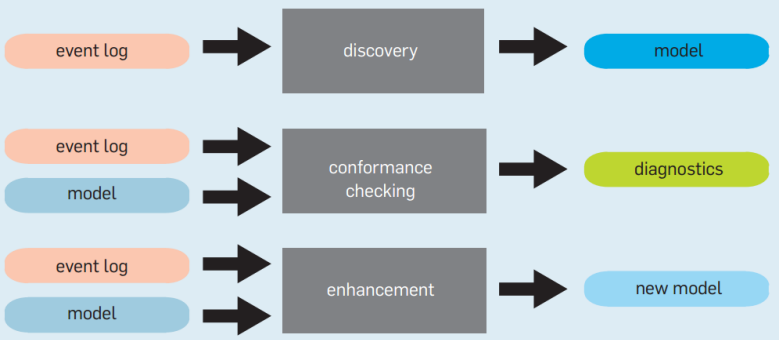


Figure 2.6: The three basic types of process mining in terms of input and output [VDA12].

- *Conformance checking.* This technique is used to check whether the real behavior of the system as registered in the logs corresponds to the process model and vice versa. Thereby, the recorded event logs are compared with the model obtained from event logs. Thus, conformance checking is applied to detect, locate, and explain discrepancies.

- *Enhancement.* The goal of this technique is to improve existing process models by means of new information derived from event logs. There are two types of enhancements: *repair* and *extension*. The first one aims to modify the system so that it better reflects the real phenomena. The latter one is used to add a new perspective to the process model, e.g., by enriching the model with performance data.

Orthogonal to the dimension of these three PM-types, there is a dimension of different PM-perspectives [vdA16]. These perspectives give a complete picture of the aspects that PM intends to analyze.

- *The control-flow perspective* reflects the ordering of activities being performed. The result of mining this perspective is a convenient generalization of all possible paths in form of, e.g., a Petri net or UML activity diagram.

- *The organizational perspective* focuses on resources which executes activities of the process (e.g., people, systems, roles, and departments). The result of mining this perspective is an organizational structure or a social network that shows organizational units and their interrelations.

- *The case perspective* is concerned with properties of the cases. A *case* or a single *process instance* is an end-to-end execution of the process. It can be characterized by its path, resources, or individuals working on it and the values of the corresponding attributes at different points in time.

- *The time perspective* focuses on timing and frequency of events. Every event has a timestamp, which allows to analyze and predict bottlenecks, actual execution times, utilization of resources, remaining processing time of running cases, etc.

2.2.2 Petri Nets as Process Models

A Petri net is a directed bipartite graph that consists of two types of nodes: *places* represented by circles and *transitions* represented by rectangles [RR98]. These nodes are connected by directed arcs. An example of a Petri net is shown in the lower part of Figure 2.7. The structure of a Petri net is static, however *tokens* can flow through the network according to the firing rules defined in transitions in order to demonstrate dynamic behavior. A transition is called *enabled* when there are enough tokens on all the input places of this transition, so that it is ready to fire [RR98].

According to van der Aalst [vdA16], Petri nets are the oldest and best investigated GPL for process modeling allowing to express concurrency. In a process two events are called concurrent, when they are executed during overlapping time periods and are not causally affecting each other [Lam78]. The firing of a Petri net is nondeterministic, meaning that several transitions may be enabled at the same time and several tokens may reside in different places. This non-determinism gives the possibility of concurrency modeling.

As mentioned in the previous Section, Petri net can be a mining outcome of the PM in control-flow perspective. Generally, a Petri net shows the control-flow backbone for an end-to-end process. Mining outcome can also be represented in notations different from Petri nets. Additionally to Petri nets van der Aalst [vdA16] discusses such kinds of notations as Business Process Model and Notation (BPMN), Business Process Execution Language (BPEL), Event-driven Process Chain (EPC), and Yet Another Workflow Language (YAWL). Throughout this thesis Petri nets are considered to be the basis PM outcome since it is a GPL and can be used for mining different kinds of processes running in software systems. Other mentioned notations are DSL specifically created to represent business processes and workflows.

The connection between an initial process model and the mining outcome in form of a Petri net is shown by an example. In the upper part of Figure 2.7 there is a business process depicted in BPMN. This process is part of a system which is deployed on a workflow engine. The system runs and produces an event log consisting of log entries. This event log is mined with a PM algorithm like the α-algorithm or inductive miner (the explanation of these algorithms is given in the Section 2.2.3). The resulting Petri net, i.e., mining outcome, is presented in the lower part of Figure 2.7. The alignment between the initial BPMN model and the Petri net shows the correct number of detected activities, as well as the correctly captured sequence and interconnections between them. Every path possible in the initial model is also possible in the Petri net. Figure 2.7 illustrates this alignment by several selected disjunctions and conjunctions.

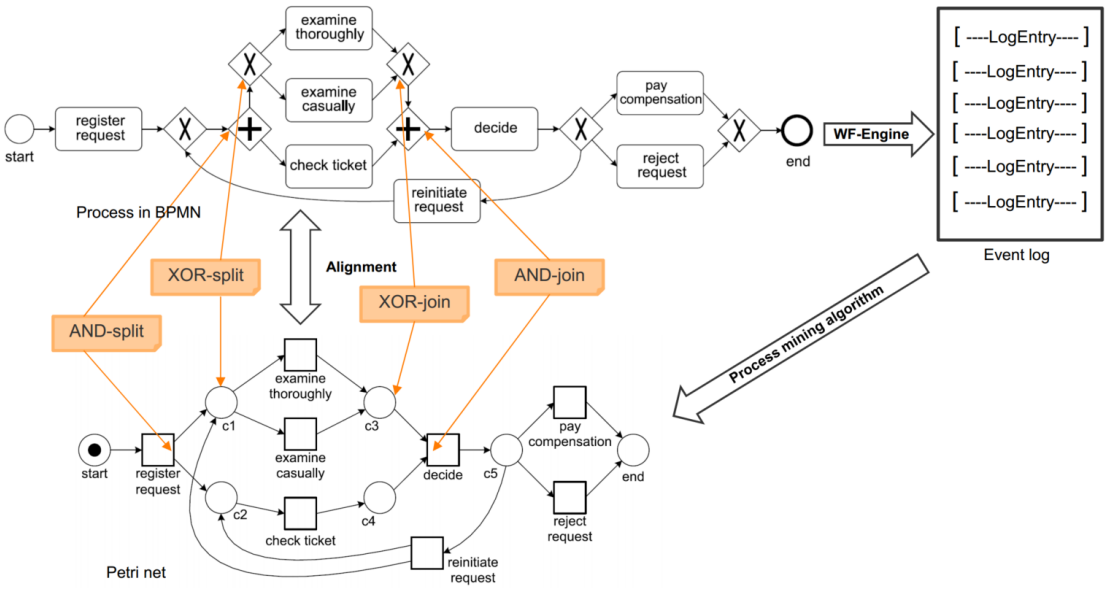


Figure 2.7: Process mining. An example. Based on [vdA16]

2.2.3 Process Mining Algorithms

Family of Alpha Algorithms

The basic α-algorithm was introduced in 2004 by van der Aalst [vdA16] as one of the first process discovery algorithms that could deal with concurrent threads. The α-algorithm is simple and provides a good introduction to the PM. However, this algorithm can not deal with complex routing constructs, noise, and infrequent or incomplete event logs. Nevertheless, the algorithm forms an essential baseline for discussing more advanced algorithms and process discovery challenges.

The following formalization is needed to further explain the α-algorithm [Den13]. Let A be a set of activities. A bag of strings over A forms an input for the α-algorithm. This bag is called an *event log* $L \in Bag(A^*)$, whereas each string $\sigma \in L$ is called a *trace*. Four ordering relations can be defined for pairs of activities $a,b \in A$:

- b directly follows a: $\boldsymbol{a >_L b}$ if and only if there is a trace $\sigma = \langle t_1, t_2, t_3, ..., t_n \rangle$ and $i \in \{1, ..., n\text{-}1\}$ such that $\sigma \in L$ and $t_i = a$ and $t_{i+1} = b$
- Sometimes b directly follows a, but a never follows b: $\boldsymbol{a \rightarrow_L b}$ if and only $a >_L b$ and $b \not> _L a$
- a and b are parallel activities: $\boldsymbol{a \|_L b}$ if and only if $a >_L b$ and $b >_L a$
- a and b are unrelated activities: $\boldsymbol{a \#_L b}$ if and only if $a \not> _L b$ and $b \not> _L a$

Precisely one of these relations holds for any pair of activities of any log L [vdAWM04]. Therefore, a distinct relation matrix of the log L, i.e. the so-called *footprint*, can be created. These log-based ordering relations are further used to discover patterns and generate a Petri net. Figure 2.8 presents examples of captured relations and corresponding discovered patterns.

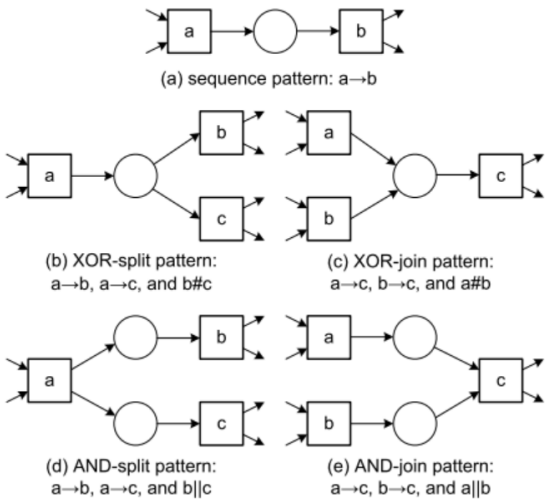


Figure 2.8: Typical process patterns and the footprints they leave in the event log [vdA16].

To put it in a nutshell, the α-algorithm consists of two basic steps [vdAWM04] (cf. Figure 2.9):

1. Reconstruction of four possible ordering relations from a log as a footprint
2. Generation of a Petri net from this footprint

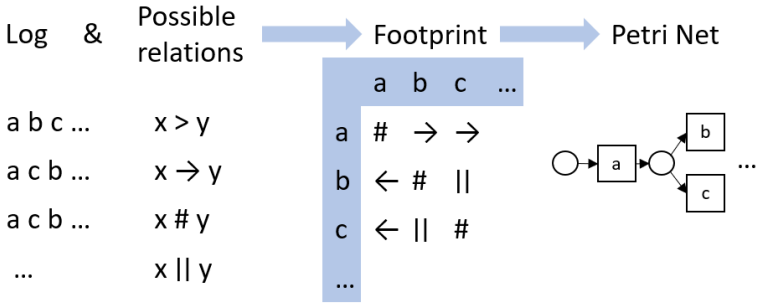


Figure 2.9: Two basic steps of the α-algorithm

These two basic steps are divided and formally described by Van der Aalst as the following eight rules [vdAWM04]:

1. L is an event log over the set of activities T. The first step checks, which activities appear in the log and creates their set T_L.
   - $T_L = \{t \in T \mid \exists_{\sigma \in L}\ t \in \sigma\}$
2. The second step creates the set of all the start activities, i.e., input activities T_I. These activities appears first in some trace.
   - $T_I = \{t \in T \mid \exists_{\sigma \in L}\ t = \mathit{first}(\sigma)\}$
3. The third step creates the set of all the end activities, i.e., output activities T_O. These activities appears last in some trace.
   - $T_O = \{t \in T \mid \exists_{\sigma \in L}\ t = \mathit{last}(\sigma)\}$
4. A is the set of input transitions and B is the set of output transitions. All elements of A should have causal dependencies with all elements of B. Additionally, the elements of A should never follow each other. The same requirement holds for B. The fourth step creates the set X_L containing all pairs that meet these requirements.
   - $X_L = \{\ (A, B) \mid A \subseteq T_L \wedge A \neq \emptyset \wedge B \subseteq T_L \wedge B \neq \emptyset \wedge$
     $\forall_{a \in A} \forall_{b \in B} a \rightarrow_L b \wedge \forall_{a_1, a_2 \in A} a_1 \#_L a_2 \wedge \forall_{b_1, b_2 \in B} b_1 \#_L b_2\}$
5. In the fifth step the set Y_L is derived from the set X_L. Y_L contains only the largest elements, i.e. maximal pairs, of X_L. These pairs will form the places of the resulting Petri net.
   - $Y_L = \{\ (A, B) \in X_L \mid \forall_{A', B' \in X_L} A \subseteq A' \wedge B \subseteq B' \implies (A, B) = (A', B')\}$
6. The sixth step combines all the discovered places P_L, including the single input place i_L and the single output place o_L.
   - $P_L = \{p_{(A,B)} \mid (A, B) \in Y_L\} \cup \{i_L, o_L\}$
7. In the seventh step the arcs F_L of the Petri net are generated. All places ($p_{(A,B)}$ have A as input nodes and B as output nodes. All start transitions in T_I get i_L as an input place, all end transitions T_O get o_L as output place.
   - $F_L = \{(a, p_{(A,B)}) \mid (A, B) \in Y_L \wedge a \in A\} \cup \{(p_{(A,B)}, b) \mid (A, B) \in Y_L \wedge b \in B\}$
     $\cup\ \{(i_L, t) \mid t \in T_I\} \cup \{(t, o_L) \mid t \in T_O\}$
8. The behavior observed in log L is described in the resulting Petri net with its places P_L, transitions T_L, and arcs F_L.
   - $\alpha(L) = (P_L, T_L, F_L)$

One of the major limitations of the α-algorithm is inability to discover *short loops of one or two events*. In the examples presented in Figure 2.10 both C transitions would stay disconnected from the rest of the mined Petri nets [vdA16]. Medeiros et.al. [dMvdAW03] present the α^+-algorithm that tackles this limitation. The α^+-algorithm uses a pre-processing phase to deal with loops of length two and post-processing phase to insert loops of length one.

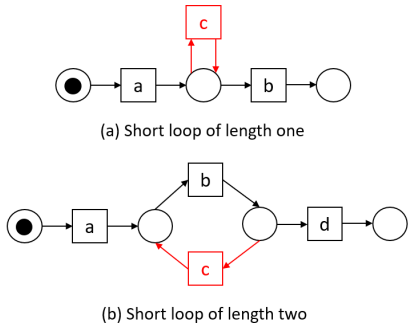


Figure 2.10: Petri Nets with short loops of length one and two.

Additionally to this limitation the α-algorithm has problems correctly discovering implicit dependencies [vdA16]. These dependencies originate from a particular use of so called *non-free choice constructs* in Petri nets. In free choice Petri nets transitions that consume tokens from the same place should have identical input sets [DE95]. However, many real-life processes do not have this property and show non-free choice constructs. These limitation cause redundant implicit dependencies in the Petri net, which do not actually appear in the log. The α^{++}-algorithm introduced by Wen et.al. [WvdAWS07] is a further extension of the α-algorithm that allows to cut down such redundant implicit dependencies.

Another limitation of the α-algorithm is its *low robustness against noise and incompleteness* [vdA16]. In α-algorithm an event log is considered to be a representative behavior description, i.e., to be complete. However, in real life event logs tend to contain noise and demonstrate incompleteness. In the context of PM under *noise* we understand rare and infrequent behavior that is not representative for the typical behavior of the process. In case of *incompleteness* some events are missing in the log, which makes it difficult to discover some of the underlying control-flow structures, i.e., ordering of activities.

The α-algorithm does not take frequencies of log activities into account, i.e. rare and frequent patterns have equal effect on mining results. This issue is resolved in the improved mining algorithms, such as *heuristic miner* and *inductive miner*.

Heuristic miner

In recent years process discovery algorithms were significantly improved. An example for this development are the heuristic mining algorithms that were first described in [WA03]

and [WR11]. These algorithms use causal nets as representation of the mining result. In a casual net each activity has input and output bindings (cf. Figure 2.11).

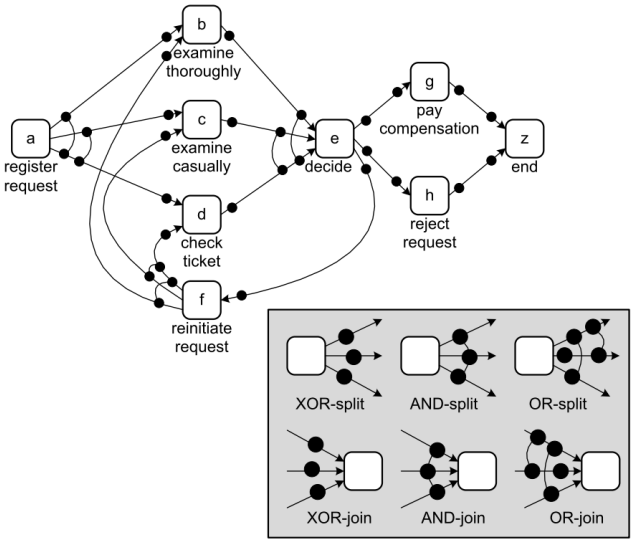


Figure 2.11: An example of a casual net [vdA16].

The basic idea of the heuristic algorithm is to exclude infrequent paths from the model. For this purpose the heuristic algorithm builds two matrices [WA03]:

- A matrix of directly-follows relations used in α-algorithm, for instance, $a >_L b$. This matrix shows for each pair of activities, how often one activity is directly followed by another activity i.e., the value of *dependency relation.*
- A matrix of calculated *dependency measures* with values between -1 and 1. For instance, a dependency measure close to 1 indicates a strong positive dependency between two activities (activity a is often the cause of b).

Using the information from these matrices and input parameters called *thresholds* we can produce the *dependency graph* that reveals the backbone of the process model [vdA16]. In a preprocessing step thresholds, such as dependency threshold and relative to best threshold, are set for both matrices. *Dependency threshold* sets the minimum value of dependency measure between events. It means, we want to accept dependency relations between activities which dependency measure is above the value of the Dependency threshold. *Relative to best threshold* sets the minimum value of the difference between event dependency value with the maximum dependency value. It means, we want to accept dependency relations between activities with dependency measure for which the difference with the best dependency measure is lower than the value of Relative to best threshold. For example, in Figure 2.12 only pairs of activities with dependency relations

higher than 10 and dependency measures higher than 0.9 are taken into the resulting dependency graph. Heuristic miner can produce different dependency graphs for the same log by adjusting the thresholds. Users can decide whether they would like to focus on mainstream behavior or also include noisy behavior into the final result. Moreover, this refinement possibility allows to better deal with loops of length two and long distance dependencies [WA03, WAM06, WR11].

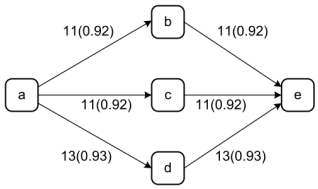


Figure 2.12: An example of a dependency graph [vdA16].

In the final step, the heuristic algorithm learns the splits and joints (cf. Figure 2.11) by replaying the event log on the dependency graph in order to complete the resulting casual net.

Inductive Miner

The family of Inductive Miner (IM) techniques represents another significant improvement of the basic α-algorithm [JJLFA13]. IM algorithms can handle huge incomplete event logs with infrequent behavior, but still ensure formal correctness criteria such as the ability to rediscover the original model [vdA16].

The α-algorithm creates a Petri net, whereas the basic IM algorithm produce an equivalent process tree (cf. Figure 2.13). Any process tree can be automatically converted into a Petri net with further reduction of excessive silent transitions. Moreover, the basic IM can discover a wide class of processes and detect correct process models in situations where the α-algorithm fails.

The basic IM algorithm uses so called *directly-follows graphs* corresponding to the directly-follows relation of the α-algorithm, i.e. $a >_L b$. The IM algorithm recursively cuts the initial event log into smaller sublogs. A directly-follows graph is created for every sublog. The iterations are repeated until a base case (sublog with only one activity) is reached. IM uses four types of cuts [JJLFA13]:

- *Exclusive-choice cut* corresponding to the operator $\times$.
- *Sequence cut* corresponding to the operator $\rightarrow$.
- *Parallel cut* corresponding to the operator $\wedge$.
- *Loop cut* corresponding to the operator $\circlearrowleft$.

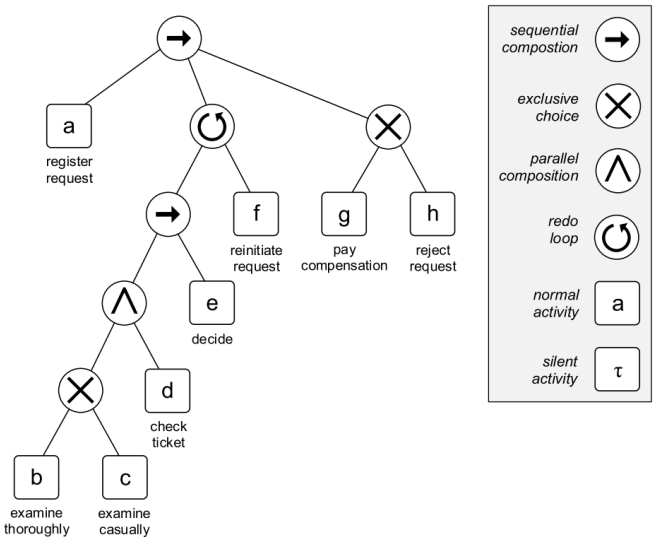


Figure 2.13: An example of a process tree [vdA16].

The IM algorithm always guarantees soundness of the produced process model, i.e. such models are able to replay the whole initial log. Activities in the resulting model are not duplicated, which makes models rather simple and general. Hence, IM algorithm may create underfitting models. This may happen when the observed behavior requires a process tree with duplicate or silent activities [vdA16].

In the basic IM algorithm the problem of handling noise and incompleteness still persists, as it does in the α-algorithm. However, two important extensions for the IM algorithm were developed. The first extension *IM-infrequent algorithm* was introduced in [JJLFA14]. This algorithm applies different types of filtering to the event log, such as filtering of infrequent arcs or activities, with the goal to capture the mainstream behavior. Additionally to directly-follows graph the IM-infrequent algorithm also uses the *eventually-follows graph*. Here IM-infrequent algorithm shows some similarities with the heuristic miner. Moreover, similar to heuristic miner, IM-infrequent algorithm uses relative threshold. An activity is only discovered by this algorithm, if the average number of its occurrences per trace of in the log is close enough the relative threshold [JJLFA14].

The *IM-incompleteness algorithm* is the next extension to the basic IM algorithm and the IM-infrequent algorithm [LFvdA14a]. The assumption of event logs being directly-follows complete is unrealistic for some relatively small real life logs. They can often be poorly structured and have missing activities, i.e., they are incomplete. The IM-incompleteness algorithm builds so called *probabilistic activity relations* based on both the directly-follows graph and the eventually-follows graph [LFvdA14a]. During the recursive cutting this algorithm uses the "most likely cut" and completes the probably missing parts of the resulting process model.

2.2.4 Timing and Performance Analysis in Process Mining

Time perspective and performance perspective are both important perspectives of PM as mentioned in Section 2.2.1. Timing analysis helps to make predictions regarding flow times of processes with an increasing number of threads [vdASS11]. Timing of the system is one of the core components to create simulation models along with runtime information about decision points and organizational structure [RMSvdA09]. Simulation models are especially important if the system is hard to test under real conditions, like medical systems.

To gain insight about the time perspective, first a discovery algorithm such as α-algorithm or inductive miner should be applied to automatically generate a basic control-flow of the system's real behaviour, i.e., a process model. Mostly this control-flow is represented in form of a Petri net. In the next step performance analysis is carried out to detect information about execution times and waiting times for activities, and probabilities for taking alternative paths. This approach is called *Replay* of event logs on the process model (cf. Figure 2.14).

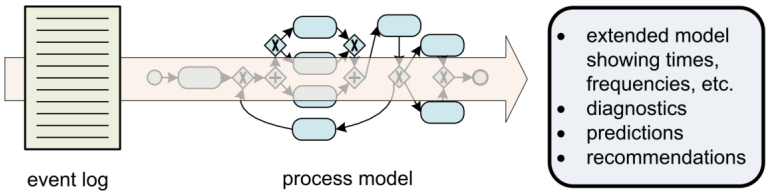


Figure 2.14: Replay of event logs on a discovered process model [vdA16].

In Figure 2.15 an extended process model is shown with additional performance information identified after event logs were replayed on the discovered Petri net. For each detected activity the *execution time (E)* and *waiting time (W)* are defined as a normal distribution N with arithmetic mean and standard deviation. Extracting such information from event logs is easy, since every event has a timestamp. The minimum and maximum execution times for single activities, as well as whole cases can also be extracted. In disjunction points the probabilistic values for selecting one or another path are calculated based on how often each path was selected during the Replay. The case arrival rate is provided at the starting node.

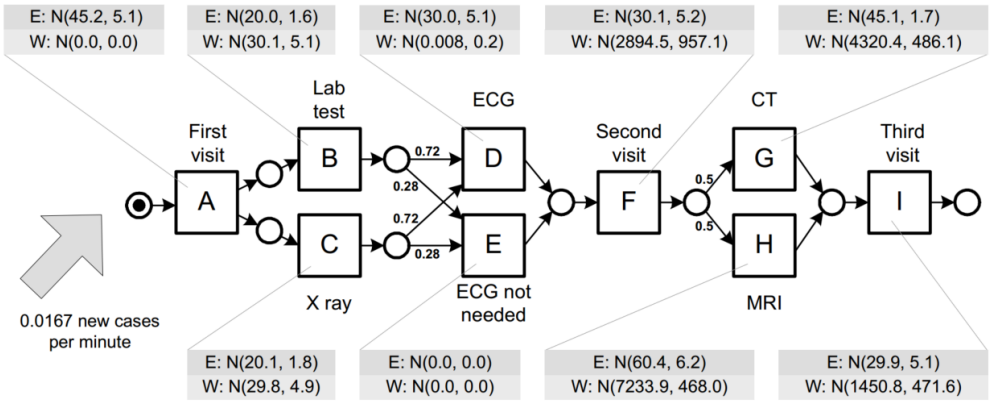


Figure 2.15: Example model enhanced with the performance perspective [RMSvdA09]

2.3 Models@run.time

There is an emerging research work focusing on real-time systems, runtime phenomena, runtime monitoring, and discussing the back propagation of runtime information to engineering [DGJ+16, SZ16]. An important concept connected with these research fields is *models@run.time (M@RT)* [BBF09, BFT+14, AGJ+14].

A terminological framework for M@RT is suggested in [BFT+14] and shown in Figure 2.16. The running software system represented in the center consists of an application and a runtime platform, where the application is deployed and executed. The environment represents the outside world that the running system interacts with in order to provide its functionality. During this interaction the system can be influenced by one or more contexts and must adapt itself to changes in the environment.

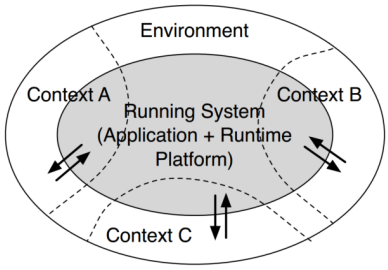


Figure 2.16: A Terminological Framework for M@RT [BFT+14].

As stated in [BBF09], M@RT are adaptation mechanisms to leverage software models according to the changing execution environment. Each M@RT is a causally connected self-representation of the running system that highlights its structure, behavior, or goals from a problem space perspective. Research on M@RT aims to increase the relevance of models produced in MDE approaches to the runtime environment. Thereby, the concept

of M@RT brings together models in MDE context and runtime information and combines them into a new class of reflective systems which are tractable and able to predict certain aspects of their own behavior for the future [AGJ+14].

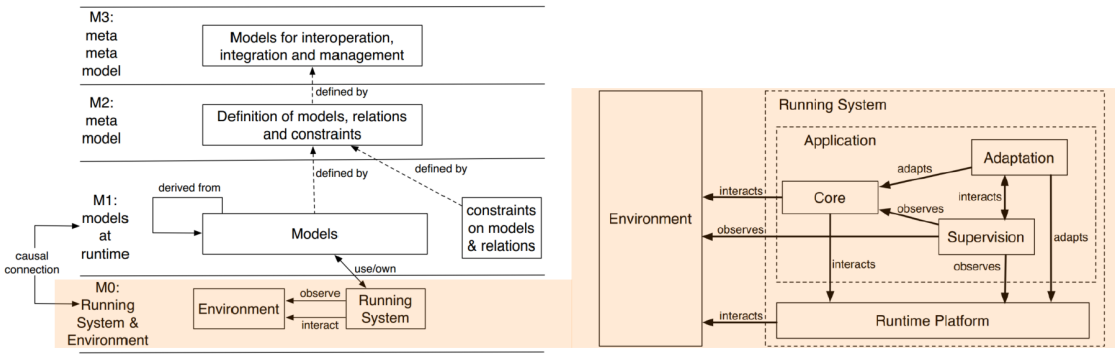


Figure 2.17: A Conceptual Reference Model for M@RT with a close-up of M0 level. Based on [BFT+14].

Bennaceur et. al. [BFT+14] propose a conceptual reference model for M@RT systems. The four levels of the model from M0 to M3 are illustrated on the left side of Figure 2.17. A close-up of the level M0 is represented on the right side of Figure 2.17. According to the terminological framework mentioned above, the M0 level contains the running system and the environment. As the application core and the runtime platform interact with the environment, the supervision component observes them and triggers the adaptation component to reason and plan an adaptation. The adaptation component performs the adaptive changes based on the observations in the runtime platform and the application core.

The higher M1 level contains the runtime models, relations between them, and their constraints. These models are causally connected with the events registered by the supervision component and actions performed by the adaptation component. Event processing can lead to adaptation of the M1 models and further propagation of these adaptive changes to the running system. Conceptually, this relation between M0 and M1 represents a feedback control loop with M0 being the feedback source. The goal of the feedback analysis is to achieve a desired level of fidelity between the M1 model and the M0 system running in its environment. Similar to the approach described in the Section 2.1.1 (Figure 2.1) the M2 level provides the metamodel of the modeling language to create the M1 models. Finally, the top M3 level includes the meta-metamodel that is used to define the M2 metamodels.

M@RT systems complement classic reflection with strong modeling foundations. In fact, modeling is a central component of the M@RT concept. M@RT directly benefits from MDE tools and approaches, such as metamodels, editors, simulators, compilers, etc [AGJ+14]. However, these tools are usually thought for usage at design time, and it is difficult to adopt them at runtime. Thus, the current MDE techniques should be

extended and adjusted to be directly applied to M@RT [AGJ+14].

One of the key aspects of M@RT systems is their ability to project some characteristics of reality, such as contexts, to the modeling space for further analysis. A M@RT system is able to perform manageable reflection and predict particular paths of its own behavior in the future with accumulated knowledge. This ability of reasoning about system's future states, i.e., predictable reflection, distinguishes M@RT systems from reflective systems, that are only able to reason on the current system configuration [AGJ+14].

A predictable reflection is especially beneficial for such interrelated application domains as safety-critical embedded systems or CPS. In CPS, physical and software components are intensely interconnected, performing multiple and distinct behaviors, differently interacting with each other depending on the context, and tightly integrated with the Internet [CPS]. Examples of CPS include process control systems, robotics systems, smart grid, autonomous automobile systems, automatic pilot avionics, and medical monitoring. Many CPS are safety-critical. Failure or malfunction of a safety-critical system may lead to serious injuries, severe property damage or environmental harm [SCS].

For safety-critical systems the majority of constructive decisions are made at design time in order to make configurations predictable. Nevertheless, there is an increasing need for more flexible adaptive systems, which are able to deal with the unforeseen, but still ensuring safety. Nowadays safety-critical systems are expected to stay up and running while reacting and adapting to changes in both internal and environmental contexts [BBF09]. This contradiction between adaptivity and safety is hard to solve. According to [AGJ+14], M@RT is the ultimate solution to enable unanticipated adaptations while ensuring required security standards. However, M@RT concept faces significant open challenges associated with the engineering of adaptive systems [BFT+14]:

1. *Developing and updating runtime models.* The supervision component observes the running system in order to develop runtime models. It is important to consider, how these runtime models are consistently updated at the levels M1, M2 and M3 (cf. Figure 2.17). Additionally, it is essential to establish a seamless connection between MDE processes and the processes of runtime models creation and evolution. The major challenge here is to maintain this connection so that the runtime models and the running system with its environment stay stable and sound.

2. *Reasoning and planning for adaptation.* Runtime models are the result of the reasoning and adaptation process. If the reasoning identifies some violations of functional or non-functional properties and the need to adapt them, runtime models are manipulated for further propagation of changes to the running system. The challenge here is to automate reasoning and adaptation mechanisms in order to perform the changes on-line on a running system.

3. *Maintaining different runtime models.* Complex software systems, such as CPS, have multiple concerns, e.g., reliability, performance, or functional properties. Typically,

separate models need to be maintained for each concern in order to provide separate reasoning. Dealing simultaneously with several concerns requires mechanisms to manage dependencies between runtime models and to keep them consistent.

4. *Establishing fidelity and consistency among models and the running system.* Propagation of changes from runtime models to the system requires mechanisms to map these changes between the levels M1 and M0. Moreover, these mechanisms should guarantee safe adaptations of the running system. This includes ensuring fidelity of runtime models with the running system to avoid drifting and instability.

In this thesis, we concentrate on the first challenge of capturing runtime models and keeping them sound through all modeling levels, as well as the third challenge of maintaining different types of runtime models for multiple concerns, such as functional properties, performance, and component interrelations. We present an approach to tackle these challenges in the next Chapter 3.

CHAPTER 3

Execution-based Model Profiling

3.1 Overview

The development of software systems traditionally has a clear distinction between design activities and runtime execution [Mao09]. However, shorter innovation cycles and rapidly changing customer needs urge for fast changes in already running systems [LSVH14]. As mentioned in Section 2.3, the source of information for such adaptations is often the running system itself and the data gathered about its actual behavior. In order to integrate this new information into a running system it is necessary to smooth the clear distinction between design and execution phases. This need is especially urgent for safety-critical software systems, which are expected to integrate changes in their execution environment without any downtime. Therefore, these systems are required, if possible, to readjust their behavior at runtime without human intervention [BBF09].

Prescriptive and descriptive models in software engineering are the key concepts to consider on the way of creating a link between design and execution. Models in software engineering are typically created as *prescriptive models* to prescribe something or as *descriptive models* to describe something [HPE+16]. In the prescriptive context the subject of modeling is not yet developed. A prescriptive model itself prescribes the subject, its scope and detalization level. Prescriptive models are defined based on the information available at design time according to the model intent. Descriptive models portray an existing subject. The observed behavior of the subject to be described is used as an input to create descriptive models [HPE+16, Lud02]. The main difference between prescriptive and descriptive models is, that the first ones are created before the subject of modeling and the second ones are created after the subject of modeling [Lud02].

These both types of models are the cornerstone of the approach we present to create a link between design and execution phases of software life cycle. Our approach, *Execution-based Model Profiling (EbMP)*, is a combined but loosely-coupled usage of MDE approaches

and PM techniques (cf. Section 3.3). Traditionally, PM algorithms are used to analyze business processes, which are executed by people using information systems (first of all, PAIS). In EbMP, we use PM algorithms to analyse processes running in autonomous software generated by MDE tools. The alignment of these two different research fields may help us, e.g., to verify the alignment of models at design time and runtime, and pave the way for further automated back-propagation of the collected feedback to a running system.

3.2 Approach

Prescriptive and descriptive models in the context of MDE and PM are shown in Figure 3.1. In MDE, models are defined by people as prescriptive models and used in the top-down process of software development to generate executable software. This is a view on the system at design time, but systems can also be observed at runtime. The generated software system runs, registers important actions and produces event logs as described in the Section 2.2.1). Event logs can be used in the bottom-up process to produce descriptive models of the observed system. These descriptive models can be processed by applying process mining algorithms. Descriptive models show how the system is actually realized, how it is operating in a certain environment, and what underlying processes are running in the system. Execution-based feedback from descriptive models forms the basis for further analysis of the real behavior of the system. Information derived from this analysis can be incorporated into prescriptive models in order to continuously improve them. In the current practice, these potential benefits of descriptive modeling and back-propagation of data from operation to design are mostly overlooked. In this thesis the term *prescriptive models* is used referring to the design models in the MDE context. The term *descriptive models* is used in relation to models produced at runtime.

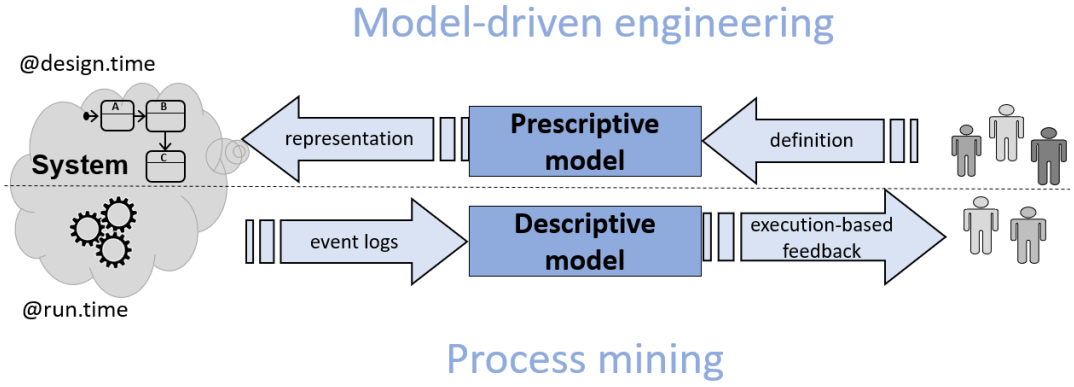


Figure 3.1: Prescriptive models vs. descriptive models.

In order to build the bridge from operation to design, it is important to consider the evolutionary aspect of engineering artifacts, i.e., the fact that they change over time. The feedback from the operation after the release can be reflected in prescriptive models to

cover the complete life-cycle of a system. The question remains: what form and semantics should this feedback have in order to provide meaningful connection between prescriptive and descriptive models.

The core idea of our approach, EbMP, is to equip MDE-based systems in such a way, so that their operational data can be stored in a structured form and transformed into abstracted model representations. In a nutshell, we take a running MDE-based system, which has some prescriptive models as a basis, register all the operational system changes with reference to the prescriptive models, and store them as structured descriptive models, i.e., observation models (see Figure 3.2). After that we analyze the observation models and create so-called *model profiles* out of them. The process of model profiling is a process of producing snapshots of system behavior during a certain time period. For example, we observe a system for an hour and then form a model containing information about its behavior, possible outliers, registered failures, etc. This aggregated meta-information is then a model profile of this system or its part for a certain time period. Since operational system changes are registered with reference to prescriptive models, the generated model profiles inherit the semantic relation to these models. Therefore, we are able to align these model profiles with prescriptive models for detecting inconsistencies between designed behavior and runtime behavior. Additionally to that, we consider our approach beneficial for maintaining runtime models for multiple concerns. The observation models produced in our approach contain structured data to evaluate not only the functional properties, but also performance. Furthermore, to mention one of the future applications, the resulting model profiles can be used for automated back-propagation of runtime data and the adaptation of prescriptive models.

3.3 Unifying Framework

For the purposes mentioned in the previous section, we define a framework providing a generic approach that allows to create a link between downstream information from the MDE-based systems and upstream information gathered at runtime. The unifying framework (see Figure 3.2) has two perspectives: the *prescriptive perspective* and the *descriptive perspective*. The prescriptive perspective is presented on the left-hand side. In this perspective the models are created before a system is built, i.e., the models are used to create the system. The descriptive perspective is presented on the right-hand side. Here the models are created after the system is built, i.e., they are extracted from the running system. In the following, we describe Figure 3.2 from left to right.

The prescriptive perspective is based on the MDE approaches described in Section 2.1.1 and shown in Figure 2.1. At the metamodeling level the *design language* is specified. This specification defines the syntax as well as the semantics of the design language. Under design language we mean any modeling language, such as UML, SysML or a DSL for particular domain of interest. An example of a DSL for the following case study is described in Section 5.3 and presented in Figure 5.1. In our approach, the design language has two different aspects: a static and a dynamic one. The static aspect

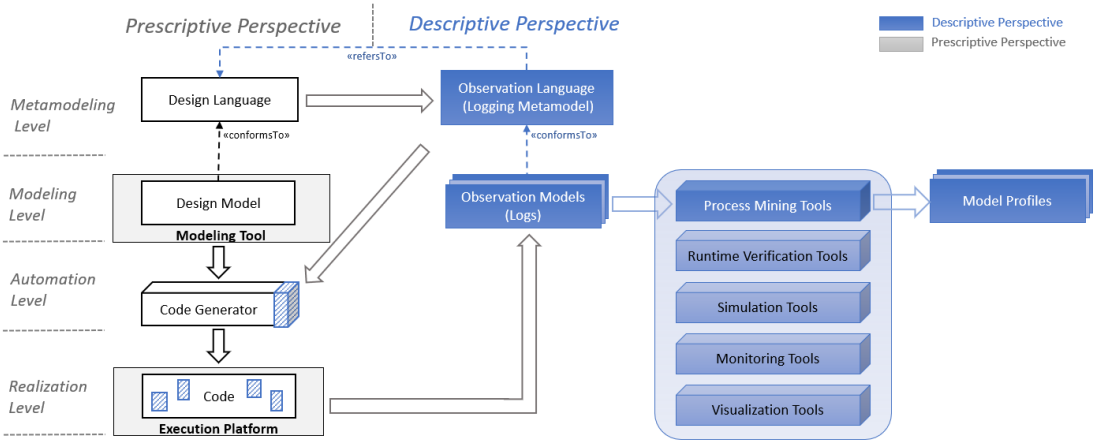


Figure 3.2: Framework for execution-based model profiling.

allows to model the main elements of the modeled entity and their relationships. The dynamic aspect allows to model the behavior of these elements in terms of events and interactions that may occur between the elements. For instance, in UML the static aspect is modeled through class diagrams or component diagrams and dynamic aspect through state diagrams, activity diagrams, or sequence diagrams.

The *design model* at the modeling level describes a certain system and conforms to the design language, i.e., design model is modeled in the design language and every element of the model can be instantiated from the corresponding classes of the design language. Examples of design models used for the case study are described in Section 5.2.3 and presented in Figures 5.2 (class diagram) and 5.3 (state diagram). At the automation level we use a *code generator* that takes a design model as input and produces source *code* for the system via M2T transformation (see Section 2.1.4). Such transformation converts model elements to corresponding code statements. Finally, at the realization level this generated source code relies on a specific platform for its execution. In order to be able to access logs from the code running at an execution platform, we need a language for specifying runtime models, i.e., descriptive models. For that purpose we present in the descriptive perspective on the right-hand side of Figure 3.2 a *logging metamodel* — the so-called *observation language*. This observation language defines the syntax and semantics of the (event) logs we want to register from the running system. In particular, we derive the observation language from the operational semantics of the used design language. Figure 3.2 indicates this dependence at the metamodeling level with a ≪`refersTo`≫ stereotype. An example of the observation language and its derivation from the design language are described in Section 5.3 and presented in Figure 5.1.

The operational semantics give us an understanding, which elements of the system modeled in this design language change at runtime. For example, names of attributes are defined at design time and typically do not change at runtime, whereas their values

do. Another example would be the current state of the running system. At design time we define the states of a system, e.g., in form of a state diagram, however, at runtime the system or its components can be in only one current state. Changes of attribute values and current state of the system belong to operational semantics of the system modeling language. For any design language equipped with operational semantics we can derive an observation language. For this purpose we introduce the ≪observe≫ stereotype to annotate specific model elements related to operational semantics. This means that a code generator creates logging lines of code for all model elements annotated by this stereotype at any place the elements are changing during system operation. Therefore, the observation language influences the code generator by specifying which runtime changes should be logged in the future system. Thus, a running system produces logs in form of *observation models* which conform to the observation language. In fact, such observation models are not just streaming amorphous logs, but structured models, whose structure is specified already at design time. Generally, observation models provide abstractions of runtime phenomena in a descriptive way and can serve as a starting point to detect design errors and to implement new design solutions into a running system, which links them with the concept of M@RT (see Section 2.3).

The approach allows to avoid time-consuming preprocessing of massive data sets in order to extract valuable runtime information. The structured observation models with specific runtime semantics can be further used, e.g., in PM-tools. For this purpose we need to perform an M2M transformation between observation language and PM input format - eXtensible Event Stream (XES). It is important to keep observation language and XES apart, so that their semantics stays separated. In comparison with unstructured log output, observation models allow to filter and observe components, classes, or variables separately in order to create specific model profiles. An additional significant advantage of observation models is their re-usability for different kinds of tools after corresponding M2M transformations, e.g., runtime verification tools, simulation tools, monitoring and visualization or animation tools, etc (see Figure 3.2).

In this thesis, we especially highlight PM-tools as an instrument to create model profiles. These tools can take our prepared observation models as input and analyze them with established algorithms (see Section 2.2.3) producing a control flow model, e.g., in the form of a Petri net (see Section 2.2.2). During these analysis PM-tools go through runtime data consisting of several system runs, which have to be marked as process instances, or *cases*. Considering the diversity of cases PM-tools build in the end a big picture of the process with all the possible traces depending on the defined detalization level. The output of this analysis, i.e., a Petri net, forms a model profile for the particular prescriptive model or its part observed at runtime for a certain time period. Model profiles created for the following case study are presented in Section 5.3.

CHAPTER 4

Technical Realization

4.1 Overview

The technical realization of the approach was implemented within an experimental frame for further evaluation (cf. Chapter 5). Figure 4.1 presents a component diagram of this realization. Generally, component diagrams show the structural relationships between the components of a system. In our specific case, we can even observe a certain workflow in this component diagram, since it reflects the workflow of the EbMP-framework itself (cf. Figure 3.2). The first component `Enterprise Architect` in the right upper corner of Figure 4.1 is a modeling tool used to create prescriptive UML models for the future system (cf. Section 4.2). The next component `Vanilla Source` is an add-on for the `Enterprise Architect`. `Vanilla Source` generates Python code from the UML models. We extended this component with logging instrumentation, which allows us to generate logging code along with the system code (cf. Section 4.3).

For the case study, we chose a simple example of a `Trafficlight` system consisting of two traffic lights, one for cars and one for pedestrians (see Section 5.2.3). The code for this system, including the logging code, is completely generated from the prescriptive models. After generation the `Trafficlight` is deployed on the execution platform `RaspberryPi` (cf. Section 4.4). A logging component `LogClient` captures the logs produced by `Trafficlight` during execution, packs them in JavaScript Object Notation (JSON) and send it via POST requests to the `Microservice Observer` (cf. Section 4.5). The `Observer` runs on an `Application Server`, which should be available anytime the `Trafficlight` is running, otherwise the logs can not be recorded.

The same `Application Server` hosts the `Transformation OL2XES` implemented in Eclipse Modeling Framework (EMF)[1] and both source and target metamodels in Ecore dialect of UML (for details see Section 4.6). The source metamodel `ObservationLanguage.ecore`

[1] Eclipse Modeling Framework: https://www.eclipse.org/modeling/emf/

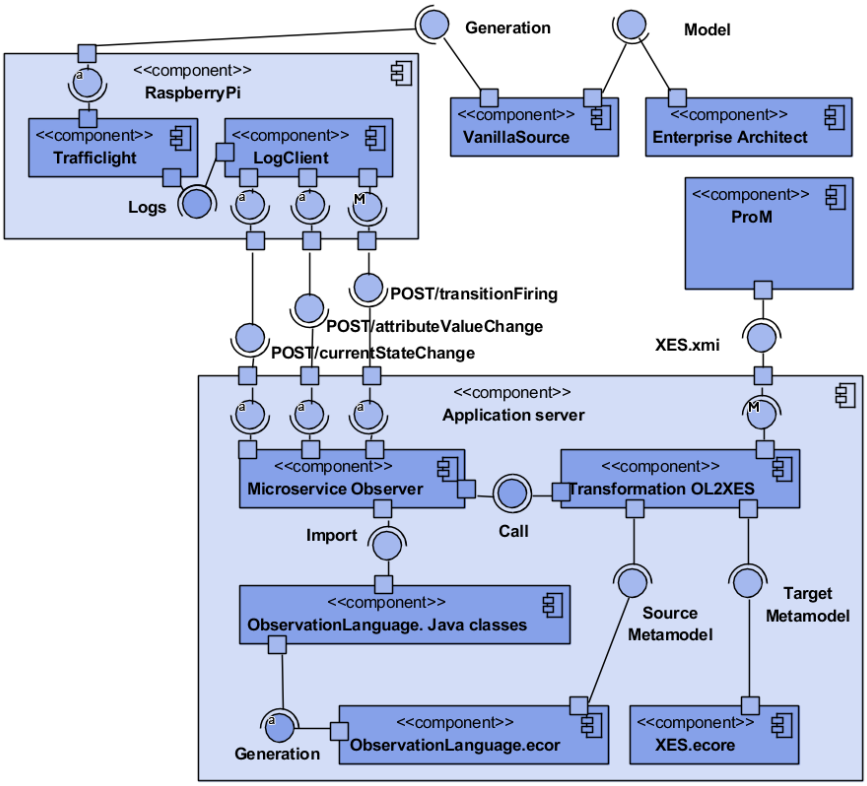


Figure 4.1: Component diagram of the technical realization of the EBMP-framework.

is also used to generate Java classes for the `Microservice Observer`. The microservice imports these classes and uses them for mapping between incoming JSON data and XML Metadata Interchange (XMI) models (cf. Figure 4.4). During the execution of `Microservice Observer` the captured logs are automatically transformed from `ObservationLanguage` to `XES`-Format. In this technical realization we chose the XES standard as target metamodel (cf. Section 4.6.3). It is important to store logs in a standardized way, so that the XES data can be exchanged between different commercial and open-source PM tools (for details see Section 4.7). In this thesis we use the open source software `ProM`, an extendable environment that supports a wide variety of PM techniques in the form of plug-ins. This tool generates descriptive models from the logs using the PM algorithms described in Section 2.2.3.

4.2 Modeling Tool and Code Generator

EA is used as a modeling tool in the context of this thesis. This advanced visual modeling tool supports a range of open industry standards for designing and modeling software and

business systems, including UML, Systems Modeling Language (SysML), BPMN[2], The Open Group Architecture Framework (TOGAF)[3] and many others [EAD]. EA has been developed by Sparx Systems since 2000. It grew into a powerful tool that covers a wide functional spectrum from system modeling and model simulation to the whole application development life cycle with its requirement engineering, project management, testing support, documentation and change management. Additionally to that, EA supports MDE and enables code generation of multiple platform-specific target solutions, e.g., C, java, or XSD, from platform independent models [EAD].

Code generation is an essential part of the technical realization of our approach. In the context of this thesis the code generator *VanillaSource* provided by our partner LieberLieber was used. VanillaSource is an EA extension that allows to generate Python code from UML models, constructed in EA. In our unifying framework EA takes the place of the modeling tool in prescriptive perspective, where the design model is created (cf. Figure 3.2). The code generator Vanilla Source resides under the EA on the automation level.

4.3 Logging Instrumentation

The C# source code of the VanillaSource code generator was extended, so that the logging code for the operational changes is produced automatically along with the system code. Such an insertion of additional code for monitoring purposes is called *instrumentation* [Ins]. For the first experiments with the prototype of our unifying framework the logging instrumentation of the system was limited to attribute value changes, current state changes, and transition firing. Detailed explanation of this selection is provided in the Section 5.3. The extension of the code generator to log attribute value changes provides logging lines at every place in code where a certain attribute changes its value. Similar to this procedure, logging is triggered as the system changes its state or fires a transition according to the state machine, which defines system's behavior at design time.

As mentioned in 1.1, the research and implementation for this thesis was conducted in the context of a scientific project at Christian Doppler Laboratory for Model-Integrated Smart Production (CDL-MINT). In particular, the logging instrumentation of the VanillaSource code generator lies outside the scope of this thesis. This instrumentation implemented as a part of the research project.

Simultaneously with the extension of the VanillaSource code generator the logging component named `LogClient` was implemented in the scope of this thesis. `LogClient` registers changes at runtime according to observational models. `LogClient` packs every runtime change, i.e., logging line, in JSON format and sends it via POST requests to the running MicroService Observer (cf. Section 4.5). In fact, logging instrumentation of the system code should be consistent with the classes and methods used by `LogClient` in

[2]OMG BPMN: http://www.bpmn.org/
[3]The TOGAF Standard: http://www.opengroup.org/subjectareas/enterprise/togaf

order to transfer runtime changes. Furthermore, the logging lines in JSON should be parsed as log entries of the observation language to be accepted by the MicroService (see Section 5.3).

4.4 Execution Platform

For implementing the unifying framework an execution platform is needed on the realization level, so that the generated code can be deployed and executed (cf. Figure 3.2). A single-board computer *Raspberry Pi* is used as such an execution platform for the generated Python code from VanillaSource. These computers are produced by the charity Raspberry Pi Foundation[4]. The aim of the Foundation is to promote the teaching of basic computer science at schools [Rasa]. Therefore, Raspberry Pi has all the essential components of a working computer and can be manufactured at a cost of only $15. Being the third best-selling general purpose computer since 2017, Raspberry Pi found its application beyond education [Rase]. Raspberry Pi can be used as a basic home automation system, weather station, web, or cloud server. Several Raspberry Pi's can be even combined into a computing cluster.

Figure 4.2 shows the board of a Raspberry Pi 2 Model B V1.1 and its components [Rasb]. This particular Raspberry Pi model was used for case study experiments in the context of this thesis (see Chapter 5). The Raspberry Pi 2 Model B uses a 32-bit 900MHz quad-core ARM Cortex-A7 CPU with 1 GB RAM and integrated graphics processing unit (GPU) VideoCore IV 3D [Rasc]. The board is powered via common Micro-USB port. Four USB 2.0 ports are available to connect diverse USB-devices, such as keyboards, mouses, flashcards, or WI-Fi adapters. A microSD card slot provides the possibility for on-board storage. The Linux-based operational system Raspbian, as well as user files, reside in the microSD card in this slot. A monitor for access to the operating system can be attached via a full HDMI port.

An attractive feature of a Raspberry Pi is the general-purpose input/output (GPIO) module. The Raspberry Pi 2 Model B has 40 GPIO pins, which have numbers to address them. These pins can be designated and controlled from software as input or output pins and used for various purposes [Rasd]. A GPIO pin designated as an output pin can be set to high (3V3) or low (0V). For example, a light-emitting diode (LED) connected to a pin can be turned on and off by sending corresponding signals from running software. A GPIO pin designated as an input pin can be read as high (3V3) or low (0V). For example, a button connected to a pin can send switching signals to running software.

Since this technical implementation is made for Python code generation and execution, a special GPIO Python library `RPi.GPIO` should be included into the running software to handle Raspberry Pi's GPIO pins. The design model and generated code used in context of this thesis can be adapted to be deployed on other single-board computers with GPIO

[4]Raspberry Pi Foundation: https://www.raspberrypi.org/

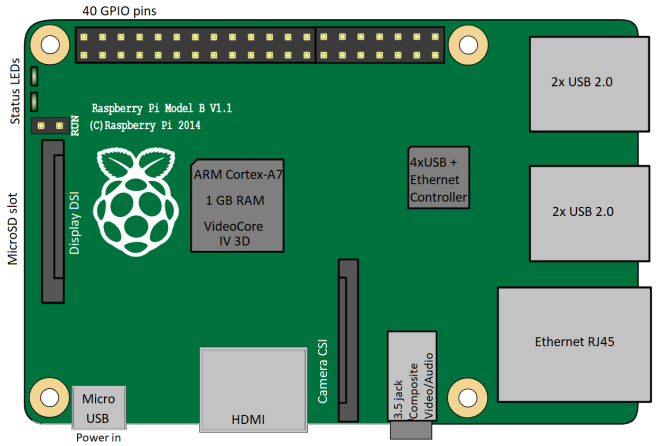


Figure 4.2: Raspberry Pi Components [Rasb]

module, such as Asus Tinker Board[5] or Banana Pi BPI-M2[6]. The critical configuration aspect that has to be considered is the addressed numbers of the input/output pins.

4.5 MicroService

A microservice is a light-weight, discrete, network-connected component [Mic]. The most important task of a microservice is *"to do one thing and do it well"* [Kra15]. The microservice named *Observer* implements a Representational state transfer (REST)-API with publicly exposed endpoints that accept log-messages from a running system. The only Observer's task is to persist logs.

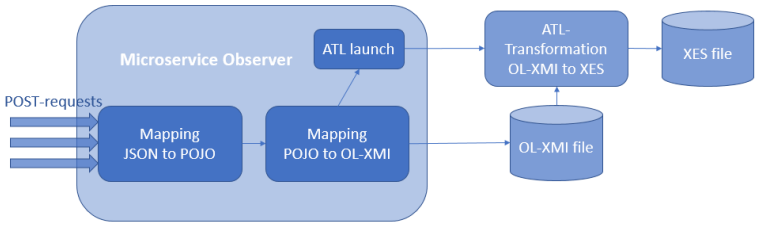


Figure 4.3: Microservice Observer.

Figure 4.3 shows the basic control and data flow inside Observer. While running Observer opens API-endpoints for POST-requests from a running system. As mentioned in Section 4.3, both requests and API-endpoints conform to the observation language (cf. Section 5.3), meaning every POST-request delivers an inserted JSON that is expected by the

[5]Tinker Board: https://www.asus.com/uk/Single-board-Computer/TINKER-BOARD/
[6]Banana Pi: http://www.banana-pi.org/

endpoint. The microservice maps this JSON to a Plain Old Java Object (POJO) (cf. upper left part of the Figure 4.4). After this the microservice maps the POJO to the XMI format (cf. lower right part of the Figure 4.4). This can be easily implemented, since the Observer reuses the generated code from the Ecore model of the observation language (cf. Section 4.6). The produced XMI file is used as input for an ATLAS Transformation Language (ATL) transformation, which is called from Observer by the component named ATLLauncher. This ATL-transformation is not a part of of the Observer and is therefore explained in details in Section 4.6. The transformation produces an XES file that conforms to the XES standard, and, thus, can be used as input for PM tool ProM (cf. Section 4.7). For every single POST-request this control and data flow is repeated and both XMI file and XES file are updated.

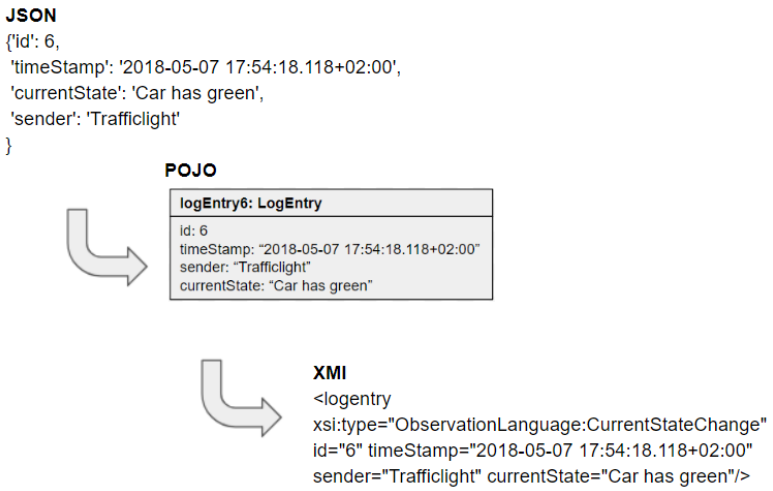


Figure 4.4: Mapping JSON to POJO and POJO to XMI. Example from the case study.

4.6 Transformations

4.6.1 Overview

Logs persisted by Observer in XMI format can not be directly used as ProM input, since they don't conform to the XES standard, which is expected by ProM. Therefore, a transformation from the XMI format to a XES conform file is needed. Such a transformation was implemented in EMF, a modeling framework and code generation facility, which allows to build tools and other applications based on a structured data model [EMF]. Ecore is the core EMF metamodel for creating other models and metamodels. Ecore, being a UML dialect, is even defined in terms of itself, which makes it its own metamodel. In a basic EMF workflow a user models a class diagram as an Ecore model and generates java code of a complete entity model for an application. In this technical realization the entity model for Observer was created in such a way from the observation language metamodel shown in Figure 4.6.

Ecore models are also used as metamodels for ATL transformations: a domain-specific language for specifying M2M transformations [JABK08] (cf. Section 2.1.4). In a nutshell, ATL provides ways to produce a set of target models from a set of source models. Both ATL language and toolkit are developed on top of the Eclipse platform by Obeo[7] and INRIA[8]. ATL was inspired by the OMG QVT[9] requirements and builds upon the OCL[10] formalism [JABK08], providing both declarative and imperative constructs [JABK08].

The transformation from *observation language* to *XES* is an exogenous out-place transformation that transforms models in two different languages and creates the target model from scratch (for transformation classification see Section 2.1.4). Figure 4.5 shows this concrete transformation based on the abstract transformation pattern, presented in the Section 2.1.4, Figure 2.4. In this pattern the source model *Log.xmi* conforming to the metamodel *ObservationLanguage.ecore* is transformed into a set of target models conforming to the metamodel *XES.ecore* according to a set of transformation definitions written in the ATL language. The transformations are enabled by the EMF Virtual Machine. The transformation definition is a model conforming to the ATL metamodel. All metamodels conform to the Meta-Object Facility (MOF) [11].

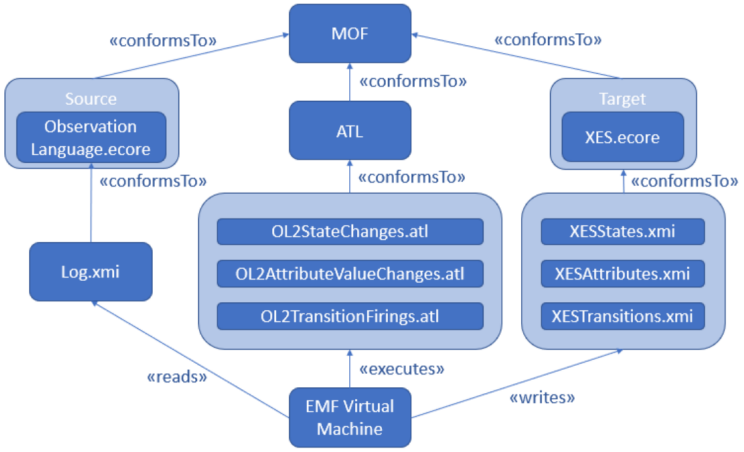


Figure 4.5: Transformation "Observation Language To XES".

4.6.2 Source metamodel *Observation language*

The metamodel of the observation language as Ecore class diagram is presented in Figure 4.6. This metamodel describes, how the transformation source model, i.e., log, is structured. Basically, the log consists of process instances, which, in their turn, consist

[7] Obeo: https://www.obeo.fr/
[8] French Institute for Research in Computer Science and Automation: https://www.inria.fr/
[9] Query/View/Transformation, OMG standard for performing model transformations: https://www.omg.org/spec/QVT/About-QVT/
[10] Object Constraint Language: www.omg.org/spec/OCL/
[11] MetaObject Facility Specification: http://www.omg.org/mof/

of log entries. The log entry entity is an abstract class with three possible instantiations, which are current state change, attribute value change and transition firing. As stated in Section 4.1, this technical realization is done within an experimental frame. Thus, this observation language is bound to the design language used in the use case and is limited to the operational semantics of this design language. The detailed explanation of observation language and its structure is provided in Section 5.3.

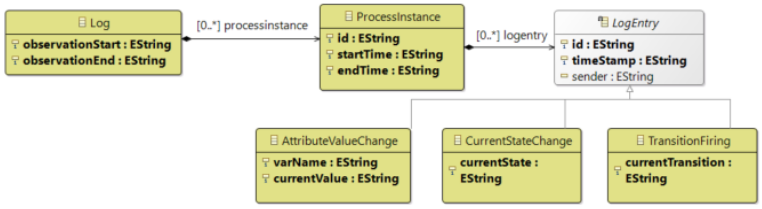


Figure 4.6: Source metamodel Observation Language. Ecore class diagram.

4.6.3 Target metamodel *XES*

The XES metamodel originates from the IEEE Standard for eXtensible Event Stream (XES) for Achieving Interoperability in Event Logs and Event Streams [XES16]. The purpose of this standard is to specify a generally acknowledged Extensible Markup Language (XML) format for the interchange of event data between information systems producing this data and analysis tools. To enable the transfer of event-driven data in a unified manner, the standard includes a XML Schema[12] describing the structure of a XES instance. This XML Schema was imported in EMF resulting in an Ecore model shown in Figure 4.7.

Additionally to the syntax defined through the XML Schema, the standard fixes the semantics of the event data [XES16]. The 4 major semantic components building the event stream concept are coloured dark yellow in Figure 4.7. `Log` component represents information related to a specific process and contains a collection of traces. `Trace` component represents a single execution (or enactment) of this specific process and contains a list of events related to this single execution. `Event` component represents an atomic part of the observed process. All these three components are of `AttributableType` meaning they are described through `Attributes` with keys and values. E.g., a login event has an attribute with the key *name* and value *login*. There are six core types: String, Date, Int, Float, ID, and Boolean.

In XES it is possible to declare particular attributes on trace and event levels as mandatory. For this purpose the log has two collections of `Globals`, i.e., global attributes: one for the traces and one for the events [vdA16]. Apart from global attributes the standard introduces a concept of so-called `Extensions`, which attach semantics for such attributes and provide points of reference for their interpretation. For example, the declared

[12]XSD (XML Schema Definition): http://www.xes-standard.org/xes-ieee-1849-2016.xsd

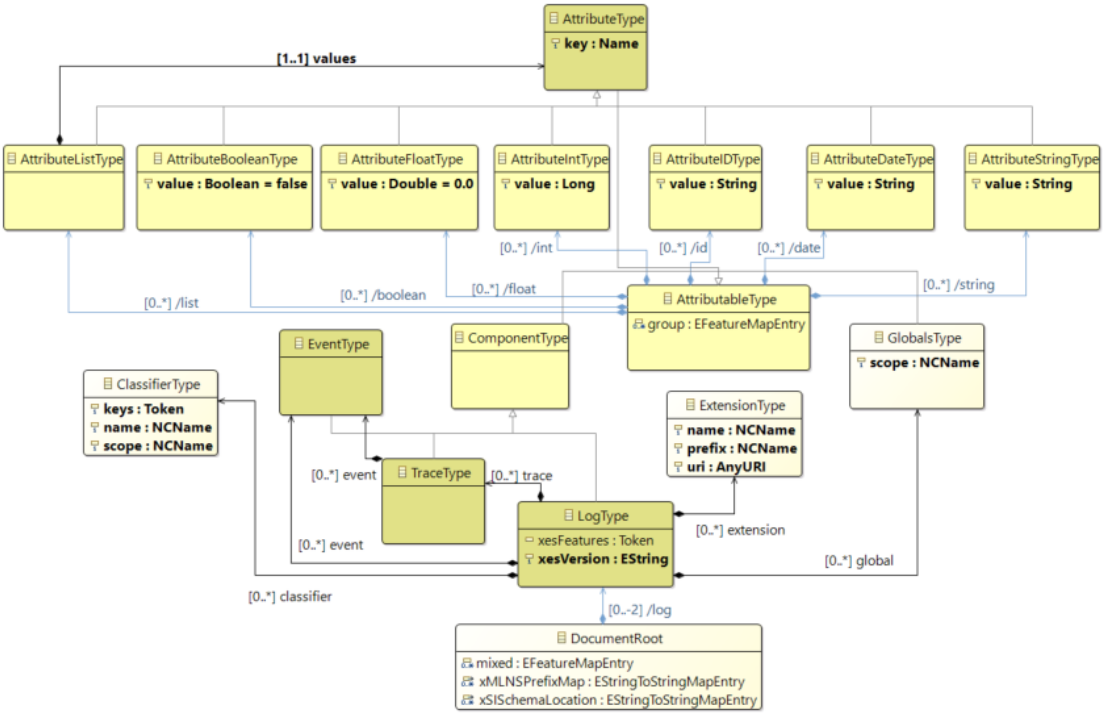


Figure 4.7: Target metamodel XES. Ecore class diagram.

extension *Time* with the prefix *time* indicates, that all the attributes, whose key starts with this prefix, should be interpreted as time-related attributes. Such attributes inform the analysis tools about the moment, when the event took place.

Another important XES concept is `Classifier`. The classifier's task is to assign an identity to each event. That makes an event comparable to others via their assigned identity [XES16]. For example, the classifier Activity classifies events based on the *concept:name* attribute. Thus, an analyzing tool identifies events with key *concept:name* and the value *green blink pedestrian* as different events referring to the same activity. In this way classifiers allow to recognize, e.g., repeating activities.

4.6.4 ATL transformations

The logs in observation language contain all types of log entries, i.e., attribute value changes, current state changes and transition firings. For model profiling we need to observe these aspects separately. Therefore, these different log entries types should be isolated from each other during the `ObservationLanguage2XES` transformation. Hence, three separate ATL transformations were implemented (see also Figure 4.5):

- `OL2StateChanges.atl` transforms

  - OL!Log[13] to XES!log,
  - OL!ProcessInstance to XES!trace,
  - and OL!CurrentStateChange to XES!event.

- OL2AttributeValueChanges.atl transforms
  - OL!Log to XES!log,
  - OL.ProcessInstance to XES!trace,
  - and OL!AttributeValueChange to XES!event.

- OL2TransitionFirings.atl transforms
  - OL!Log to XES!log,
  - OL!ProcessInstance to XES!trace,
  - and OL!TransitionFiring to XES!event.

In all three transformations OL!Log is transformed to XES!log, which are, as follows from the name, the same semantic units. The same logic applies to OL!ProcessInstance and XES!trace, which are also very similar semantic units, since both represent one end-to-end execution of the process. The three transformations differentiate though in the way they transform OL!LogEntry to XES!event. Every transformation transforms one of the instantiations of the abstract class OL!LogEntry to XES!event (cf. Figure 4.6). From the semantic point of view both log entry and event are defined and globally unique process steps (cf. Section **??**). A log entry can be considered as a way of registering an event. Therefore, the semantics of events is applicable to log entries as well.

A transformation definition written in ATL language is composed of a header section, an import section, a set of transformation rules, and a set of functions called helpers. The *header section* starts with the name of the module and declaration of the source and target models as variables typed by their metamodels. The keyword *create* indicates the target model, whereas the keyword *from* indicated the source model. The *import section* specifies the paths of the metamodels.

Matched transformation rules are the basic constructs in ATL to express transformation logic. A matched rule consists of a source pattern and a target pattern, i.e., it defines, in which way elements of the source model should be transformed into elements of the target model. Algorithm 4.1 shows the complete code of the rule StateChange2Event as an example. The rule implements the logic for transforming a current state change log entry (OL!CurrentStateChange) to an event (XES!EventType) and applies to all log entries of type CurrentStateChange in the source model (Log.xmi).

[13]Here and in the following text the prefix OL! indicates that an element belongs to the observation language, whereas the prefix XES! shows a reference to the XES metamodel.

Algorithm 4.1: CurrentStateChange to Event Transformation

```
1 rule CurrentStateChange2Event
2 from currentStateChange : OL!CurrentStateChange
3 to event : XES!EventType ( string <- id, string <- resource, string <- eventName,
    date <- timestamp, string <-activity ),
4       resource : XES!AttributeStringType ( key <- 'org:resource', value <-
    currentStateChange.sender ),
5       eventName : XES!AttributeStringType ( key <- 'concept:name', value <-
    currentStateChange.currentState ),
6       id : XES!AttributeStringType ( key <- 'identity:id', value <-
    currentStateChange.id ),
7       timestamp : XES!AttributeDateType ( key <- 'time:timestamp', value <-
    currentStateChange.timeStamp.replaceAll(' ', 'T') ),
8       activity : XES!AttributeStringType ( key <- 'Activity', value <-
    currentStateChange.currentState )
```

In Algorithm 4.1 the first line defines the rule's name. The second line declares the variable `currentStateChange` for the source element `OL!CurrentStateChange`, whereas the third line declares the variable `event` for the target element `XES!EventType`. The lines 4-8 specify bindings from the attributes of the `currentStateChange` to the attributes of the `event`. The symbol "<-" is used to initialize the target attributes on the left from the source attributes on the right.

The first attribute `resource` in the line 4 corresponds to the `sender` of the log entry `currentStateChange`. This allows to align the event to the specific resource, i.e., sender, during PM. The second attribute `eventName` in the line 5 corresponds to the `currentState`, which the system reached by this state change. The attribute `id` takes the value of the log entry id to keep it unique, whereas the attribute `timestamp` allows to keep the order of the events. The timestamp is an important concept in PM, since it signifies the order, in which the events have been observed, allowing to reconstruct the process. The attribute `activity` is a classifier that assigns an identity to the event making it comparable to othe events (cf. Section 4.6.3). After this mapping the single attributes are bound to the `event` in line 3 as attributes of types `string` or `date`.

The last section of the ATL transformation is *helpers*. A helper can be specified on a source metamodel type or on an Object Constraint Language (OCL) type. The concept of helper comes from the OCL specification [OCL]. Helper `hasThisSender` defined in the `LogEntry` context in `OL2CurrentStateChanges.atl` and `OL2TransitionFirings.atl` transformations allows to filter events belonging to a certain component that sent them. Similarly the helper `isThisAttribute` limits the event stream to a certain attribute in `OL2AttributeValueChanges.atl` to observe its behaviour and values at runtime. This data flow is presented in Figure 4.8.

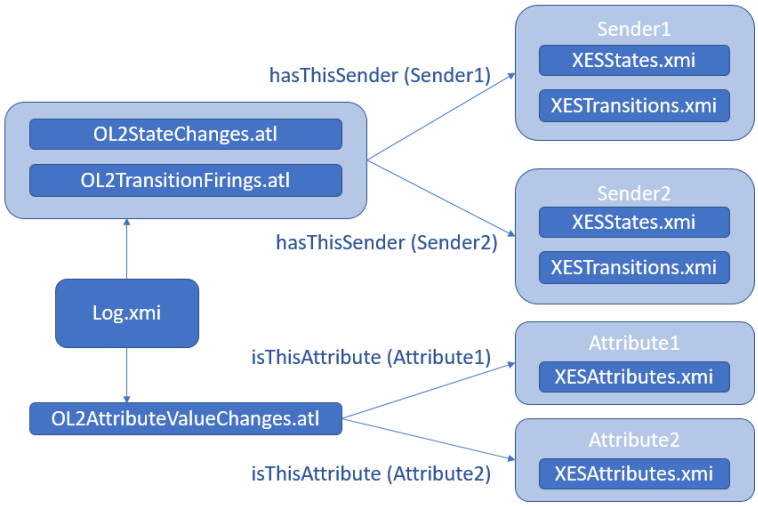


Figure 4.8: Transformations with helpers.

4.7 Process Mining Tool

A PM tool is used in the descriptive perspective of the unifying framework. This tool takes event logs as an input, analyzes them, and produces execution-based model profiles. This technical realization considers *ProM*[14] being the most suitable tool. ProM is open-source software developed by the Process Mining Group[15] at Eindhoven Technical University[16]. Hence, significant part of the academic research in PM field is made using and extending ProM. Apart from that, commercial PM tools, like Celonis[17], Disco[18], or recent Minit[19] and myInventio[20], are based on algorithms first implemented and evolved in ProM [vdA16].

ProM framework consists of a core program and a set of plug-ins[Pro]. The core serves as a common basis for the plug-ins and provides the infrastructural functions, such as import, plug-in search and enabling, and visualization of results. Figure 4.9 shows the core user interface. On the left side a user can choose an input among imported files, in the central part there is a list of installed plug-ins, on the right side the program shows, what output generates a certain plug-in. As described in the Section 4.6, ProM can load XES, MXML, and CSV files. As a result of analysis ProM produces formal high-level process models, i.e., end-to-end models allowing for choices, concurrency, loops, etc. This includes amongt others BPMN models, EPC models, UML activity diagrams, Petri nets,

[14]ProM Tools: http://www.promtools.org/doku.php
[15]Process Mining Group: http://www.processmining.org/
[16]Eindhoven Technical University: https://www.tue.nl/
[17]Celonis: https://www.celonis.com/
[18]Disco: https://fluxicon.com/disco/
[19]Minit: https://www.minit.io/
[20]myInventio: https://www.my-invenio.com/

and process trees. Figure 4.10 presents ProM user interface showing a mining outcome of the Integer Linear Programming (ILP) Miner plug-in in form of a Petri net [vdA16].

Figure 4.9: ProM 6 user interface [vdA16].

Plug-ins provide the whole variety of ProM functionality. PM techniques and algorithms can be installed as plug-ins additionally to the core. The first functional version of the ProM (ProM 1.1) was released in 2004 and contained 29 plug-ins. ProM 6 released in 2010 was the first version based on the new architecture and could import XES additionally to MXML. The current ProM version 6.7[21] is released in 2017 and is XES-certified, which means ProM 6.7 correctly imports and exports any XES file. In the recent years ProM grew to over 1500 plug-ins supported by different versions. Dozens of process discovery algorithms are implemented as ProM plug-ins. Next to the ILP miner shown in Figure 4.10 and the α-algorithm, also heuristic mining, fuzzy mining, and various forms of inductive mining are supported [vdA16] (cf. Section 2.2.3).

Figure 4.11 shows another example of an advanced plug-in called Visual Inductive Miner. This plug-in produces a sound process model and is able to process large event logs with a lot of noise. Nevertheless, the miner can provide, if needed, perfects fitness. Mining results can be converted to Petri nets, EPCs, statecharts and BPMN models. Additionally, the Visual Inductive Miner supports bottleneck analysis and outlier detection and is able to show replay of the mined process [vdA16].

The ProM core is distributed under GNU Public License (GPL)[22] open source license, which means that the installation is free, but any software that integrates the core must be distributed under GPL license. Plug-ins are distributed under Lesser GNU Public License (L-GPL)[23], meaning that it is allowed to distribute software that uses these plug-ins (unchanged) using one's own license [Pro]. The ProM java source code is

[21] ProM 6.7: http://www.promtools.org/doku.php?id=prom67
[22] GNU General Public License: https://www.gnu.org/licenses/gpl-3.0.en.html
[23] GNU Lesser General Public License: https://www.gnu.org/licenses/lgpl-3.0.en.html

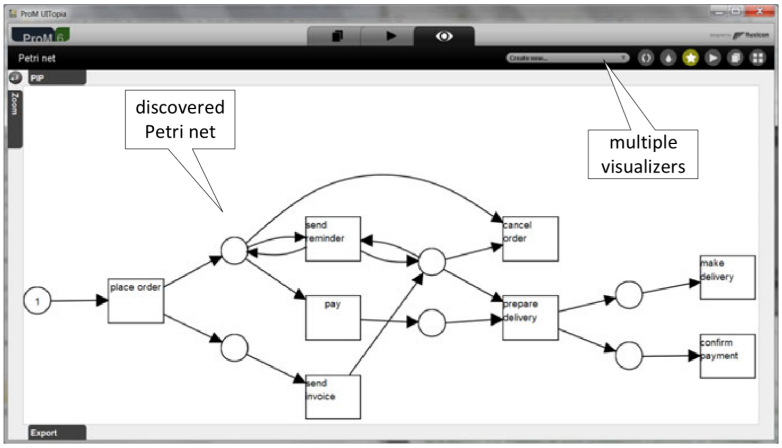


Figure 4.10: Discovered Petri net [vdA16].

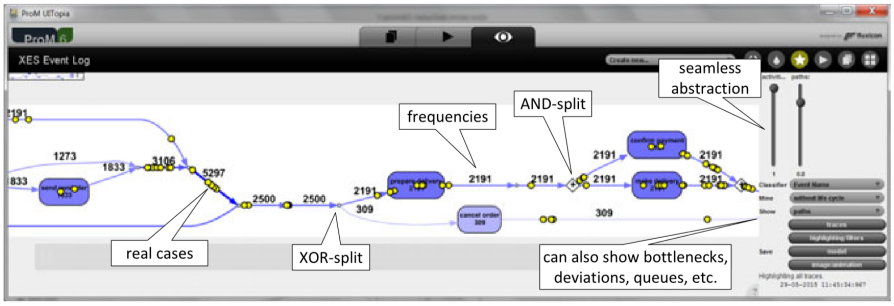


Figure 4.11: Visual inductive miner [vdA16].

publicly available and can be used and integrated in any project that doesn't violate the corresponding licenses.

In this technical realization ProM is used as a stand-alone not embedded application that requires manual input and user interaction. Since ProM is a open source project, in the following future work it can be built inside an automated workflow, where collected logs will be transformed in XES and directly fed into ProM for instant analysis.

CHAPTER 5

Evaluation: Case Study

5.1 Overview

This chapter presents a case study for the evaluation of our approach, the *Execution-based Model Profiling (EbMP)*. The case study is both explanatory and exploratory as per guidelines of Runeson and Höst [RH09]. The explanatory part provides answers to the research questions formulated in Section 1.2. The research interest here is to equip the MDE-based system under study in such a way, so that its operational data can be derived from the operational semantics of the system's design language. The exploratory part detects further possible research directions and formulates research questions for future work.

5.2 Design

This Section defines requirements for the case and provides description of the case, including definition of the modeling language, description of the models and setup of the experiments.

5.2.1 Requirements

As an appropriate input to this case study, we require a PAIS (cf. Section 2.2.1) to be able to capture its behavioral aspects on the basis of event data. Additionally to that the system should be operational within the implemented technical realization of unifying framework. Based on these two general requirements the following specific requirements were formulated:

- *Requirement 1.* The system should be modeled in an executable modeling language with clear syntax and operational semantics, which allows to observe system

behavior.

- *Requirement 2.* It should be possible to generate the system code from the model with the available code generator (cf. Section 4.2).
- *Requirement 3.* The system should be deployable on the chosen execution platform for conducting experiments (cf. Section 4.4)

The upcoming Section 5.3 gives a definition of a modeling language, which fulfills the first requirement. The second and the third requirements are covered by the Sections 5.2.3 and 5.2.4, where the system and its two models are described.

5.2.2 Modeling Language

For the first experiments with the unifying framework it was necessary to consider, which part of operational semantics will be registered for further analysis. Generally, a PAIS has two different aspects: the *static aspect* which describes the main ingredients of the system to be modeled, i.e., its entities and their relationships, and the *dynamic aspect* which describes the behavior of these ingredients in terms of events and interactions that may occur among them, as well as changes of the overall state of the system. Therefore, we defined the language *Class/State Charts (CSC)*, that covers these two aspects of the system to be modeled.

CSC is a small subset of UML, consisting of simpler versions of UML class diagrams representing the static aspect and UML state diagrams representing the dynamic one. Operational semantics of CSC class diagrams includes attribute value changes, whereas operational semantics of CSC state diagrams includes current state changes and transition firing. Therefore, as mentioned in 4.2, the observable operational changes of the system modeled in CSC are narrowed to attribute value changes, current state changes and transition firing. In the future work this set can be extended by operation/method calls.

Figure 5.1 presents the two main parts of CSC, which are *design language* and *observation language*. The design language defines, how structure and behaviour of the running system should be modeled, whereas the observation language defines, in what form the observations of the system, i.e., logs, should be registered.

The CSC design language. The upper section of the Figure 5.1 shows the CSC design language. Similar to UML, the design language allows to model the static aspect via `Classes` and their `Attributes`, which have names and values. The dynamic aspect can be modeled as a `StateMachine` consisting of `States` and `Transitions`. Incoming transitions distinguish a predecessor state, whereas outgoing ones point out a successor. A transition can be triggered by an `Event`. Both states and transitions can call `Operations`.

The CSC observation language. The observation language, derived from the design language , is shown in the lower section of Figure 5.1. In a nutshell, the observation

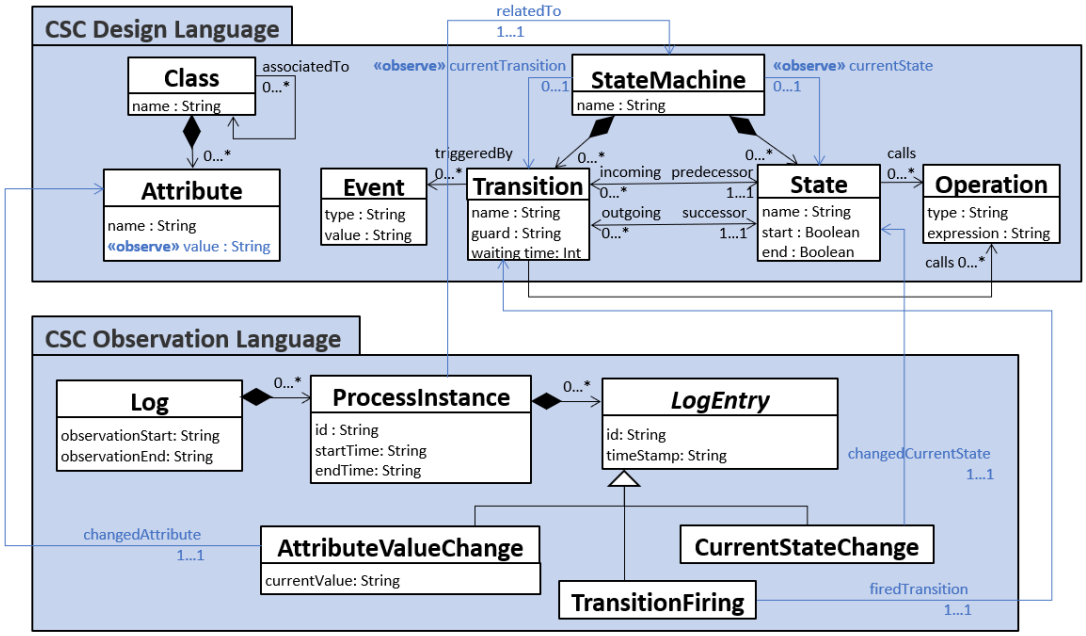


Figure 5.1: Design language and observation language.

language shows the structure of the collected event logs. The class `Log` represents a logging session of a certain running software system, modeled in design language, with a registered `observationStart` and an `observationEnd`. The class `Log` consists of `ProcessInstances` related to the `StateMachine`. It means that one run of a state machine from the start state to the end state is logged as one process instance. 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). The specific types of the generalized `LogEntry` depend on the operational semantics of the design language and include `AttributeValueChange`, a `CurrentStateChange`, or a `TransitionFiring`.

In the Figure 5.1 all the operational semantics-related elements, i.e., everything that is changing in a system at runtime, are coloured blue. According to the unifying framework (cf. Section 3.3), these elements are marked with the ≪`observe`≫ stereotype. In the class diagram the only parameter changing at runtime is the attribute's value. This change has an association with `AttributeValueChange` type of `LogEntry`. In the behavioural aspect the changes of the `currentState` and the `currentTransition` of the `StateMachine` happen also at runtime. These changes are associated with the `LogEntry` types `CurrentStateChange` and `TransitionFiring`. This set of changes, when registered and persisted as a log, gives an essential outline of the system's behaviour. Therefore, the Requirement 1 (cf. Section 5.2.1) is fulfilled: the described design language is an executable modeling language with clear syntax and operational

semantics, which is captured by the observation language.

The next possible extension of the observation language would be to observe and register operation calls, including operation entry and operation exit. This elements of operational semantics lie outside of scope of this thesis and can be considered as future research.

5.2.3 TrafficLight System

The PAIS chosen for the case study is called *TrafficLight*. This IoT-system implements an operation process of two traffic lights. The first traffic light controls the cars traffic, and the second one controls pedestrians traffic. The system is modeled in EA and it's python code can be generated by Vanilla Source code generator (cf. Section 4.2). Thus, the complete code for the system can be generated via one click and is ready to be deployed on the execution platform, i.e., RaspberryPi (cf. Section 4.4).

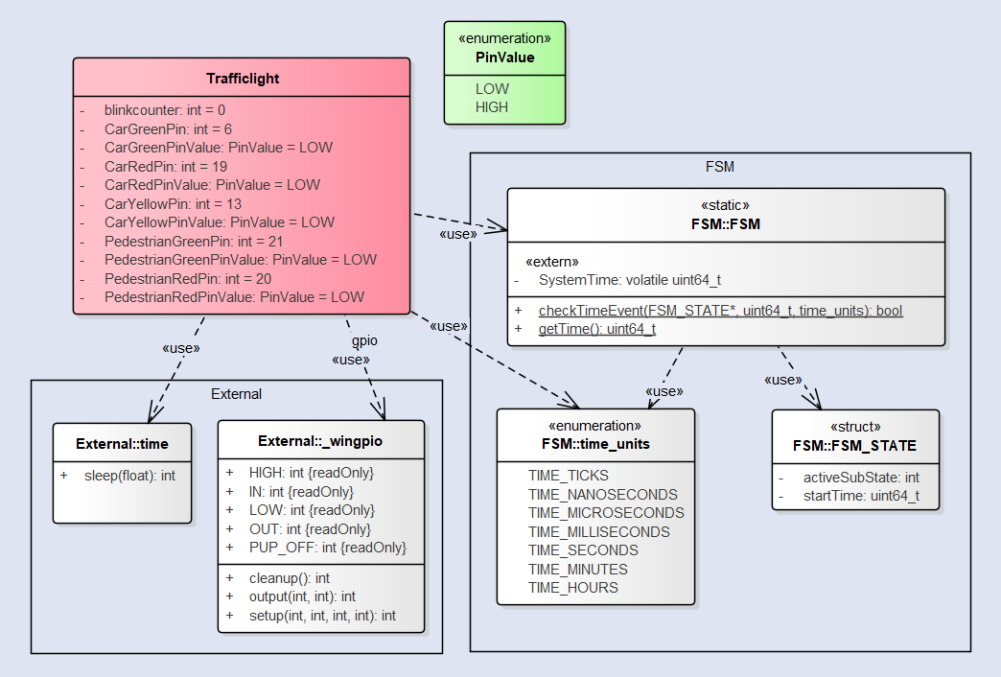


Figure 5.2: TrafficLight Class Diagram.

TrafficLight is modeled in CSC, so its structure is defined by a class diagram and its behaviour is specified as a state diagram. These two diagrams are prescriptive models of the TrafficLight system (cf. Section 3.1). The Figure 5.2 shows the class diagram, as it is modeled in EA. The TrafficLight system related class is painted red, the auxiliary classes, which are needed for the state machine and GPIO support, are grey and an enumeration of attribute's values is green.

In fact, the TrafficLight system has the only class `Trafficlight`, which defines its whole structure. TrafficLight has three light for cars, namely green, red and

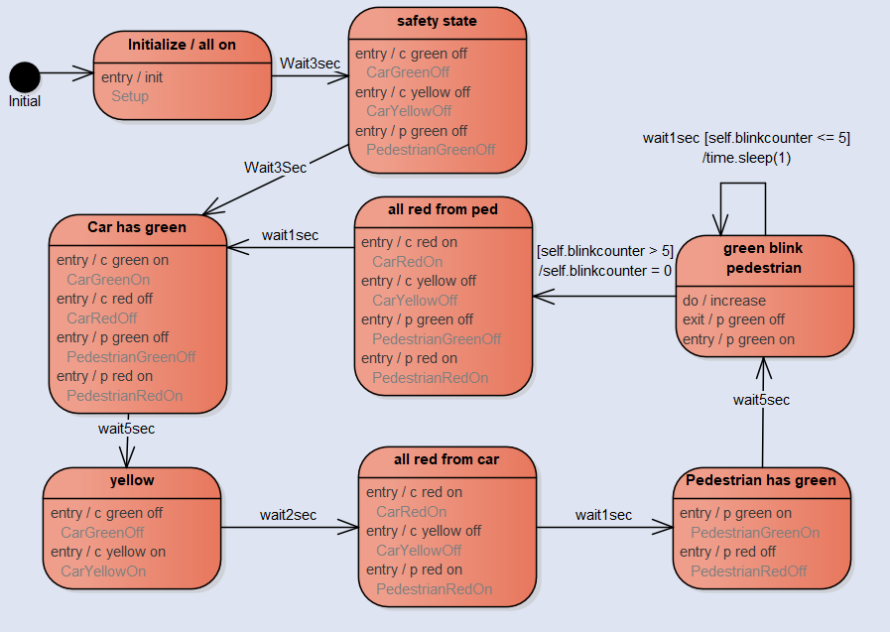


Figure 5.3: TrafficLight. State Diagram.

yellow, and two lights for pedestrians, namely green and red. The lights are connected to a RaspberryPi via GPIO pins (cf. Section 4.4). The attrubutes for pins, e.g., `Trafficlight.CarGreenPin: int = 6`, specify, to which output pin TrafficLight must send a signal to turn the light on or off. The attrubutes for pin values, e.g., `Trafficlight.CarGreenPinValue: PinValue = LOW`, register, which signal came as last. There are only two possible signals specified in the enumeration `PinValue`, which are `HIGH` and `LOW`. The initial pin values are `LOW`. The purpose of the attribute `Trafficlight.blinkcounter` is explained together with the TrafficLight state diagram.

The package finite-state machine (`FSM`) contains classes which define the structure of an abstract state machine having an active state and the start time when the active state was triggered. The package `External` is responsible for communication with RaspberryPi's GPIO. These grey auxiliary classes provide an infrastructure for TrafficLight and make it a functional state machine-based system, which is deployable on RaspberryPi. Any other system, whose behaviour can be defined as a state machine, can be associated with these auxiliary classes instead of `Trafficlight`, so that complete system's code can be generated.

Figure 5.3 presents the state diagram, as it is modeled in EA. Operations, which send signals to the pins, are specified for every state, including their calling time point (`entry`, `exit`, or `do`). Every transition has a guard, e.g. `wait3sec`, which prescribes, how long the system should stay in the current state. In the first initialization state all the lights are turned on. After that system switches to `safety state` when only red lights are

on for both cars and pedestrians. With the next transition car traffic light turns green while pedestrian light stays red and the real working cycle of the system begins. In the next state cars have yellow for a while, then both traffic lights turns red simultaneously. In the next step green light turns on for pedestrians while cars are having red. After green blinking for pedestrians system switches to the state with all lights red and the cycle starts all over again. The attribute `Trafficlight.blinkcounter` is used here to limit the blinking cycle to 5 iterations.

This rather basic TrafficLight system is used to achieve first results and to check the implemented technological chain from prescriptive models to descriptive models, i.e., mined Petri Nets. For further experiments this basic system was refactored into an object oriented system (cf. next Section 5.2.4) with identical behaviour, so that direct comparison of mined outcomes is possible.

5.2.4 Object oriented TrafficLight

This Section presents the refactored object oriented TrafficLight system called *TrafficLightOO*. In Figure 5.4 the class `Trafficlight` of the initial system is split into three separate classes, which are `ControlTrafficLight`, `PedestrianTrafficlight`, and `CarTrafficlight`.

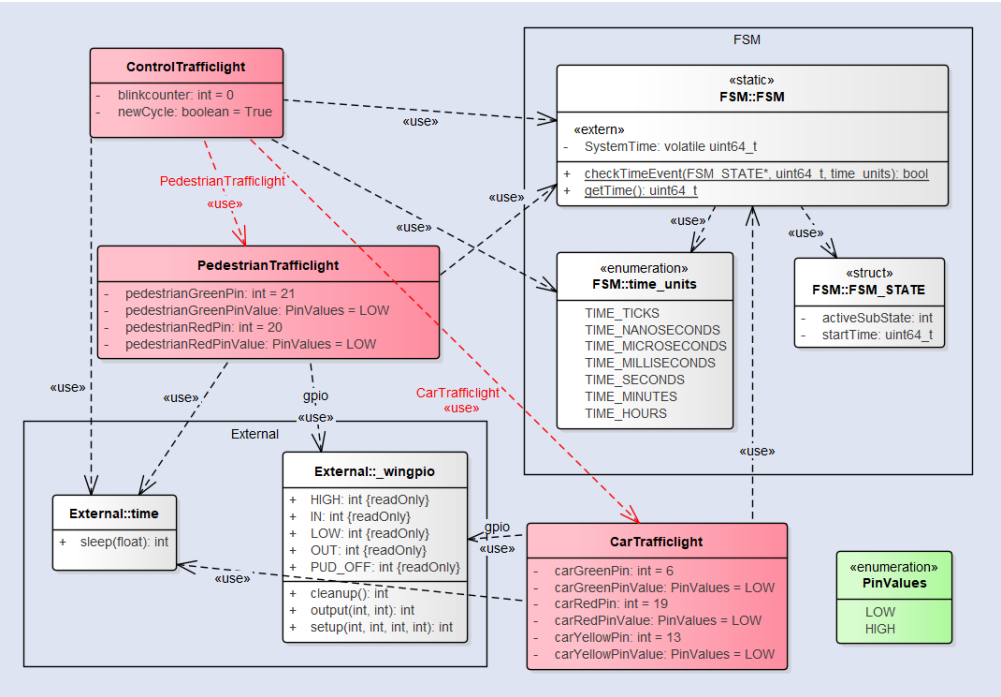


Figure 5.4: Object oriented TrafficLight. Class Diagram.

As follows from the name, the class `ControlTrafficLight` takes control over car and

pedestrian traffic lights. These three classes are highlighted red and represent the whole system's structure. The object oriented approach increases system modularity and allows to easily arrange different combinations with several traffic lights of different types. The attributes stayed the same and were distributed to corresponding classes. The packages `FSM` and `External` stay unchanged and provide the same infrastructure, as described in Section 5.2.3.

In TrafficLightOO there is three state diagrams prescribing behaviour of the three classes. The state diagram for `ControlTrafficLight` (cf. Figure 5.5) shows behaviour of the whole system. The states and their arrangement are identical to the initial TrafficLight system (cf. Figure 5.3). States of this state diagram don't send signals to the pins, the way the initial system does, but to the state machines controlling pedestrian and car traffic lights.

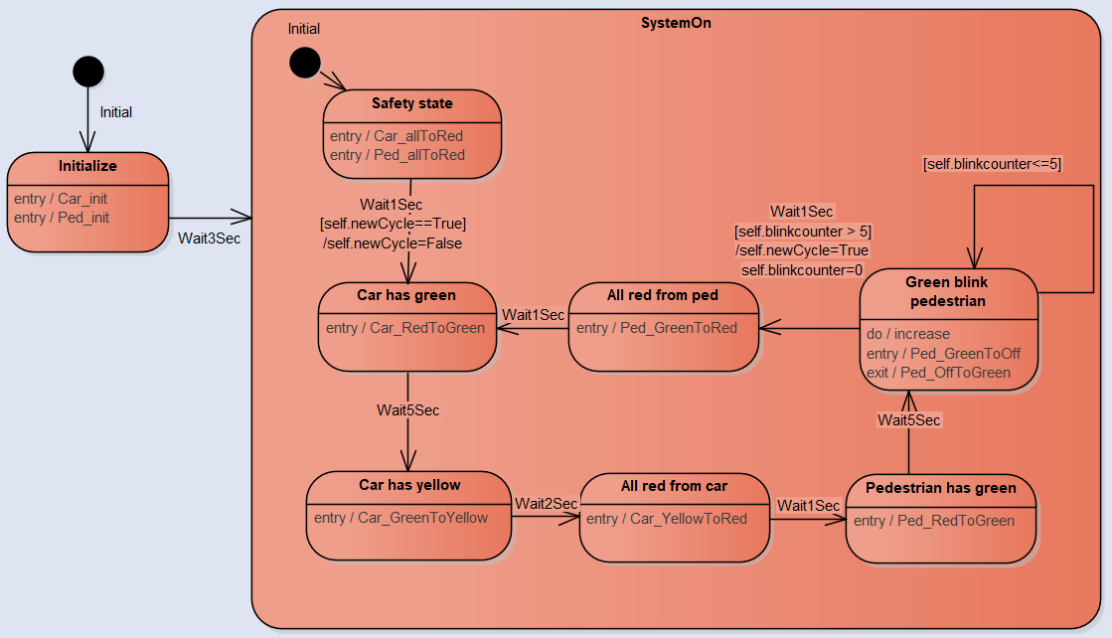
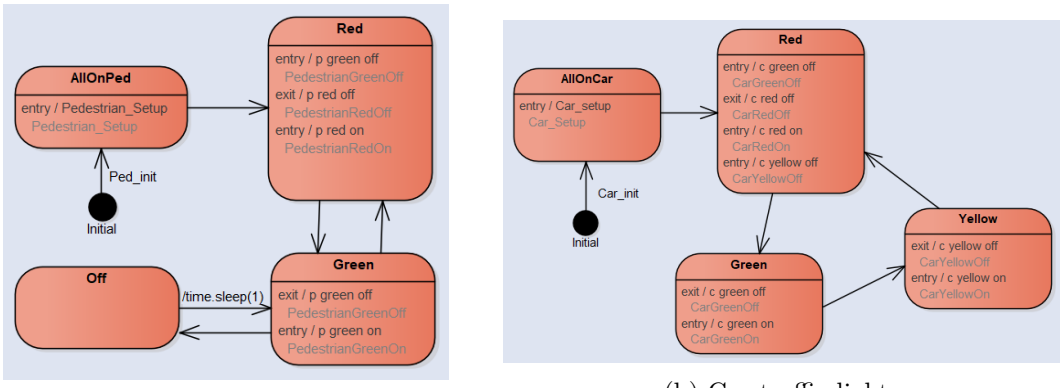


Figure 5.5: Object oriented TrafficLight. State Diagram. Control.

Figure 5.6 shows, which states the pedestrian and car traffic lights can have. These states are modeled independent from the control state machine and can themselves call operations, which send signals to the GPIO pins.

Such model-driven systems are easily accessible for engineers without software engineering background, since modularity allows to combine and reuse components and a code generator provides deployable code. Engineers can change the model, e.g., for a new intersection with a new, different traffic lights configuration, and get a ready system without writing code.



(a) Pedestrian traffic light.

(b) Car traffic light.

Figure 5.6: Object oriented TrafficLight. State Diagrams.

5.2.5 Setup and Experiments

This section defines the setup and groups of experiments for data collection and the further empiric evaluation and assessment of this data. Data sources for the data collection are the subjects under study, namely the initial Trafficlight (cf. Section 5.2.3) and the object oriented TrafficLightOO (cf. Section 5.2.4) systems.

For the experiments both initial and object oriented systems were generated from their prescriptive models and deployed on the Raspberry Pi execution platform to simulate real systems and produce event logs. These event logs were captured by the Observer microservice and stored as qualitative data ready to be processed by the ProM tool. In this tool the event logs were analyzed by various plug-ins. The result of every experiment is a descriptive model of the system under study created for a certain period of its operation, i.e., a *model profile*.

Preliminary we defined four groups of experiments, but not the particular experiments. Since the outcome is not predictable, the exact setting and order of experiments should be specified and adjusted as we get the first results.formulate positive, outcome of expermients are usualle not predictable These four groups are specified as follows:

1. Experiments with the initial Trafficlight system.
2. Experiments with the initial Trafficlight system producing noisy logs.
3. Experiments with the object oriented TrafficlightOO system.
4. Experiments with both systems from performance perspective.

As stated in Section 1.2, the research objective of this thesis is to develop a unifying framework realizing a model profiling approach for a combined but loosely-coupled usage

of MDE and PM techniques. In order to reach the research objective the specified research questions should be answered. The first and the second groups of experiments collect enough data for answering research questions 1 and 2. This data will serve as a basic prove of concept for the unifying framework and its implementation showing the actual possibility to generate a descriptive model from a specially equipped MDE-based system. The third group of experiments collect additional qualitative data from the second more complex TrafficlightOO system. Altogether, model profiles resulting from these three groups of experiments give a basis to verify the systems' behavior at runtime for answering research question 3. The third and the fourth groups of experiments collect data for answering research question 4, whether it is possible to maintain model profiles for multiple concerns, such as functionality, performance, and component interrelations, through unifying framework.

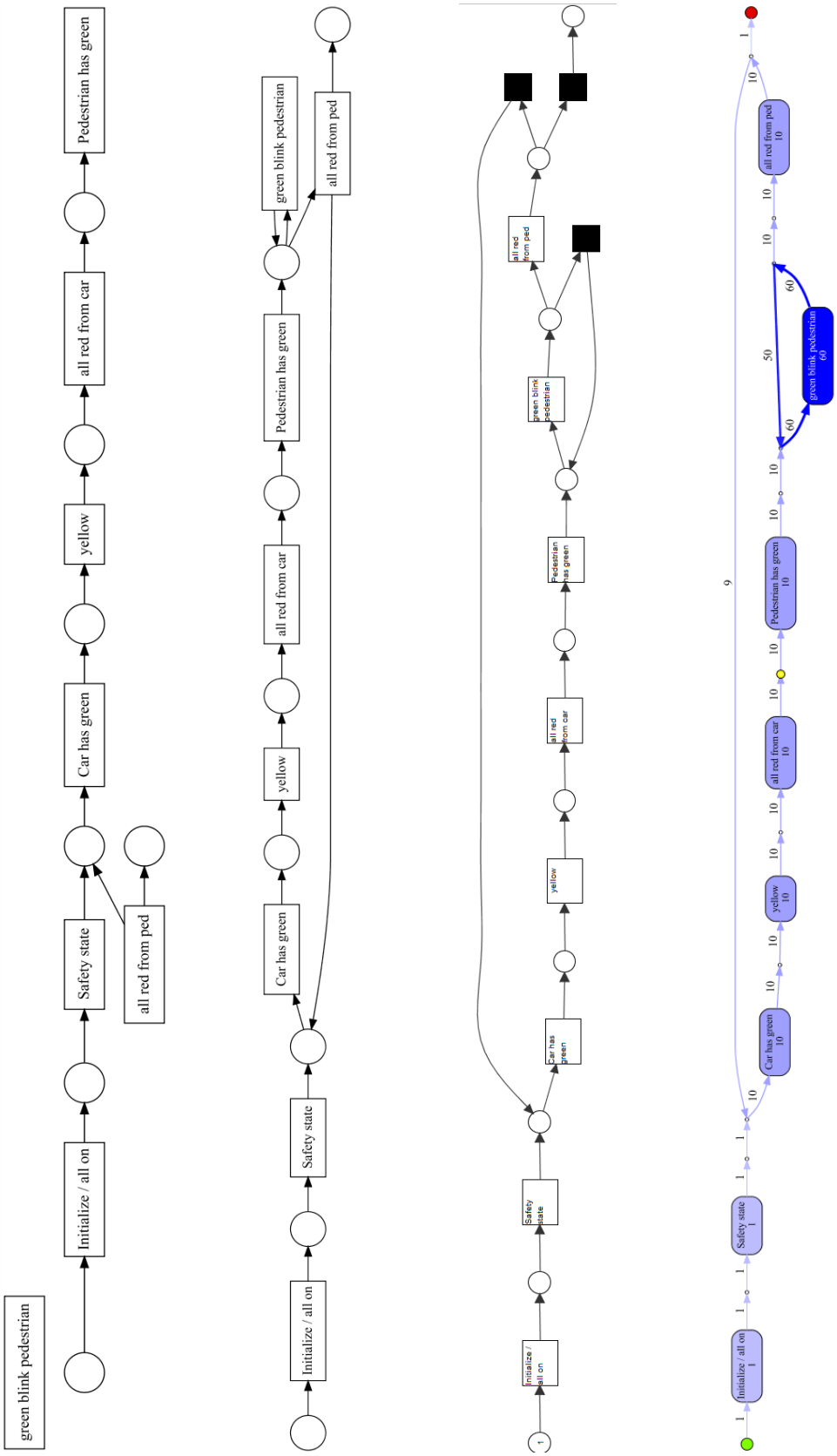
5.3 Results

This section presents the results of the experiments, i.e. the model profiles of the systems under study created within the unifying framework. Table 1 in Appendices presents the complete list of twenty two conducted experiments in the scope of this case study. In the column *System under study* it is defined, which system produced the event logs for the experiment. The next column *Observed aspect* contains the observed aspect of the operational semantics, e.g., state changes or attribute value changes during system operation. The column *Algorithm (Plug-in)* specifies the algorithm applied to the collected event logs. The last column *Special conditions* points out, whether the event logs contain noise or a certain threshold was set for the experiment. In the following text we refer to the experiments listed in this table.

Group 1. Experiments with the initial Trafficlight system (1 to 7).

For the first group of experiments we produced an observation model consisting of ten full runs of the initial Trafficlight system without any special conditions, i.e., without interruptions or noise (see observation model number 1 in Appendices, Table 2). These ten runs correspond to ten *process instances* according to the CSC observation language (see Section 5.3). PM uses the term *case* for the same concept. Since in this section we describe model profiles created by the PM-tool ProM, we use the term case as well.

In the first five experiments we observed state changes of the Trafficlight system. We applied the `OL2StateChanges.atl` transformation to the observation model in order to isolate log entries of type `CurrentStateChange`. *Experiment 1* was conducted using the *Alpha Miner* plug-in and the α-algorithm producing a Petri net. The resulting model profile is shown in Figure 5.7a. If we compare this model profile with the prescriptive model (see Section 5.2.3, Figure 5.3), we can assert, that the set of the states was discovered correctly, as well as their major order. Nevertheless, we also find several inconsistencies.



(a) E1. (b) E2. (c) E3. (d) E4.

Figure 5.7: Model Profiles for Experiments 1, 2, 3, and 4.

Firstly, the state *green blink pedestrian* is detached from the rest of the model profile. Secondly, the returning path from *all red from ped* to *car has green* is missing, i.e., the loop is not closed. In Section 2.2.3 we mentioned, that the α-algorithm is unable to discover short loops of one or two events. The detached state *green blink pedestrian* is the only one in the loop, so the algorithm could not handle it. This could also lead to the second issue of not closing the loop, since one state inside the loop was dislocated. Nonetheless, the algorithm discovered the end point of the cycle, which is *all red from ped*. Thus, being aware of this weak point of the α-algorithm, we can conclude, that the real reason of these inconsistencies lies in the algorithm itself and not in the behavior of the generated Trafficlight at runtime.

Therefore, the *Experiment 2* was conducted using the *Alpha Miner* plug-in, but with the improved version of the α-algorithm, i.e., the α^{++}-algorithm. In contrast to the first model profile, the second one shows the correct set of states, as well as their correct order (see Figure 5.7b). This is the first confirmation that the implementation of the unifying framework allows us to verify runtime behavior for the Trafficlight system by comparing a prescriptive (design) model and a corresponding model profile. In order to obtain more evidence we conducted more experiments with the algorithms described in Section 2.2.3, namely inductive miner and heuristic miner.

The model profile produced in *Experiment 3* by the *Inductive Miner* plug-in [JJLFA13] is shown in Figure 5.7c. This model profile shows the correct behavior corresponding to the design model in terms of state space and state ordering. The only difference to the second model profile is the black transitions in the Petri net. In PM such transitions are called *silent transitions* meaning they are artificial and not observed in the event log [vdA16]. The initial mining result of the inductive miner is a process tree which is further mapped into and shown as a Petri net. This mapping often produces silent transitions to redo the loops or end a Petri net, as we can see in Figure 5.7c.

We conducted *Experiment 4* with the *Visual inductive Miner* plug-in [LFvdA14b]. This plug-in uses Inductive miner as process discovery algorithm and adds new visualization and animation. The resulting model profile is show in Figure 5.7d. Since the discovery algorithm stays the same as in the previous experiment, the state space and state ordering are correctly discovered. Additionally to this, the model profile is enriched with frequency of path execution. The most frequent paths and elements have more intense color. For example, we see that *initialize* and *safety state* were performed once as the system was started. Then the big loop was entered and performed ten times, although the returning path was performed nine times. This means, the tenth successful cycle ended after *all red from ped*. Moreover, the *Visual inductive Miner* plug-in computes process replay for animation purposes. The yellow dot in Figure 5.7d symbolizes a current state of the system. In the animation the dot is actually moving according to the registered timestamps of the events.

During *Experiment 5* we used *Heuristic Miner* [**?**] producing a heuristic net, i.e., a casual net based on the dependency graph (see Section 2.2.3, Figure 2.12). The resulting model profile is shown in Figure 5.8. The heuristic net shows the correct state space and

space ordering according to the design model. Moreover, the formalism and visualization of such a heuristic net corresponds to the formalism of state diagrams, which makes the direct comparison more convenient. Additionally, *Heuristic Miner* computes and visualizes the frequency of path execution, as well as the frequency of entering a state. For example, in the lower part of 5.8 the properties of the state *green blink pedestrian*, which is highlighted in red color, are shown. This state was entered 60 times via two different inputs, i.e., from two previous states. 10 times (or 16.67%) the previous state was *pedestrian has green*, and 50 times (or 83.33%) the state *green blink pedestrian* was reentering itself, which corresponds to the design intention.

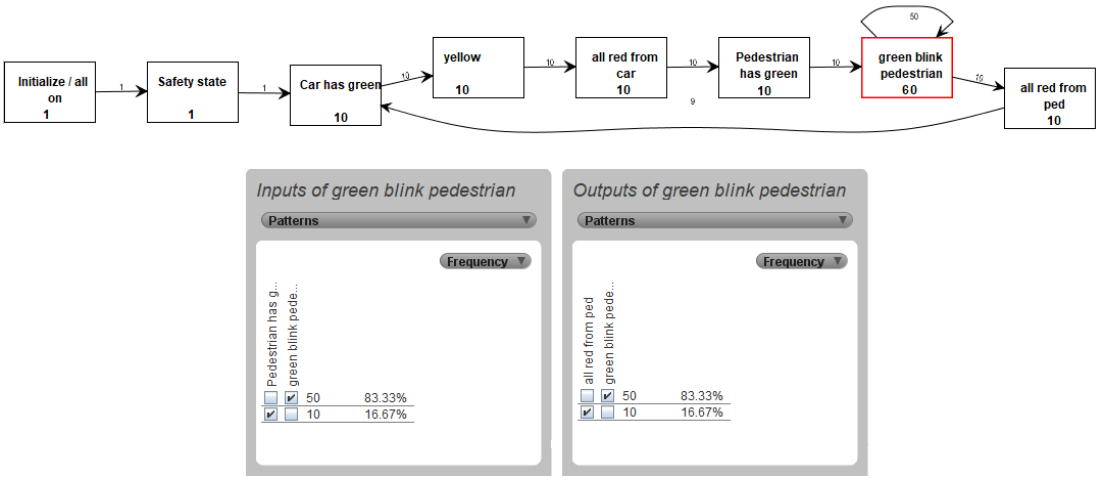


Figure 5.8: Model Profile for Experiment 5.

In *Experiments 6 and 7* we observed value changes of the attribute `blinkcounter` (see Section 5.2.3, Figure 5.2). Since the pin related attributes stay constant and the pin values attributes are bound to an enumeration with two possible values (LOW and HIGH), the most interesting attribute to observe is the `blinkcounter`. According to the state diagram, the values of `blinkcounter` should lie within the range [0..6] (see Section 5.2.3, Figure 5.3). In the similar way as we did for the first experiments, we applied the `OL2AttributeValueChanges.atl` transformation to the observation model in order to isolate log entries of type `AttributeValueChange`. The *Experiment 6* was conducted using *Alpha Miner* plug-in and the α^{++}-algorithm. Figure 5.9 shows the resulting Petri net for Experiment 6, i.e., a model profile of value changes of the attribute `blinkcounter`. The model profile shows the correct values of the attribute, i.e., the range [0..6], as well as their correct sequence starting with "1". After the initial value "0", the first registered value change is "1", which is also the first transition in the resulting Petri net. In *Experiment 7* we used *Heuristic Miner* to verify additionally, how many times the `blinkcounter` took different values. Since we analyzed a log with ten cases, every value has ten occurrences at runtime as designed (see Figure 5.10).

In the first group of experiments we observed the Trafficlight system at runtime and

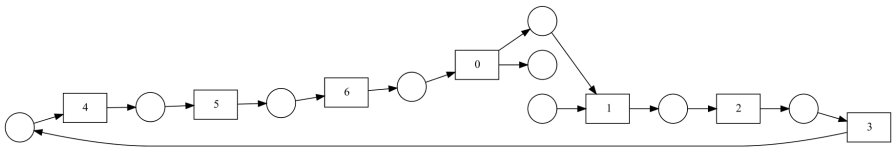


Figure 5.9: Model Profile for Experiment 6.

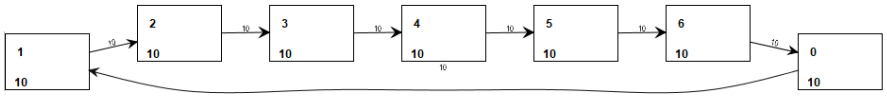


Figure 5.10: Model Profile for Experiment 7.

registered its behavior in form of observation models. Unlike unstructured log output, observation models and corresponding transformations allowed us to observe separately different aspects of the operational semantics of the CSC language (see Section 5.3), such as current state changes (see Figure 5.8) or attribute value changes (see Figure 5.10), and to create specific model profiles. This serves as a basic prove of concept for the unifying framework showing the actual possibility to generate a model profile from an especially equipped MDE-based system.

For the observation model containing clean runs of the Trafficlight system without noise all applied algorithms, except for the α-algorithm, showed comparable results and captured the runtime behaviour correctly, including correct state space and state ordering. *Visual inductive Miner* and *Heuristic Miner* both show additional information in model profiles, such as path and state frequency (see Figures 5.7d, 5.8, and 5.10).

Group 2. Experiments with the initial Trafficlight system producing noisy logs (8 to 12).

The second group of experiments is to be conducted with the previously applied algorithms using noisy event logs. Since in the first group of experiments *Alpha Miner* with α^{++}-algorithm, *Visual inductive Miner*, and *Heuristic Miner* showed comparable model profiles for a clean event log, in the second group of experiments we use these three algorithms and compare the model profiles they produce from noisy logs. Considering the fact, that both our systems under study produce sound logs without any noise by design, we artificially inserted a component that simulates system failure according to the bathtub reliability curve [LB17]. This means, every electronic component, in this case a light-emitting diode, is exposed to a failure risk. In the beginning of operation the failure rate is decreasing (early failures), then the constant failure rate phase follows (random failures). As the electronic component tires, the failure rate is increasing (wear-out failures). This change of the failure rate over time forms a so called bathtub reliability curve. Using this failure simulation we can produce noisy logs, that are not reflecting the system behaviour as prescribed by initial models, and observe how different algorithms deal with noisy logs.

For the comparison of model profiles resulting from different algorithms PM offers four

competing quality criteria [vdA16]:

1. Fitness
2. Precision
3. Generalization
4. Simplicity

A discovered model profile with good *fitness* allows the major behavior registered in the event log. A model profile with perfect fitness represents all possible paths of the log. The log can be fully replayed on such a model profile. However, a discovered model profile can allow "too much" behavior if it lacks *precision.* Thus, such a model profile is called not precise, or "underfitting", since it allows behavior completely unrelated to the one shown in the event log. *Generalization* is related to the concept of "overfitting". An overfitting model profile is too specific for the purpose, i.e., it does not generalize enough. The *Simplicity* criterion refers to Occam's Razor: *"don't multiply entities beyond necessity."* [Occ]. According to this principle, we should aim for the simplest possible model that can explain the behavior shown in the event log, but not simpler than that. The complexity of the model profile could be determined, e.g., by the number of its nodes and arcs. The more complex a model profile is, the harder it is to read.

It is quite a challenge to keep these four quality criteria in balance. Depending on the purpose of model profile creation, we might need a more generalized one, or a more precise one. Since quantified evaluation of algorithms is not the research objective of this thesis, we aim to explore, which algorithm produce the optimal model profile out of noisy logs in order to allow runtime verification. The results of this comparison are presented in the Table 5.2, where every criterion is evaluated as high or low. Nevertheless, the four quality criteria can be quantified as described in [vdA16] and used for the precise evaluation of systems with more sophisticated behavior and more complex, noisy event logs.

For the second group of experiments we produced an observation model consisting of twenty one runs, i.e., cases, of the initial Trafficlight system (see observation model number 2 in Appendices, Table 2). Five out of twenty one cases resulted in a system failure. Table 5.1 presents the states at which the Trafficlight system failed. Since *green blink pedestrian* is the most frequently used state due to the blinking cycle, it produced the most failures. In this group of experiments we observed state changes of the Trafficlight system, since further observation of attribute value changes will not bring new insights. We applied the `OL2StateChanges.atl` transformation to the observation model in order to isolate log entries of type `CurrentStateChange`.

In *Experiment 8* we produced a model profile using the *Alpha Miner* plug-in with the α^{++}-algorithm (see Figure 5.11). Since the model profile with noise is more complicated than the clean one, we introduced colored highlighting over the discovered model profile for better comprehension. The blue colored paths shows the normal behavior of the Trafficlight system without noise. This behavior corresponds to the design model in

Failed state	Number of failures
car has green	1
green blink pedestrian	3
all red from car	1

Table 5.1: List of system failures

terms of state space and state ordering. One additional blue arrow from the beginning of the Petri net to the *safety state* indicates, that the *initialize* state was not included in several cases and the cycle was started from the *safety state*. This pattern happens when the system enters the *system down* state and starts a new case beginning with the *safety state*. The *system down* state was artificially inserted into the observation model by the failure simulator component. The red arrows leading to this state show that the system failed from the states *car has green*, *green blink pedestrian*, and *all red from car*. Therefore, the model profile has high fitness, since it allows the normal behavior of the system (see Figure 5.3) and all three of its failures (see Table 5.1). On the other hand, the model profile also contains yellow paths, which are not allowed in the event log., e.g., the paths from *pedestrian has green* to *system down / yellow*. Thus, the model profile created by *Alpha Miner* lacks precision. Additionally, we can conclude, that the model profile has low generalization, since it does not consider the frequency of failures and shows all possible failure paths. Moreover, the simplicity of this model profile is also low, considering the duplicated paths from *pedestrian has green* to *system down / yellow* and the overall complexity of the Petri net. Here and in the following discussion we evaluate all four criteria in relation and comparison to other model profiles in this group of experiments.

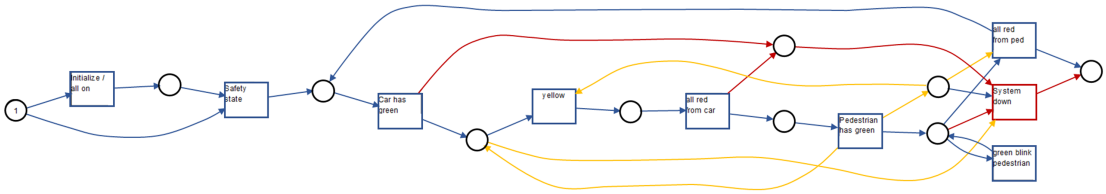


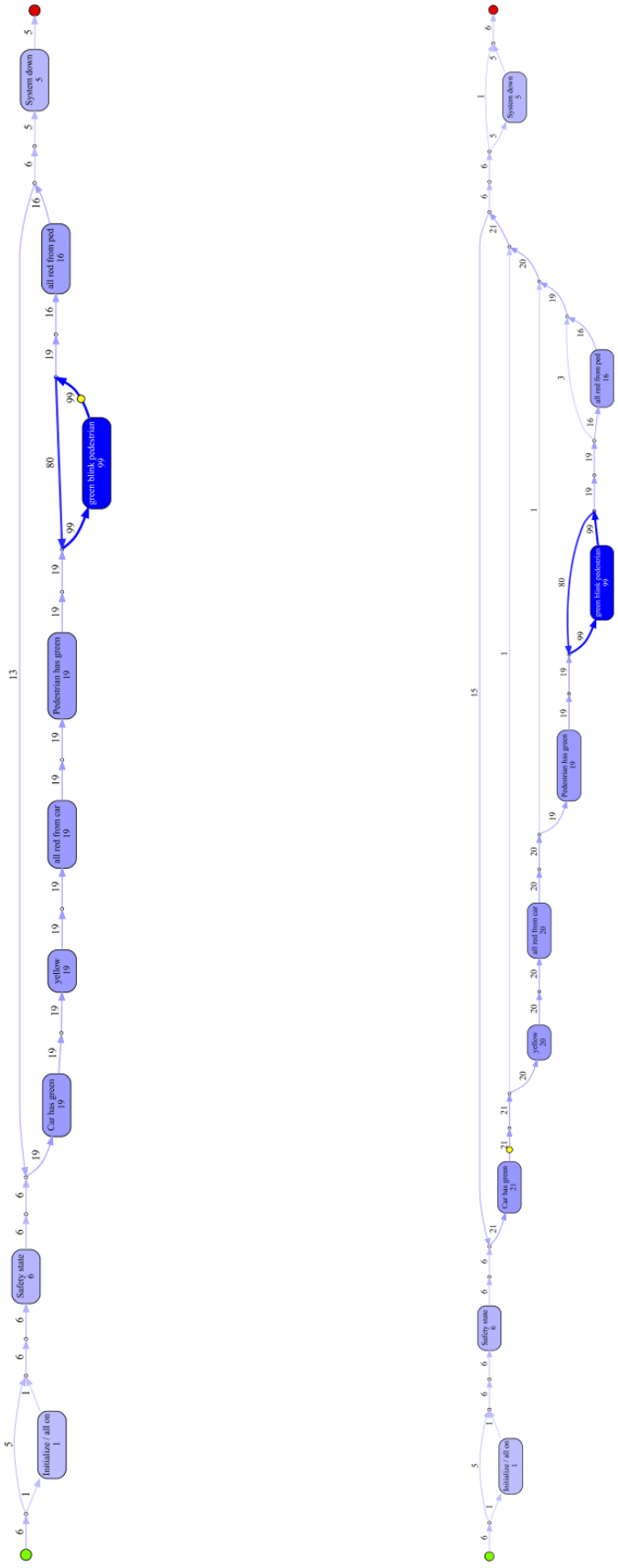
Figure 5.11: Model Profile for Experiment 8

In *Experiments 9 and 10* we discovered model profiles using the *Visual inductive Miner* plug-in (see Figure 5.12). In the *Experiment 9* we set the relative threshold to "0.8", which is also the default value in the *Visual inductive Miner* plug-in (see Section 2.2.3). This gave us a rather generalized model profile (see Figure 5.12a). As mentioned in the description of Experiment 4, *Visual inductive Miner* enriches the model profile with the information about path frequency, which is an advantage over *Alpha Miner*. The discovered profile captures normal behavior, indicates the *system down* state, and provides information about its frequency. Apart from that, the Petri net is simple and not

overloaded. However, we can not explicitly read, at which states the five failures occurred. We can assume, that three failures happened at the state *green blink pedestrian*, since the frequency of the path drops from 19 to 16 after this state, and the other two failures happened somewhere between *car has green* and *pedestrian has green*, since the frequency of this path is 19 instead of the expected 21. Therefore, we conclude, that the model profile at this threshold does not provide neither high fitness, nor high precision. On the other hand, the optimal application of this model profile depends on the goal, which an observer would like to achieve. If the precise location of the failures is irrelevant for the observer, this model profile gives the desirable information about the failure frequency.

In *Experiment 10* we aimed to find out exactly, at which states the failures had happened. Thus, we set the relative threshold to "1.0", which allows us to achieve the highest possible fitness. Here we made this decision in favor of fitness at the cost of low generalization and low simplicity. However, this model profile gives us more information about the source of the failures (see Figure 5.12b). Here we see, that the normal behavior of the Trafficlight system was interrupted after the states *car has green*, *green blink pedestrian*, and *all red from car*. This is shown both by additional paths and by the reducing frequency in the normal trace from *car has green* to *pedestrian has green*. Nevertheless, the model profile lacks precision, i.e., it allows behavior not observed in the event log. For example, although we know, that three failures happened at *green blink pedestrian* and there is a path showing the *all red from ped* was skipped, the model profile itself allows the misleading path from *all red from ped* to *system down*. This profile is only readable, if we keep the failures in mind and know what we are looking for, which is in practice not usually the case.

In *Experiments 11 and 12* we produced model profiles using the *Heuristic Miner* plug-in (see Figures 5.13 and 5.14). In *Experiment 11* we set the dependency threshold to "90" and the relative to best threshold to "5", which are the default values in the *Heuristic Miner* plug-in (see Section 2.2.3). With these settings the resulting model profile provides a generalized and simple view at the runtime behavior. As mentioned in the description of Experiment 5, *Heuristic Miner* computes and visualizes the frequency of path execution, as well as the frequency of entering a state, similar to the *Visual inductive Miner*. The normal behavior of the Trafficlight system and the presence of the *system down* state were correctly discovered. In comparison to the generalized model profile produced in Experiment 9 by the *Visual inductive Miner*, this model profile not only shows the *system down* state, but also locates its most frequent source, which failed three times, namely the *green blink pedestrian* state. Therefore, it demonstrates better fitness. Additionally, it presents, that the *system down* state was entered five times, leaving us with a question, what the other two failed states are. Similar to the approach we used in the description of Experiment 10, we can conclude based on reducing frequency, that the two missing failed states are *car has green* and *all red from car*. Since these two states are not exactly located, i.e. there are no paths from them to the *system down* state, the overall fitness of the model profile is still low. On the other hand, the model profile does not show any paths not presented in the event log, which can be characterized as high precision.



(a) E9. (b) E10.

Figure 5.12: Model Profiles for Experiments 9 and 10.

In order to locate the exact failed states, in the *Experiment 12* we set the dependency threshold to "0" and the relative to best threshold to "100", which are the maximum values in the *Heuristic Miner* plug-in for the highest possible fitness. As expected, with these settings the model profile provides a less generalized view at the runtime behavior and shows the paths from the *car has green* and *all red from car* states to the *system down* state (see Figure 5.14). This model profile shows deterministic behavior in comparison to the model profile of Experiment 10, i.e., it does not leave any space for misinterpretation (see Figure 5.12b). Here we can say exactly, which states failed and how often it happened. Since the model profile only shows the behavior actually seen in the log, its precision is high. Furthermore, the model profile in form of a heuristic net provides high simplicity as its number of nodes and arcs is lower in comparison, e.g., to the model profile in form of a Petri net produced in Experiment 10.

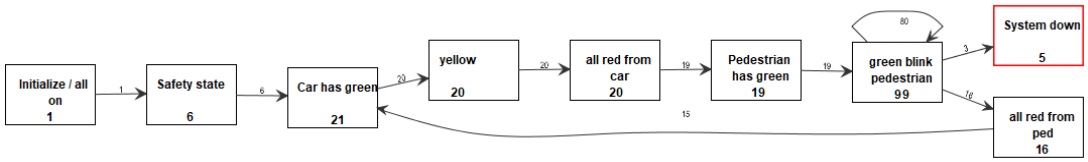


Figure 5.13: Model Profile for Experiment 11.

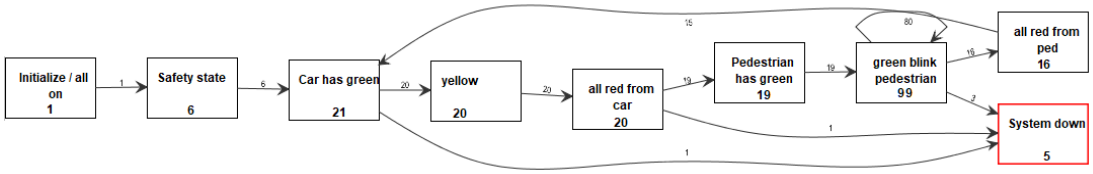


Figure 5.14: Model Profile for Experiment 12.

In the second group of experiments we observed the Trafficlight system at runtime and registered its behavior, including simulated failures, in form of observation models. We evaluated model profiles discovered by different PM algorithms from noisy operational data. The overview of the qualitative evaluation of the four criteria for the second group of experiments is presented in the Table 5.2.

Quality criteria	**E8: Alpha++**	**E9: Inductive 0.8**	**E10: Inductive 1.0**	**E11: Heuristic 90/5**	**E12: Heuristic 0/100**
Fitness	high	low	high	low	high
Precision	low	low	low	high	high
Generalization	low	high	low	high	low
Simplicity	low	high	low	high	high

Table 5.2: Comparison of model profiles

Based on this evaluation we came to conclusion that the model profiles produced by the *Heuristic Miner* plug-in give the most valuable insights about runtime behaviour of the MDE-based Trafficlight system (see Figures 5.13 and 5.14). These model profiles demonstrate high fitness and precision while keeping high simplicity. They provide visualization and formalism that allow their straightforward comparison with state diagrams used in the MDE-context. Thresholds used in the *Heuristic Miner* plug-in are advantageous to balance the trade-off between an "overfitting" model profile and an "underfitting" one. Thus, observers can decide based on their observation goal, whether a model profile should show an exact behavior or focus on more frequent patterns.

Group 3. Experiments with the object oriented TrafficlightOO system (13 to 19).

For the third group of experiments we produced two observation models of the running object oriented TrafficlightOO system. The first observation model consists of ten cases, without any special conditions, i.e., without interruptions or noise (see observation model number 3 in Appendices, Table 2). The second one consists of 29 cases, and 4 of them resulted in a system failure (see observation model number 3 in Appendices, Table 2). In the second group of experiments we evaluated the *Heuristic Miner* plug-in as an optimal choice for the MDE-based Trafficlight system. Therefore, for the third group of experiments with the object oriented TrafficlightOO system we use this plug-in with thresholds values set for the maximum fitness in order to detect all failures. In this group of experiments we observed state changes of the different components of the TrafficlightOO system. We applied the `OL2StateChanges.atl` transformation to the observation model in order to isolate log entries of type `CurrentStateChange`. Particularly, we applied helpers (see Section 4.6.4, Figure 4.8) in order to separate events sent from different components, i.e., `controlTrafficlight`, `carTrafficlight` and `pedestrianTrafficlight`, so that we can observe their distinct runtime behavior.

In *Experiment 13* we discovered a model profile for `controlTrafficlight`, which is identical to the one discovered in *Experiment 5* (see Figure 5.8). This proves that the control component of the TrafficlightOO system shows at runtime the same behavior as the initial Trafficlight system (see Figures 5.2 and 5.4). In *Experiment 14* we produced a model profile `controlTrafficlight` using noisy event logs with four failures: two at the state *green blink pedestrian*, one at *pedestrian has green*, and one at *all red from car*. The normal behavior of the TrafficlightOO system, as well as frequency and location of failures were correctly discovered (see Figure 5.15).

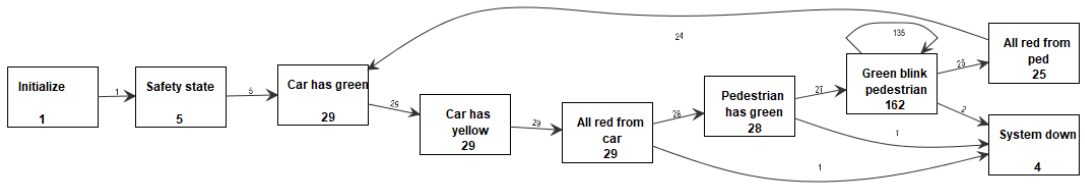
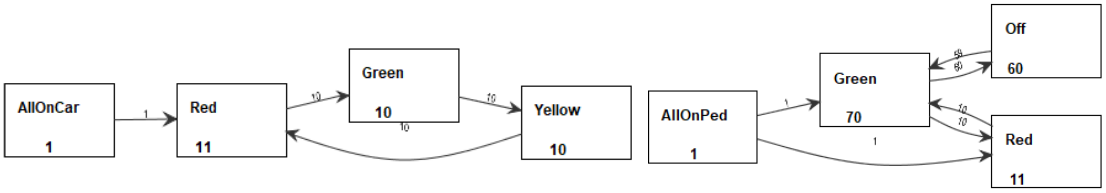


Figure 5.15: Model Profile for Experiment 14.

In *Experiments 15 and 16* we observed the components `carTrafficlight` and `pedestrianTrafficlight` as they perform without failures (see Figures 5.16a and 5.16b). In these experiments we analyzed operational data from distinct components that have underlying state machines different from the initial Trafficlight system. In both experiments the discovered model profiles correspond to the designed state diagrams in terms of state space and state ordering (see Figures 5.6b and 5.6a). Therefore, these experiments provide additional confirmation that the unifying framework enables us to verify runtime behavior based on operational data captured with reference to design models in the MDE context.



(a) E15.

(b) E16.

Figure 5.16: Model Profiles for Experiments 15 and 16.

In *Experiment 17 and 18* we observed the same two components running with the failure simulator. Both discovered model profiles indicate the *system down* state (see Figures 5.17 and 5.18). Moreover, based on these model profiles we can locate the failures more precisely. With this separation of control state machine and dependent component state machines we can dig deeper into dependent component model profiles and see at which states of these components the system failed exactly. For example, in the model profile for `carTrafficlight` in Figure 5.17 we see that this component failed at the *red* state. If we take a look at the model profile of the initial Trafficlight system (see Figure 5.14), we will only see at which overall state the system failed, but not which component it was.

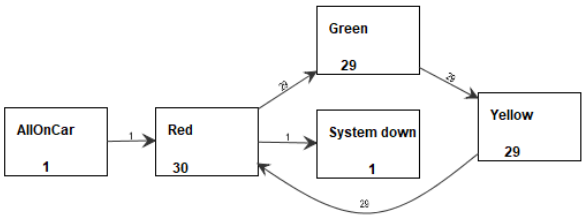


Figure 5.17: Model Profile for Experiment 17.

In the previous eighteen experiments we explored the so-called control-flow perspective of the PM. This perspective allows to discover state space and state ordering of the underlying processes happening during system operation. The result of mining this perspective is a generalized (or precised) model profile showing system behavior at

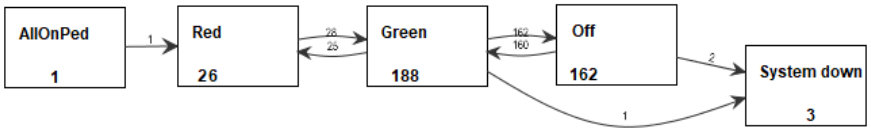


Figure 5.18: Model Profile for Experiment 18.

runtime. In *Experiment 19* we explored the organisational perspective of the PM. This perspective focuses on resources, i.e., actors, which actually perform the process. In the business context this could be, e.g., people, roles, and departments, whereas in the software engineering context the interaction happens, e.g., between systems, components, and classes. The result of mining this perspective is an organizational structure or a social network that shows actors and their interrelations.

In particular, we used the *Subcontracting social network Miner* plug-in to create a model profile showing the subcontracting relations between the three components of the TrafficlightOO system (see Figure 5.19). The main idea of subcontracting is to count how often one actor executed an activity in-between two activities executed by another actor. This may serve as an indication that the second actor subcontracts work to the first actor [vdARS05]. For this experiment we filtered all state changes from the observation model 3 (see Appendices, Table 2), so that the event log contains the state changes from all senders, i.e., components. In TrafficlightOO the components `pedestrianTrafficlight` and `carTrafficlight` are clear subcontractors of `controlTrafficlight` by design, since `controlTrafficlight` explicitly triggered their state machines. The model profile in Figure 5.19 confirms that these relations are held at runtime. The component `controlTrafficlight` has mutual relations with both its subcontractors, but the subcontractors themselves have no such relations to each other.

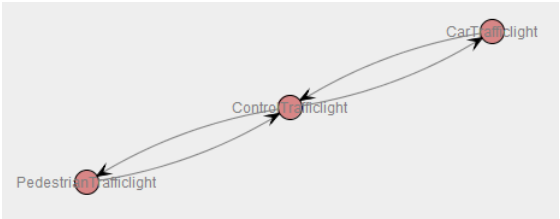


Figure 5.19: Model Profile for Experiment 19.

In the third group of experiments we observed the object oriented TrafficlightOO system at runtime and registered its behavior including simulated failures in form of observation models. We were able to verify runtime behavior of several dependent components and locate particular failures in these components. Additionally, we explored component interrelations from the organisational perspective. The experiments in the third group showed us, that with the unifying framework we can scale the creation of observation

models to several components and discover their behaviour with PM separately, as we did for the state machines, or as a whole system, as we did for the subcontracting social network.

Group 4. Experiments with both systems from performance perspective (20 to 21).

In the fourth group of experiments we explored the performance perspective of the PM (see Section 2.2.4). We used the observation model 1 without noise for the Trafficlight system and the observation model 2 without noise for the TrafficlightOO system (see Appendices, table 2). In the CSC design language the timing characteristics are explicitly assigned to `Transition`, namely the `waitingTime` property (see Section , Figure 5.1). In the behavioral design model, i.e., the state diagram, this property is set on a transition as, e.g., `wait2Sec` (see Figure 5.3). Since we would like to make timing observations, we applied the `OL2TransitionFirings.atl` transformation to the observation model in order to isolate log entries of type `TransitionFiring`.

For performance analysis ProM offers a plug-in called *Replay for Performance Analysis*. This plug-in replays a log on a Petri net that was discovered from this log. Therefore, in order to create this second input we used the *Inductive Miner* plug-in to produce Petri nets for both systems under study. In *Experiment 20* we used the event log and the discovered Petri net for the original Trafficlight system. The resulting model profile with a close-up is shown in Figure 5.20. In this figure every box symbolizes a transition. As expected, all transitions and their ordering were recognised correctly by *Inductive Miner*. Through replay the *Performance Analysis* plug-in enriches the model profile with timing information. This information is available through a click on a transition and includes minimum time, maximum time, average time, frequency, and a standard deviation of all time observations. We summarized the timing values for all transitions in Table 5.3. Coloring on the Petri net indicates the relative length of the waiting times, yellow transitions are the shorter ones and red transitions are the longer ones. As we can conclude from the results in Table 5.3, average transition times are exceeded at runtime in comparison with designed values (20 to 140 milliseconds average delay per repeating transition). Although for the Trafficlight system delays in the range of milliseconds are not crucial, for time critical systems this information is urgently important. Knowing the average delays engineers can adapt the waiting times directly in the design models to compensate for the timing discrepancies.

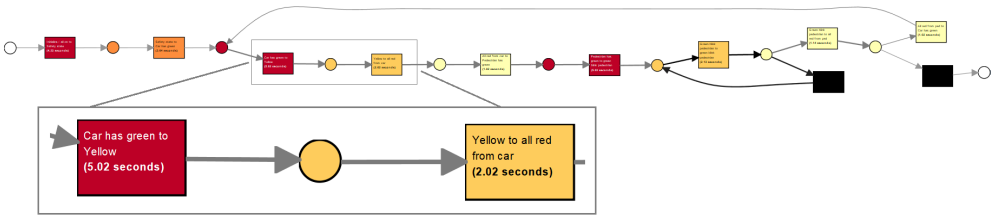


Figure 5.20: Model Profile for Experiment 20.

Transition	**Design time**	**Min. time**	**Max. time**	**Avg. time**	**Std. Dev.**	**Freq**
initialize **to** *safety state*	3.00 sec	3.22 sec	3.22 sec	3.22 sec	-	1
safety state **to** *car has green*	3.00 sec	3.04 sec	3.04 sec	3.04 sec	-	1
car has green **to** *yellow*	5.00 sec	5.01 sec	5.03 sec	5.02 sec	5.15 ms	10
yellow **to** *all red from car*	2.00 sec	2.01 sec	2.03 sec	2.02 sec	3.90 ms	10
all red from car **to** *pedestrian has green*	1.00 sec	1.01 sec	1.03 sec	1.02 sec	4.33 ms	10
pedestrian has green **to** *green blink pedestrian*	5.00 sec	5.01 sec	5.03 sec	5.02 sec	3.63 ms	10
green blink pedestrian **to** *green blink pedestrian*	1.00 sec	1.12 sec	1.15 sec	1.13 sec	7.79 ms	50
green blink pedestrian **to** *all red from ped*	1.00 sec	1.13 sec	1.17 sec	1.14 sec	13.19 ms	10
all red from ped **to** *car has green*	1.00 sec	1.01 sec	1.02 sec	1.02 sec	6.46 ms	9

Table 5.3: Performance evaluation for the Trafficlight system

Transition	**Design time**	**Min. time**	**Max. time**	**Avg. time**	**Std. Dev.**	**Freq**
initialize **to** *safety state*	3.00 sec	3.42 sec	3.42 sec	3.42 sec	-	1
safety state **to** *car has green*	3.00 sec	3.02 sec	3.02 sec	3.02 sec	-	1
car has green **to** *yellow*	5.00 sec	5.03 sec	5.04 sec	5.03 sec	3.48 ms	10
yellow **to** *all red from car*	2.00 sec	2.01 sec	2.05 sec	2.02 sec	11.23 ms	10
all red from car **to** *pedestrian has green*	1.00 sec	1.01 sec	1.03 sec	1.02 sec	6.70 ms	10
pedestrian has green **to** *green blink pedestrian*	5.00 sec	5.01 sec	5.02 sec	5.02 sec	3.50 ms	10
green blink pedestrian **to** *green blink pedestrian*	1.00 sec	1.13 sec	1.15 sec	1.14 sec	8.06 ms	50
green blink pedestrian **to** *all red from ped*	1.00 sec	1.13 sec	1.15 sec	1.13 sec	5.64 ms	10
all red from ped **to** *car has green*	1.00 sec	1.01 sec	1.02 sec	1.02 sec	3.08 ms	9

Table 5.4: Performance evaluation for the TrafficlightOO system

In the *Experiment 21* we performed the same operations as in the previous experiment and found the timing discrepancies for the `ControlTrafficlight` component of the

object oriented TrafficlightOO system. The resulting model profile looks identical to the one produced for the original Trafficlight system (see Figure 5.20). We summarized the timing values for all transitions of TrafficlightOO system in Table 5.3. In the model profile for TrafficlightOO calculated average transition times slightly differ from those in the previous experiment, although the difference lies in the range of milliseconds (20 to 140 milliseconds average delay per repeating transition) and can be neglected for such systems.

The primary purpose of the experiments in this group was not to compare the two systems under study, but rather prove that we can use the unifying framework for creating observation models that can be reusable for multiple concerns, such as functionality verification and performance evaluation. Based on the results of the experiments we conclude, that observation models used for control-flow discovery in the groups of experiments 1 and 3 can be also used in a performance perspective in order to analyze actual execution times and potential discrepancies between prescriptive models and performance at runtime.

5.4 Interpretation of Results

Answering research question 1.

In this thesis we realized the EbMP approach by definition of the unifying framework and its implementation within an experimental frame. The framework creates a link between design and execution phases of the software life cycle and smooths the clear distinction between them. On the design side, i.e., the prescriptive perspective of the framework, we used the MDE approach with its different levels. On the metamodeling level we introduced the so-called observation language defining the syntax and semantics of the execution-based data, i.e., observation models, that are to be created on the modeling level. We used these observation models on the execution side, i.e., the descriptive perspective of the framework, in order to produce model profiles of the initial design models via conclusively proven PM algorithms. Although we created a link between MDE and PM in our framework, they both stay combined in a loosely-coupled way, meaning the way they are originally established is not changed. We provide interfaces from MDE to PM to enable the transfer of the execution-based data in order to create model profiles. These interfaces are implemented as ATL transformations from the language-specific observation metamodels to the general XES format of existing PM tools.

To implement the framework we employed commercial tools, such as EA and its extension Vanilla Source for modelling and generating the systems under study. We also employed open source tools, such as EMF for implementing transformations to XES format and ProM for creating model profiles out of observation models. Additionally, we implemented LogClient and Microservice Observer to transfer and persist the execution-based data.

Answering research question 2.

In the implemented unifying framework for the EbMP the operational semantics can be transferred from the design language to the observational viewpoint via observation language. The observation language captures the operational semantics of the design language in the prescriptive perspective and transfers it through an especially equipped code generator and logging into the descriptive perspective. An important concept enabling this transfer is the ≪observe≫ stereotype which is used to mark specific design language elements related to operational semantics in order to register them at runtime as logging statements and include them into observation models. Therefore, observation models, created as a result of this flow, contain semantically structured information about the system behavior at runtime.

In particular, we report on our results concerning two MDE-based traffic light systems which are enhanced with execution-based model profiling capabilities. The first results were shown as a basic prove of concept in the first group of experiments, where we automatically persisted operational data as observation models. In the following, we were able to use these observation models to automatically produce model profiles by applying PM algorithms to them. These model profiles are semantically aligned to the initial design models. Furthermore, the results were confirmed in the second and the third groups of experiments. Thus, we conclude that the operational data can be automatically stored as descriptive models derived from the operational semantics of the design language.

There are three points of human intervention at the current stage of the implementation of the unifying framework. Firstly, the creation of the design models is performed manually. After that the executable code is generated automatically, execution-based data is automatically captured and stored. The second point of manual assistance is starting the PM tool to create model profiles. The third human-powered activity is the comparison of design models with discovered model profiles. Both latter activities can be automated in the future development of the unifying framework. So the only manual work left would be the design model creation.

Answering research question 3.

Since operational system changes are registered with reference to design models, the discovered model profiles inherit a semantic relation to these models. In particular, we are able to recognise state space and state ordering in these model profiles and to align them with design models for runtime verification and detecting inconsistencies between designed behavior and runtime behavior.

In the first group of experiments we observed the Trafficlight system performing as designed, without interruptions or noise. We applied different PM algorithms to discover model profiles and received satisfying results. Particularly, *Alpha Miner* with the α^{++}-algorithm, *Inductive Miner*, and *Heuristic Miner* demonstrated especially satisfactory results. Model profiles discovered by these algorithms show correct control-flow in terms of state space and state ordering. Moreover, we applied transformations to isolate different aspects of the operational semantics of the CSC language, such as state changes and

attribute value changes. Therefore, we were able to observe them separately and verify these aspects against the design models.

In the second group of experiments we introduced a failure simulator for the Trafficlight system in order to evaluate how different algorithms deal with irregular behavior and noisy event logs. *Heuristic Miner* provided the most valuable insights into the runtime behaviour of the Trafficlight system. The model profiles discovered by *Heuristic Miner* demonstrate high fitness and precision while keeping high simplicity. They provide visualization and formalism for their straightforward comparison with state diagrams used in the MDE-context, and, therefore, system's runtime verification. Thresholds used in the *Heuristic Miner* are advantageous to balance the trade-off between an "overfitting" model profile and an "underfitting" one. Thus, observers can decide based on their observation goal, whether a model profile should show an exact behavior or focus on more frequent patterns.

Finally, in the third group of experiments we observed runtime behavior of the object oriented TrafficlightOO system. Model profiles discovered for separate components of the system show their runtime behaviour as designed. Additionally, simulated failures were discovered not only in model profile showing the general behavior of the system, but also precisely in a particular model profile of a component. The experiments in the third group showed that with the unifying framework we can scale the creation of observation models to several components and discover and verify their behaviour separately.

Overall, the model profiles produced by the implemented unifying framework provided valuable insights into the runtime behavior of the systems under study and their components, including irregular noisy behavior. Therefore, we conclude that resulting model profiles for this case study are sufficient enough to verify a system's behavior at runtime.

Answering research question 4.

In the conducted experiments we explored three different perspective of PM, namely control-flow, organisational and performance perspective. These perspectives cover multiple concerns that arise when we verify a system's runtime behavior. The control-flow perspective reflects the backbone of the runtime behavior, i.e., the functional properties of a system. In the first and the second groups of experiments we were able to successfully verify the system's state changes and attribute values changes at runtime against the design models. In the third group of experiments we explored component interrelations of the object oriented system through the organisational perspective and were able to confirm the subcontracting pattern. This could be especially beneficial for a system with many components, where their mutual communication patterns are critical.

In the fourth group of experiments we analyzed time inconsistencies between the real life performance and the design models through the performance perspective. These inconsistencies lie within the range of milliseconds (20 to 140 milliseconds average delay per repeating transition) and are not significant for systems such as the Trafficlight system. Nevertheless, for time critical and safety critical systems this information is crucial. Knowing the average delays, engineers can adapt the waiting times directly in

the design models to compensate for the timing discrepancies and 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.

It is important to mention that for all three perspectives we used the same observation models produced within the implemented unifying framework. We didn't need any major prepossessing steps, the transformation and filtering of the models were conducted automatically according to the settings in the Microservice. The observation models contain all necessary operational data for creating model profiles in three different perspectives. Therefore, we conclude that the unifying framework allows us to maintain model profiles for multiple concerns, such as functionality, performance, and component interrelations.

5.5 Threats to Validity and Limitations

To critically reflect on our results, we discuss several threats to validity of our study. In the current implementation of the unifying framework we do not consider the logging instrumentation overhead which may increase the execution time of the instrumented application. This may be a crucial point for time critical systems and has to be validated further in the future. Furthermore, in the current implementation we assume to have an execution platform with network access to send the registered logs to the Microservice. This requirement may be critical in restricted environments, and measurements of network traffic have to be done. Another assumption is that the code generator undergoes large-scale testing and delivers reliable results, so that the potential runtime errors do not stem from it. Nevertheless, such errors can not be completely excluded in practice. In fact, the unifying framework can also be beneficial for code generator validation.

Regarding the systems under study, we would like to mention, that both of them do not deal with concurrency, since they are running single threaded. Extensions for supporting concurrency may result in transforming the strict sequences in partially ordered ones. The next limitation of our case study is the manual comparison of design models and model profiles based on human judgment. In this case the size of the systems under study is comprehensible enough to allow for such a comparison. For bigger and more complex systems we recommend to implement automatic comparison, e.g., via model transformation. Results of this automatic comparison can further be used in the back propagation of the new information to the design models and their adaptation. Nevertheless, it is important to mention, that such automatic comparison as well as automatic adaptation demand versatile and extensive testing with big data sets. Finally, we have to emphasize that we currently only investigated one modeling language and two similar systems modeled in this language. Therefore, more experiments are needed to verify if the results can be reproduced for a variety of modeling languages and more complex systems.

CHAPTER 6

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 [AGN15], Process Mining (PM) [vdA16], complex event processing [Luc01], specification mining [DKM+10], Finite-state Automata (FSA) learning [GMC+92], as well as knowledge discovery and data mining [FPSS96]. All these techniques aim at better understanding of 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 M@RT, where runtime information is propagated back to software engineering. In comparison to M@RT, which also refers to runtime adaptation mechanisms and provides infrastructure to instantiate models, our approach additionally focusing on the history of changes, since it captures model profiles over different periods of time saving historical data. Observing model profiles helps to grasp the changes happening in a running system and takes the evolutionary aspect of engineering into concern. Furthermore, there are several approaches for runtime monitoring. Blair et al. [BBF09] 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. [HMF+15] 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. [DGJ+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 [SZ16] discuss the question if information provided by models can also improve the analysis capabilities of human users. Earlier these authors presented a collection of runtime event types that supports the assessment of current system states [SZ15]. Heldal et al. [HPE+16] 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 the industry. Kuhne [KÏ16] highlights the differences between explanatory and constructive modeling, which give rise to two almost disjoint modeling universes, each one based on different, mutually incompatible assumptions, concepts, techniques, and tools.

Among the most recent research on the matter of runtime monitoring and verification we can mention Zdun et al. [SZ17], who present a language to specify recurring monitoring patterns. These patterns can be automatically expanded into monitoring code for given models of the analyzed system. In their further work the authors [SZ18] designed a taxonomy that captures essential architectural decisions when implementing a system and supports the analysis of its runtime behaviour using models. Bencomo and Garcia Paucar in [BG19a] tackle a challenge of updating runtime models during system execution. They present an approach that allows to update runtime models collected by the monitoring infrastructure using Bayesian inference. Furthermore, they demonstrate how to propagate the changes from a runtime model back to the monitored system, i.e., to produce the corresponding self-adaptations. To continue the topic of updating runtime models, Brand and Giese [BG19b] introduce the fundamental idea of a generic modeling language for adaptable architectural runtime model instances.

CHAPTER 7

Summary and Future Work

In this thesis, we pointed out 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 create a link between these two approaches. Therefore, we presented EbMP, whose core idea is to equip MDE-based systems in such a way, so that their operational data can be stored in a structured form and transformed into abstracted model representations.

In order to realize the EbMP approach we defined a generic unifying framework for EbMP that allows to align downstream information from the MDE-based systems with upstream information gathered at runtime. On the design side, i.e., the prescriptive perspective of the framework, we used the MDE approach with its different modeling levels. On the execution side, i.e., the descriptive perspective of the framework, we introduced observation models in order to produce model profiles via conclusively proved PM algorithms. To keep both approaches combined in a loosely-coupled way we provide interfaces, i.e., ATL transformations, from the language-specific observation metamodels to the general XES format of existing PM tools. The EbMP approach allows to avoid time-consuming preprocessing of massive data sets in order to extract valuable runtime information. An observation model can be investigated and explored using different tools, such as simulation tools, monitoring and visualization or animation tools, etc. It may be subject to various transformations, it may have textual and visual representations. With model profiles an engineer can gain insights into runtime behavior by monitoring design models during the system's execution, and detect actual or predicted violations of functional or non-functional properties.

We implemented the unifying framework within an experimental frame and conducted a case study with two MDE-based systems for traffic lights. In this case study we proved that operational data can be automatically stored as observation models derived from the operational semantics of the design language. Moreover, the unifying framework allows to store and analyze operational data separately for different aspects of operational

semantics, such as state changes or attribute value changes. Furthermore, we were able to verify a system's runtime behavior based on model profiles discovered by PM algorithms from observation models. *Heuristic Miner* provided the most valuable insights into the runtime behaviour of the systems under study and their components, including irregular noisy behavior. The model profiles discovered by *Heuristic Miner* demonstrate high fitness and precision while keeping high simplicity. Finally, we proved that the unifying framework allows us to maintain model profiles for multiple concerns, such as functionality, performance, and components interrelations. The observation models contain all necessary operational data for creating model profiles in different perspectives of PM.

While the first results seem promising, there are still several open challenges. The most important next step would be to automate the comparison of design models and discovered model profiles, e.g., via definition of their metamodels and further transformation. Results of this automatic comparison can further be used in the back propagation of the new information to the design models and their adaptation. Nevertheless, it is important to mention, that such automatic comparison as well as automatic adaptation demand versatile and extensive testing with big data sets. Thereafter, the whole workflow starting from design models till the PM visualization can be automated. The open source PM tool ProM that we used allows complete integration and can be automatically started for continuous analysis with visualization. So the only manual work left would be the design model creation.

Currently we only investigated one modeling language and two similar systems modeled in this language. Therefore, more experiments are needed to verify if the results can be reproduced for a variety of modeling languages and more complex distributed systems. The same applies to the exploration of different PM perspectives. In the organisational perspective it would be beneficial to observe more complex systems with multiple components and more sophisticated interrelations. In the performance perspective it would be important to observe a system for a sufficiently long period of time to have enough runtime information for reliable statistical values.

List of Figures

2.1 Overview of the MDE methodology (top-down process)[BCW12]. 8
2.2 A typical MDE-based software development process [BCW12]. 9
2.3 Relationships between metamodel and model [BCW12]. 11
2.4 Definition of a transformation between models [BCW12]. 12
2.5 Process mining as the bridge between data science and process science [vdA16]. 13
2.6 The three basic types of process mining in terms of input and output [VDA12]. 14
2.7 Process mining. An example. Based on [vdA16] 16
2.8 Typical process patterns and the footprints they leave in the event log [vdA16]. 17
2.9 Two basic steps of the α-algorithm 17
2.10 Petri Nets with short loops of length one and two. 19
2.11 An example of a casual net [vdA16]. 20
2.12 An example of a dependency graph [vdA16]. 21
2.13 An example of a process tree [vdA16]. 22
2.14 Replay of event logs on a discovered process model [vdA16]. 23
2.15 Example model enhanced with the performance perspective [RMSvdA09] 24
2.16 A Terminological Framework for M@RT [BFT$^+$14]. 24
2.17 A Conceptual Reference Model for M@RT with a close-up of M0 level. Based on [BFT$^+$14]. 25

3.1 Prescriptive models vs. descriptive models. 30
3.2 Framework for execution-based model profiling. 32

4.1 Component diagram of the technical realization of the EBMP-framework. 36
4.2 Raspberry Pi Components [Rasb] 39
4.3 Microservice Observer. 39
4.4 Mapping JSON to POJO and POJO to XMI. Example from the case study. 40
4.5 Transformation "Observation Language To XES". 41
4.6 Source metamodel Observation Language. Ecore class diagram. 42
4.7 Target metamodel XES. Ecore class diagram. 43
4.8 Transformations with helpers. 46
4.9 ProM 6 user interface [vdA16]. 47
4.10 Discovered Petri net [vdA16]. 48
4.11 Visual inductive miner [vdA16]. 48

5.1 Design language and observation language. 51
5.2 TrafficLight Class Diagram. 52
5.3 TrafficLight. State Diagram. 53
5.4 Object oriented TrafficLight. Class Diagram. 54
5.5 Object oriented TrafficLight. State Diagram. Control. 55
5.6 Object oriented TrafficLight. State Diagrams. 56
5.7 Model Profiles for Experiments 1, 2, 3, and 4. 58
5.8 Model Profile for Experiment 5. 60
5.9 Model Profile for Experiment 6. 61
5.10 Model Profile for Experiment 7. 61
5.11 Model Profile for Experiment 8 63
5.12 Model Profiles for Experiments 9 and 10. 65
5.13 Model Profile for Experiment 11. 66
5.14 Model Profile for Experiment 12. 66
5.15 Model Profile for Experiment 14. 67
5.16 Model Profiles for Experiments 15 and 16. 68
5.17 Model Profile for Experiment 17. 68
5.18 Model Profile for Experiment 18. 69
5.19 Model Profile for Experiment 19. 69
5.20 Model Profile for Experiment 20. 70

Acronyms

ATL ATLAS Transformation Language. 38, 39, 41–43

BPEL Business Process Execution Language. 15

BPM Business Process Management. 3

BPMN Business Process Model and Notation. 15, 35, 44, 45

CPS Cyber-Physical System. 1, 25, 26

CSC Class/State Charts. 48, 50, 55, 59, 68, 71

DSL Domain-Specific language. 10, 15, 29

EA Enterprise Architect. 4, 34, 35, 50, 51, 70

EbMP Execution-based Model Profiling. 27–29, 33, 47, 70, 71, 77

EMF Eclipse Modeling Framework. 33, 38–40, 70

EPC Event-driven Process Chain. 15

FSA Finite-state Automata. 75

GPIO general-purpose input/output. 36, 50, 51, 53

GPL General-Purpose language. 10, 15

IM Inductive Miner. 21, 22

JSON JavaScript Object Notation. 33–38

M2M model-to-model. 8, 11, 31, 39

M2T model-to-text. 8, 11, 12, 30

M@RT models@run.time. 5, 23–26, 31, 75

MARTE Modeling and Analysis of Real Time and Embedded systems. 10

MDE Model-Driven Engineering. 1–5, 7–11, 24–29, 35, 47, 55, 59, 65, 66, 70–72, 75, 77

OCL Object Constraint Language. 43

PAIS Process-Aware Information System. 12, 13, 28, 47, 48, 50

PM Process Mining. 3–5, 12–16, 19, 22, 28, 31, 34, 38, 43–45, 55, 57, 59, 64, 66–68, 70–72, 75, 77, 78

POJO Plain Old Java Object. 38

REST Representational state transfer. 37

UML Unified Modeling Language. 10, 14, 29, 30, 33, 35, 38, 44, 48

XES eXtensible Event Stream. 31, 34, 38–42, 44–46, 70, 77

XMI XML Metadata Interchange. 34, 38

XML Extensible Markup Language. 40

YAWL Yet Another Workflow Language. 15

Bibliography

[AGJ+14] Uwe Aßmann, Sebastian Götz, Jean-Marc Jézéquel, Brice Morin, and Mario Trapp. *A Reference Architecture and Roadmap for Models@run.time Systems*, pages 1–18. Springer International Publishing, Cham, 2014.

[AGN15] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. Profiling relational data: A survey. *The VLDB Journal*, 24(4):557–581, August 2015.

[Bau17] Bauernhansl. *Industrie 4.0 - Whitepaper FuE-Themen*, 2015 (accessed June 3, 2017)".

[BBF09] Gordon Blair, Nelly Bencomo, and Robert B. France. Models@ run.time. *Computer*, 42(10):22–27, October 2009.

[BCW12] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool, 2012.

[BFT+14] Amel Bennaceur, Robert France, Giordano Tamburrelli, Thomas Vogel, Pieter J. Mosterman, Walter Cazzola, Fabio M. Costa, Alfonso Pierantonio, Matthias Tichy, Mehmet Akşit, Pär Emmanuelson, Huang Gang, Nikolaos Georgantas, and David Redlich. *Mechanisms for Leveraging Models at Runtime in Self-adaptive Software*, pages 19–46. Springer International Publishing, Cham, 2014.

[BG19a] N. Bencomo and L. H. Garcia Paucar. Ram: Causally-connected and requirements-aware runtime models using bayesian learning. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 216–226, Sep. 2019.

[BG19b] T. Brand and H. Giese. Modeling approach and evaluation criteria for adaptable architectural runtime model instances. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 227–232, Sep. 2019.

[BGS+14] Amine Benelallam, Abel Gómez, Gerson Sunyé, Massimo Tisi, and David Launay. Neo4emf, a scalable persistence layer for emf models. 8569, 04 2014.

[CH06] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–645, July 2006.

[CPS] Cyber-physical systems (CPS), howpublished = `https://www.nsf.gov/publications/pub_summ.jsp?ods_key=nsf13502&org=nsf`, note = Accessed: 2019-02-17.

[DE95] Jörg Desel and Javier Esparza. *Free Choice Petri Nets*. Cambridge University Press, New York, NY, USA, 1995.

[Den13] Nicolas Denz. *Process-oriented analysis and validation of multi-agent-based simulations*. PhD thesis, Uni Hamburg, 2013.

[DGJ+16] Nondini Das, Suchita Ganesan, Leo Jweda, Mojtaba Bagherzadeh, Nicolas Hili, and Juergen Dingel. Supporting the model-driven development of real-time embedded systems with run-time monitoring and animation via highly customizable code generation. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, MODELS '16, pages 36–43, New York, NY, USA, 2016. ACM.

[DKM+10] Valentin Dallmeier, Nikolai Knopp, Christoph Mallon, Sebastian Hack, and Andreas Zeller. Generating test cases for specification mining. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 85–96, New York, NY, USA, 2010. ACM.

[dLGC15] Juan de Lara, Esther Guerra, and Jesús Sánchez Cuadrado. *Model-driven engineering with domain-specific metamodelling languages*, volume 14. SoSyM, 2015.

[dMvdAW03] A.K.A. de Medeiros, W.M.P. van der Aalst, and A.J.M.M. Weijters. Workflow mining: Current status and future directions. In *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE, volume 2888 of Lecture Notes in Computer Science*, pages 389–406. Springer-Verlag, 2003.

[DvdAtH05a] Marlon Dumas, Wil M. van der Aalst, and Arthur H. ter Hofstede. *Process-aware Information Systems: Bridging People and Software Through Process Technology*. John Wiley & Sons, Inc., New York, NY, USA, 2005.

[DvdAtH05b] Marlon Dumas, Wil M. van der Aalst, and Arthur H. ter Hofstede. *Process-aware Information Systems: Bridging People and Software Through Process Technology*. John Wiley & Sons, Inc., New York, NY, USA, 2005.

[EAD] Enterprise architect. product details. `http://www.sparxsystems.com/products/ea/index.html`. Accessed: 2018-05-04.

[EMF] Eclipse modeling framework (emf), howpublished = https://www.eclipse.org/modeling/emf/, note = Accessed: 2018-05-15.

[FPSS96] Usama M. Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. Advances in knowledge discovery and data mining. chapter From Data Mining to Knowledge Discovery: An Overview, pages 1–34. American Association for Artificial Intelligence, Menlo Park, CA, USA, 1996.

[GMC+92] C. Lee Giles, Clifford B. Miller, Dong Chen, Hsing-Hen Chen, Guo-Zheng Sun, and Yee-Chun Lee. Learning and extracting finite state automata with second-order recurrent neural networks. *Neural Computation*, 4(3):393–405, 1992.

[HMF+15] T. Hartmann, A. Moawad, F. Fouquet, G. Nain, J. Klein, and Y. Le Traon. Stream my models: Reactive peer-to-peer distributed models@run.time. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, pages 80–89, Sept 2015.

[HPE+16] Rogardt Heldal, Patrizio Pelliccione, Ulf Eliasson, Jonn Lantz, Jesper Derehag, and Jon Whittle. Descriptive vs prescriptive models in industry. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, MODELS '16, pages 216–226, New York, NY, USA, 2016. ACM.

[HSM01] David J. Hand, Padhraic Smyth, and Heikki Mannila. *Principles of Data Mining*. MIT Press, Cambridge, MA, USA, 2001.

[Ins] Introduction to instrumentation and tracing, howpublished = https://msdn.microsoft.com/en-us/library/x5952w0c(v=vs.85).aspx, note = Accessed: 2018-05-22.

[JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Atl: A model transformation tool. *Science of Computer Programming*, 72(1):31 – 39, 2008. Special Issue on Second issue of experimental software and toolkits (EST).

[JJLFA13] Sander J. J. Leemans, Dirk Fahland, and Wil M. P. Aalst. Discovering block-structured process models from event logs - a constructive approach. pages 311–329, 01 2013.

[JJLFA14] Sander J. J. Leemans, Dirk Fahland, and Wil M. P. Aalst. Discovering block-structured process models from event logs containing infrequent behaviour. volume 171, pages 66–78, 05 2014.

[KÖ6] Thomas Kühne. Matters of (meta-) modeling. *Software and Systems Modeling (SoSyM)*, 5(4):369–385, December 2006.

[KÏ16] Thomas Kühne. Unifying explanatory and constructive modeling: Towards removing the gulf between ontologies and conceptual models. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, MODELS '16, pages 95–102, New York, NY, USA, 2016. ACM.

[Kle09] Anneke Kleppe. *Software Language Engineering: Creating Domain-specific Languages Using Metamodels*. Addison-Wesley, Upper Saddle River, NJ, 2009.

[KMR00] Dean Karnopp, Donald MARGOLIS, and Ronald Rosenberg. System dynamics: "modeling and simulation of mechatronic" systems. 01 2000.

[Kra15] L. Krause. *Microservices: Patterns and Applications: Designing Fine-Grained Services by Applying Patterns*. Lucas Krause, 2015.

[Lam78] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. pages 558–565, July 1978.

[LB17] Jens Lienig and Hans Bruemmer. *Fundamentals of Electronic Systems Design*. Springer Publishing Company, Incorporated, 1st edition, 2017.

[LBK14] Jay Lee, Behrad Bagheri, and Hung-An Kao. A cyber-physical systems architecture for industry 4.0-based manufacturing systems. *SME Manufacturing Letters*, 3, 12 2014.

[LFvdA14a] S. J. J. Leemans, D. Fahland, and W. M. P. van der Aalst. Discovering block-structured process models from incomplete event logs. In *Business Process Management Workshops: BPM 2013 International Workshops, Revised Papers [Lecture Notes in Business Information Processing, Volume 171]*, Lecture Notes in Computer Science, pages 91–110. Springer, 2014.

[LFvdA14b] Sander J. J. Leemans, Dirk Fahland, and Wil M. P. van der Aalst. Process and deviation exploration with inductive visual miner. In *BPM*, 2014.

[LSVH14] Christoph Legat, Daniel Schütz, and Birgit Vogel-Heuser. Automatic generation of field control strategies for supporting (re-)engineering of manufacturing systems. *Journal of Intelligent Manufacturing*, 25(5):1101–1111, Oct 2014.

[Luc01] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[Lud02] Jochen Ludewig. Modelle im software engineering - eine einführung und kritik. In Martin Glinz and Günther Müller-Luschnat, editors, *Modellierung*, volume 12 of *LNI*, pages 7–22. GI, 2002.

[Mao09] Shahar Maoz. Using model-based traces as runtime models. *Computer*, 42(10):28–36, October 2009.

[Mic] Improve time to market with microservices, howpublished = https://www.ibm.com/cloud/garage/architectures/microservices, note = Accessed: 2018-05-15.

[MW16] Alexandra Mazak and Manuel Wimmer. Towards liquid models: An evolutionary modeling approach. In *Proceedings of the 18th IEEE Conference on Business Informatics (CBI 2016)*, pages 1–9, 2016. IEEE Conference on Business Informatics (CBI 2016), Paris, France; 2016-08-29 – 2016-09-01.

[MWPB18] Alexandra Mazak, Manuel Wimmer, and Polina Patsuk-Bösch. *Execution-Based Model Profiling*, pages 37–52. 01 2018.

[Occ] William of ockham, howpublished = https://plato.stanford.edu/entries/ockham/#4.1, note = Accessed: 2019-11-17.

[OCL] About the object constraint language specification version 2.4, howpublished = https://www.omg.org/spec/ocl/about-ocl/, note = Accessed: 2018-05-22.

[Pro] Prom tools, howpublished = http://www.promtools.org/doku.php, note = Accessed: 2018-05-09.

[Rasa] A 15 pound computer to inspire young programmers. http://www.bbc.co.uk/blogs/thereporters/rorycellanjones/2011/05/a_15_computer_to_inspire_young.html. Accessed: 2018-05-06.

[Rasb] Meet the raspberry pi. https://projects.raspberrypi.org/en/projects/raspberry-pi-getting-started/3. Accessed: 2018-05-06.

[Rasc] Raspberry pi 2 model b, howpublished = https://www.raspberrypi.org/products/raspberry-pi-2-model-b/, note = Accessed: 2018-05-08.

[Rasd] Raspberry pi gpio. https://www.raspberrypi.org/documentation/usage/gpio/. Accessed: 2018-05-08.

[Rase] Raspberry pi sold over 12.5 million boards in five years, howpublished = https://www.theverge.com/circuitbreaker/2017/3/17/14962170/raspberry-pi-sales-12-5-million-five-years-beats-commodore-64, note = Accessed: 2018-05-08.

[RH09] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Softw. Engg.*, 14(2):131–164, April 2009.

[RMSvdA09] A. Rozinat, R. S. Mans, M. Song, and W. M. P. van der Aalst. Discovering simulation models. *Inf. Syst.*, 34(3):305–327, May 2009.

[RR98] Wolfgang Reisig and Grzegorz Rozenberg, editors. *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the Volumes Are Based on the Advanced Course on Petri Nets*, London, UK, UK, 1998. Springer-Verlag.

[SCS] Critical systems, howpublished = `http://iansommerville.com/software-engineering-book/web/critical-systems/`, note = Accessed: 2019-02-17.

[SZ15] M. Szvetits and U. Zdun. Reusable event types for models at runtime to support the examination of runtime phenomena. In *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 4–13, Sep. 2015.

[SZ16] Michael Szvetits and Uwe Zdun. Controlled experiment on the comprehension of runtime phenomena using models created at design time. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, MODELS '16, pages 151–161, New York, NY, USA, 2016. ACM.

[SZ17] M. Szvetits and U. Zdun. Automatic generation of monitoring code for model based analysis of runtime behaviour. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 660–665, Dec 2017.

[SZ18] M. Szvetits and U. Zdun. Architectural design decisions for systems supporting model-based analysis of runtime events: A qualitative multi-method study. In *2018 IEEE International Conference on Software Architecture (ICSA)*, pages 115–11509, April 2018.

[VDA12] Wil Van Der Aalst. Process mining. *Commun. ACM*, 55(8):76–83, August 2012.

[vdA16] Wil M. P. van der Aalst. *Process Mining: Data Science in Action*. Springer, Heidelberg, 2 edition, 2016.

[vdARS05] Wil M. P. van der Aalst, Hajo A. Reijers, and Minseok Song. Discovering social networks from event logs. *Computer Supported Cooperative Work (CSCW)*, 14(6):549–593, Dec 2005.

[vdASS11] W. M. P. van der Aalst, M. H. Schonenberg, and M. Song. Time prediction based on process mining. *Inf. Syst.*, 36(2):450–475, April 2011.

[vdAWM04] Wil van der Aalst, Ton Weijters, and Laura Maruster. Workflow mining: Discovering process models from event logs. *IEEE Trans. on Knowl. and Data Eng.*, 16(9):1128–1142, September 2004.

[VHLL15] Birgit Vogel-Heuser, Jay Lee, and Paulo Leitão. Agents enabling cyber-physical production systems. *Automatisierungstechnik*, 63:777–789, 2015.

[Vya13] Valeriy Vyatkin. Software engineering in industrial automation: State-of-the-art review. *Industrial Informatics, IEEE Transactions on*, 9:1234–1249, 08 2013.

[WA03] A.J.M.M. Weijters and W.M.P. Aalst, van der. Rediscovering workflow models from event-based data using little thumb. *Integrated Computer-Aided Engineering*, 10(2):151–162, 2003.

[WAM06] A. Weijters, Wil M. P. Aalst, and Alves Medeiros. *Process Mining with the Heuristics Miner-algorithm*, volume 166. 01 2006.

[WR11] A. J. M. M. Weijters and J. T. S. Ribeiro. Flexible heuristics miner (FHM). In *CIDM*, pages 310–317. IEEE, 2011.

[WvdAWS07] L. Wen, Wil M. P. van der Aalst, J. Wang, and J. Sun. Mining Process Models with Non-Free-Choice Constructs. *Data mining and knowledge discovery*, 15(2):145–180, 2007.

[XES16] Ieee standard for extensible event stream (XES) for achieving interoperability in event logs and event streams. *IEEE Std 1849-2016*, pages 1–50, Nov 2016.

Appendices

№	System under study	Observed aspect	Algorithm (Plug-in)	Special conditions
Group 1				
1	TrafficLight	State changes	Alpha Miner	-
2	TrafficLight	State changes	Alpha++ Miner	-
3	TrafficLight	State changes	Heuristic Miner	-
4	TrafficLight	State changes	Inductive Miner	-
5	TrafficLight	State changes	Visual Inductive Miner	-
6	TrafficLight	Attribute value changes	Alpha++ Miner	-
7	TrafficLight	Attribute value changes	Heuristic Miner	-
Group 2				
8	TrafficLight	State changes	Alpha++ Miner	With noise
9	TrafficLight	State changes	Visual Inductive Miner	With noise, noise threshold 0.8
10	TrafficLight	State changes	Visual Inductive Miner	With noise, noise threshold 1.0
11	TrafficLight	State changes	Heuristic Miner	With noise, dependency threshold 90, relative to best threshold 5
12	TrafficLight	State changes	Heuristic Miner	With noise, dependency threshold 0, relative to best threshold 100

Group 3				
13	TrafficLightOO	State changes, Control	Heuristic Miner	-
14	TrafficLightOO	State changes, Control	Heuristic Miner	With noise
15	TrafficLightOO	State changes, Car	Heuristic Miner	-
16	TrafficLightOO	State changes, Pedestrian	Heuristic Miner	-
17	TrafficLightOO	State changes, Car	Heuristic Miner	With noise
18	TrafficLightOO	State changes, Pedestrian	Heuristic Miner	With noise
19	TrafficLightOO	State changes	Social network Miner	-
Group 4				
20	TrafficLight	Transition firing	Inductive Miner, Performance Analysis	Timing observations
21	TrafficLightOO	Transition firing, Control	Inductive Miner, Performance Analysis	Timing observations

Table 1: List of experiments

№	**System under study**	**Number of cases**	**Number of failures**	**Experiments**
1	TrafficLight	10	0	1, 2, 3, 4, 5, 6, 7, 20
2	TrafficLight	21	5	8, 9, 19, 11, 12
3	TrafficLightOO	10	0	13, 15, 16, 19, 21
4	TrafficLightOO	29	4	14, 17, 18

Table 2: List of observation models