Josep Carmona
Boudewijn van Dongen
Andreas Solti
Matthias Weidlich

Conformance Checking

Relating Processes and Models

Springer

Conformance Checking

Josep Carmona • Boudewijn van Dongen •
Andreas Solti • Matthias Weidlich

Conformance Checking

Relating Processes and Models

Springer

Josep Carmona
Department of Computer Science
Universitat Politècnica de Catalunya
Barcelona, Spain

Boudewijn van Dongen
Dept of Mathematics and Computer Science
Technische Universiteit Eindhoven
Eindhoven, The Netherlands

Andreas Solti
Institute for Information Business
Vienna University of Economics
and Business
Vienna, Austria

Matthias Weidlich
Department of Computer Science
Humboldt-Universität zu Berlin
Berlin, Germany

ISBN 978-3-319-99413-0 ISBN 978-3-319-99414-7 (eBook)
https://doi.org/10.1007/978-3-319-99414-7

Library of Congress Control Number: 2018956267

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Foreword

Conformance checking is an important—but also challenging—topic in process mining. Most people who see process mining for the first time are dazzled by the process discovery capabilities of today's process mining tools. However, when people really start to use process mining, more detailed questions emerge, and it is no longer sufficient to look at fancy process diagrams composed of boxes and arrows. Initially, only academic tools like ProM supported conformance checking. However, in the last two years, also commercial tools started to support conformance checking, and this is a great development. Moreover, it is wonderful that there is now this book dedicated to conformance checking written by my dear colleagues Josep Carmona, Boudewijn van Dongen, Andreas Solti, and Matthias Weidlich. The book nicely complements my earlier Springer book "Process Mining: Data Science in Action" by diving deep into the topic of conformance checking and giving it the attention it needs.

Process mining relates event data and process models and bridges the gap between data science and process science. It is a very natural, and seemingly obvious, combination of forces. Most organizations have a continuous desire to improve their processes and with the availability of data it makes sense to do this in an evidence-based manner. On the one hand, it aligns well with the current excitement about Big data, machine learning, and artificial intelligence. On the other hand, the techniques used are very different as demonstrated in this book. We already had two "AI Winters" (1974–1980 and 1987–1993), and I'm sure the third one is not far away. Therefore, it is important to see that process mining is very different and also very concrete (as opposed to some of the hyped AI topics). When people see process mining for the first time, they wonder why they did not hear about this before. This book will hopefully contribute to a better positioning of the topic.

Process mining research at Eindhoven University of Technology started in 1999. The focus in the first year was on process discovery, and algorithms like the α-miner and the heuristic miner were developed. We started to systematically work on conformance checking in 2004 in the context of Anne Rozinat's PhD thesis. Before this, we only compared "footprint matrices" and implemented ad hoc fitness

functions in the context of genetic algorithms. Anne's seminal work resulted in the token-based replay techniques also described in this book. Conformance was expressed in terms of missing and remaining tokens. Several other notions were developed in this period (e.g., variants of what is today called precision) and token-based replay became the standard, not only for conformance checking, but also for decision mining and performance analysis. However, as already pointed out in Anne Rozinat's PhD thesis, token-based replay has several limitations. The next major breakthrough was realized in the context of Arya Adriansyah's PhD work (2009–2013). He developed the seminal notion of alignments which is one of the main topics in this book. Alignments provide a "closest path" through the process model. It can be turned into a shortest-path problem by creating a kind of synchronous product between the event log and the process model.

Part II of this book uses alignments as a central notion and takes a "deep dive" into the world of conformance checking (including linear algebraic techniques, heuristic search, and decompositions). The different chapters demonstrate that conformance checking is very different from traditional model-based process analysis (e.g., simulation and other business process management techniques) and data-centric analysis techniques such as machine learning and data mining. The confrontation between event data (i.e., recorded behaviour) and process models (hand-made or discovered automatically) triggers many interesting and highly relevant questions.

Part III of the book zooms in on the applications of process mining and corresponding tools. Aligning recorded behaviour and modelled behaviour is relevant in any domain where event data are collected, including logistics, manufacturing, finance, healthcare, customer relationship management, e-learning, and e-government. Moreover, decision mining and bottleneck analysis depend on it. For example, the profession of auditors will dramatically change because of conformance checking. Undoubtedly, conformance checking will be in the toolbox of future generations of analysts, consultants, managers, process owners, software engineers, etc.

How about the future of conformance checking? What lies ahead? There is broad consensus among process mining experts that conformance checking will become more important. This provides two main challenges: (1) How to improve the performance of conformance checking when models and logs get bigger? (2) How to balance between precision and deliberate vagueness? Compared to state-of-the-art discovery techniques, conformance checking techniques tend to be time-consuming. It is not always required to compute optimal alignments and there is room for approximations or alternative conformance notions. Existing conformance approaches also require formal models. However, this is in stark contrast with the informal models generated by commercials tools. When there is not enough evidence in the data, one should not show very strict and "binary" process-model constructs. However, if there is enough evidence in the data, one should not use process-model constructs without clear semantics. Some of the tools resolve this by having two types of process models: one for discovery and one for conformance checking. Obviously, this is only a temporary solution. Process

mining tools need to support hybrid process models that are precise when possible and vague when needed. Such models should also take into account probabilities of process paths. These challenges show that conformance checking continues to provide wonderful opportunities for researchers.

I would like to congratulate Josep, Boudewijn, Andreas, and Matthias with doing an excellent job in joining the state-of-the-art approaches in a single book. The book will be a reference for people interested in the intricate relationship between models and reality. Enjoy reading!

RWTH Aachen University
Aachen, Germany
March 2018

Prof. Dr. Ir. Wil van der Aalst

Preface

A model is an artefact to represent a specific concept. It maps properties of the concept into some abstract representation, driven by the purpose of the model. As such, models represent a universal means to capture concepts in a wide range of domains: In architecture, a model may represent the outline of a building structure; in engineering, a model may capture the mechanics of a pump; and in computer science, a model of a processor may correspond to the hardware supporting a particular system infrastructure. Regardless of the specific concept that is captured, a model always has a pragmatic feature. It is created for a specific purpose, it may be the design or construction of the concept or its analysis and exploration.

In this book, we consider models of processes. A process is a set of activities that are executed in a coordinated manner to achieve a certain goal. It is very hard to not encounter a process in most of the things we do in our daily life: *taking the car to go to work*, *extracting money from an ATM*, *ordering goods using an online shop*, or *visiting the doctor*—all are examples of processes. Each of them involves activities, whereas the ability to reach a specific goal depends on their effective coordination.

Process models are a valuable source of information. They may be the result of a significant effort in formalising a complex process. This formalisation encompasses the decision which properties of a process should be mapped into the model. For processes, this decision primarily relates to the choice of the activities to consider and the possible ways in which they can be executed. However, further information may be considered, such as the involved actors, the data used or generated, or the timing of the activities. Modelling a real process can only be done if some abstraction is applied. To escape from unnecessary details and retain the essence of the process, only some process properties are mapped to a model. Hence, by definition, modelling incurs some loss of information.

Yet, the fact that not all information about a process can be captured in a model is not the only reason that adds to the uncertainty in the relation between a process model and the process itself. A model may also have been formalised incorrectly, simply became outdated with respect to the process, or describe behaviour that deviates from the actual process. In the past, this uncertainty in the relation between a process and a model thereof has been a critical problem, which was hard to detect.

The massive availability of data, however, has changed this situation dramatically. Nowadays, systems record the footprints left by executions of the process, which enable data-driven analysis. Therefore, the recorded behaviour of a process, as manifested in so-called event logs, is now available as an important source of information.

Conformance checking relates modelled and recorded behaviour of a process to each other. It provides techniques and methods to compare and analyse observed instances of a process in the presence of a model, independent of the origin of that model. Conformance checking therefore aims at answering questions, such as:

- Is the process being executed as it is documented in a model?
- Is the model of a process still up-to-date?
- Have there been violations of rules in the recorded instances of a process?
- How much flexibility is allowed in the execution of a process?

The aim of this book is to introduce readers to the field of conformance checking as a whole and outline the foundations of the relation between modelled and recorded behaviour. The book strives for an overview of the essential techniques and methods in this field on the intuitive level as well as an understanding through a precise formalisation of fundamental notions of conformance checking.

Structure of This Book

This book is structured in three parts, each being guided by a different question and therefore a different approach to the field of conformance checking:

Part I: What are the main ideas behind conformance checking? Readers shall benefit from the first part of the book as a concise and comprehensive overview of the essential ideas to relate modelled and recorded behaviour to each other. This part serves as a reference to assess how conformance checking efforts could be applied in particular domains. Outlining these ideas on the intuitive level, this first part is of interest to students, academics, and practitioners alike, who have a technical background, but are new to the field of conformance checking.

Part II: What are the state-of-the-art methods for conformance checking? The second part of the book aims at providing readers with detailed insights into algorithms for conformance checking. This includes the most commonly used formal notions for conformance checking and their instantiation for specific analysis questions. This part aims at enabling readers to initiate research in conformance checking. While all basic definitions are given explicitly, readers benefit from prior knowledge in formalisms for process modelling, such as Petri nets.

Part III: What are the applications that build on top of conformance checking? The third part of the book presents applications that help to make sense of conformance checking results, thereby providing a necessary next step to increase the value of a process model. Specifically, such applications help to interpret the

results of conformance checking and incorporate them through enhancement and repair techniques. This includes remarks on conformance checking software.

Teaching Using This Book

In the light of the above structure, three itineraries are suggested when using the book as a reference for teaching:

- Chapters 1–4 and 12: These chapters give a general description of conformance checking, including remarks on tool support, from a practical perspective.
- Chapters 5–9: This part gives an algorithmic view on how to relate modelled and recorded behaviour, focusing on formal foundations of conformance checking.
- Chapters 1–6 and 10–12: An end-to-end view on conformance checking and its applications, covering formal foundations on the basic level.

The chapters of the book incorporate material in the following form:

- Each chapter contains teaching material in the form of exercises to allow for effective learning of the theoretical concepts introduced.
- For those chapters amenable for tool practising, a section named *In the Lab* is included, which introduces tools and/or datasets for exploring the contents of the chapter on real data.

Website of the book

Check out the website for teaching materials for the book. Throughout the book, you find links to support lab sessions for specific chapters.

http://www.conformancechecking.com/CC_book

Barcelona, Spain — Josep Carmona
Eindhoven, The Netherlands — Boudewijn van Dongen
Vienna, Austria — Andreas Solti
Berlin, Germany — Matthias Weidlich

Acknowledgements

The idea of writing a book on conformance checking started back in early 2016, in the course of several visits of Boudewijn to Josep's group in Barcelona, and to Andreas' group in Vienna. While in Barcelona, apart from working hard on writing papers (and performing quite a few experiments on Boudewijn's tiny laptop), the discussion on the lack of a textbook setting the basis for the field of conformance checking was recurrent. On one of these visits, the paper from Andreas and Matthias on generalized conformance checking was just out from being published in the BPM conference in Rio de Janeiro. That was the trigger to invite Andreas and Matthias to join the team and make the formal step to write the book.

Writing a book is always a hard task, and the long trip to accomplish it was not always smooth. Throughout the journey, 75% of the authors went through promotions or job changes. But in our trip we were never alone. In here, we would like to list all the people and institutions that have been supporting us for this endeavour.

First we would like to thank the early reviewers, who really did a great job in guiding us for redirecting parts of the first draft of the book: Arya Adriansyah, Henrik Leopold and Seppe vanden Broucke. They provided very useful insights, constructive criticism and encouragement to continue working (even more) after reaching the first draft. In a second iteration, Eric Verbeek's review was amazingly deep and insightful, that allowed us to amend many inconsistencies and better balance the content of several chapters.

The research community of conformance checking is growing considerably. The results presented in this book are not limited to the work from the authors. Instead, many members of our community contributed to the body of work available which allowed us to write this textbook: Wil van der Aalst, Arya Adriansyah, Abel Armas-Cervantes, Bart Baesens, Thomas Baier, Seppe vanden Broucke, Joos Buijs, Andrea Burattin, Thomas Chatain, Raffaele Conforti, Jonathan Cook, Massimiliano De Leoni, Benoît Depaire, Claudio Di Ciccio, Chiara Di Francescomarino, Marlon Dumas, Dirk Fahland, Avigdor Gal, Luciano García-Bañuelos, Arthur H. M. ter Hofstede, Gert Janssenswillen, Toon Jouck, Maria Teresa Gómez López, Marcello La Rosa, Sander Leemans, Henrik Leopold, Xixi Lu, Fabrizio Maria Maggi,

Felix Mannhardt, Ronny Mans, Jan Mendling, Marco Montali, Jorge Munoz-Gama, Artem Polyvyanyy, Hajo Reijers, Anne Rozinat, Arik Senderovich, Natalia Sidorova, Farbod Taymouri, Jan Vanthienen, Eric Verbeek, Ingo Weber, Jochen De Weerdt, Sebastiaan J. van Zelst and many others.

Also especially we would like to thank Wil van der Aalst for accepting to write a foreword for this book. We feel that textbooks like the one presented here may complement Wil's bestseller on process mining, a book that has enabled the community to not only consolidate, but also to grow considerably in the last decade.

Many thanks to Springer-Verlag for giving us the opportunity to publish this book. The interaction with Ralf Gerstner was helpful to understand the full process of writing a book. Ralf's patience and support was crucial in some phases of the process.

We are also indebted to the support from our institutions, namely Universitat Politècnica de Catalunya, Technische Universiteit Eindhoven, Wirtschaftsuniversität Wien, and Humboldt-Universität zu Berlin. This book was written under the support of the Spanish Ministry for Economy and Competitiveness (MINECO) and the European Union (FEDER funds) via grants COMMAS (ref. TIN2013-46181-C2-1-R) and GRAMM (TIN2017-86727-C2-1-R). Matthias is grateful for the support by the German Research Foundation (DFG) as part of the Emmy Noether Programme.

Finally, we acknowledge the warm support received from our families, whose names are listed now: Bruno and Gio (Josep), Boris, Emma and Maja (Boudewijn), Sie-Youn (Andreas), Emiliana, Jonathan, Benjamin and Anna (Matthias).

This book was written in the period October 2016–April 2018.

Barcelona, Spain — Josep Carmona
Eindhoven, The Netherlands — Boudewijn van Dongen
Vienna, Austria — Andreas Solti
Berlin, Germany — Matthias Weidlich
September 2018

Contents

Part I
Basics of Conformance Checking

Preface to the First Part

The first part of this book, Chapters 1–4, presents an informal overview of the field of conformance checking. In the first chapter, the book defines the context of conformance checking, outlines the drivers for conformance analysis, and reflects on the relevance of conformance checking in practice.

Chapter 2 then moves to basic notions of process modelling and event logs. Again, the focus is to provide an intuitive overview rather than a comprehensive formalisation. Based thereon, Chapter 3 describes two main dimensions along which modelled and recorded behaviour are related: *Fitness* relates to the share of recorded behaviour that is captured in a model, whereas *precision* quantifies the share of modelled behaviour that is covered by the instances of a process as recorded in an event log.

The first part of the book closes with three main algorithmic perspectives for grounding conformance checking in Chapter 4. We discuss the main ideas of relating modelled and recorded behaviour based on rule checking, token replay, and alignments.

Chapter 1
Introduction to Conformance Checking

Conformance checking refers to the analysis of the relation between the intended behaviour of a process as described in a process model and event logs that have been recorded during the execution of the process. In this context, a process refers to a generic behavioural concept, i.e. the coordinated execution of a set of activities to reach a specific goal or outcome. These activities and their coordination is exactly what is captured by a model of the process. However, as soon as data signalling how a process is actually executed becomes available, typically in the form of event logs, the question of conformance emerges: how do the *modelled* behaviour of a process and its *recorded* behaviour relate to each other?

This chapter outlines the background of conformance checking. In Section 1.1 we first discuss the big picture of digital transformations of organizations and then in Section 1.2 review essential notions, such as processes, process models, and systems supporting the execution of processes. Reflecting on these notions, we derive drivers for conformance checking in Section 1.3. Next, in Section 1.4 we discuss a spectrum of conformance checking techniques, which puts the question of conformance into perspective, relating it to other questions on the interplay of processes, models, and event logs. Finally, in Section 1.5 we turn to the relevance of conformance checking and elaborate on common use cases and application domains.

1.1 The Big Picture: The Renewed Value of Process Models

Historically, process models have played an important role in organizations. They proved to be a key element for describing, analysing and monitoring the execution of the processes that structure the operations of organizations.

As part of digital transformation initiatives, the demand emerged to expose process models to the event data signalling the executions of the actual processes. Relating event data and process models enables unprecedented types of analysis,

J. Carmona et al., *Conformance Checking*,
https://doi.org/10.1007/978-3-319-99414-7_1

providing insights that are often beyond expectations. In this section, we show how this link provides a renewed value of process models in organizations, from three different perspectives.

Agile Compliance Management Exposing data to models represents an agile way to show compliance of operations to legal requirements. For example, the European Union recently approved the General Data Protection Regulation (GDPR), a regulation that establishes how data of EU citizens should be used by organizations. The GPDR applies to all organizations processing and holding the personal data of subjects residing in the European Union, regardless of the organization's location. Once the regulation starts to be enforced in practice, it will have a clear impact in organizations, and in the extreme case, may cause heavy fines in case of non-compliance. Organizations that store personal data of EU citizens will need to prove that their operations, as structured by processes, adhere to the GPDR.

Proving process compliance to the applicable regulations is a task that can be costly, slow and cumbersome, especially if it is done manually. Moreover, since processes evolve continuously in an organization, and laws change over time, compliance to regulations needs to be continuously assessed. Organizations that do not have an agile approach to face this issue will be penalised from several angles: adherence to law, competitiveness, adaptability to changes, etc.

Figure 1.1 depicts the added value of relating modelled and process behaviour, from the perspective of agility. If on the one hand, process models are shown to satisfy the regulations, and on the other hand, they are aligned to the process data and no deviations are detected, one can conclude on the absence of violations of the regulations for the underlying processes. This way, auditing of processes and regulations can be supported.

If instead of regulations, other aspects are enforced on top of process models, such as quality levels or an organization's policies, and process models are aligned to the corresponding event data, similar outcomes can be obtained: The quality of a certain service or the implementation of a particular policy can be ensured in the organization.

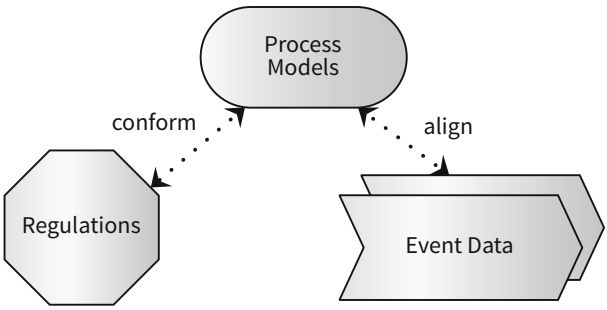


Figure 1.1 Process models as the key link between regulations and event data

Accurate Process Prediction Among the manifold use cases of process models within an organization, a technique that received significant attention recently is *predictive monitoring*: the prediction of quantifiable metrics of a running process instance. For example, using predictive monitoring, one can predict whether the current session of a client will end up with the purchase of an item or not, or if the remaining time of processing a loan application is expected to stay below a certain bound.

Process models can be obtained through manual design or by automated process discovery. The latter aims at the construction of a process model from an event log and respective techniques have received a lot of attention in recent years. Independent of the origin of the process model, its accuracy in reflecting the reality of the process execution cannot be assessed by inspecting only the respective model. If a model has been designed manually, which is often a tedious and error-prone task that may involve several persons across an organization, it may fail to capture all the possible variants, or may be defined at the wrong level of granularity. Likewise, process models learned from event data through discovery techniques, may be inaccurate for many reasons, among them noise or incompleteness in the data and the inability of a discovery algorithm to recognise particular relations between activities.

If models are inaccurate, however, the value of predictive monitoring is significantly hampered, as one cannot trust the estimations if the process model is disconnected from the reality of process execution. As such, it may happen that insights from predictive monitoring based on inaccurate process models may lead to misleading analysis results. Decisions made on the basis of such wrong insights may, in turn, have severe consequences for an organization. Relating the modelled and recorded behaviour of a process to each other, therefore, helps to improve the quality of predictive monitoring.

Intelligent Automation and Human Augmentation It is anticipated that Artificial Intelligence (AI) will have a considerable effect on jobs and business automation in the next few years. The main contributor to the net job growth resulting from this trend is expected to be *AI augmentation*. It refers to the combination of human and artificial intelligence, such that both complement each other to achieve elaborated reasoning. This way, it lays the foundations for advanced control and enactment of processes.

The augmentation of humans for decision support is therefore a tendency that will grow in the coming years. In order to exploit AI for this task, an understanding of regularities and general procedures, as well as of the handling of exceptional situations in processes is needed. Reliable, up-to-date process models may serve this purpose, as the explicit representation of a process used in augmentation. Based thereon, support of a process may move from predictive analysis as outlined above, to prescriptive analysis that supports humans in the enactment of a process. To realize this vision, process models need to be linked to data about current executions as well as to contextual information.

1.2 Setting of Conformance Checking

Conformance checking is about the relation between a model and an event log of a process. A process describes the behaviour of a system by specifying a set of activities, elementary units of work, along with causal dependencies that govern their execution. As such, a process is executed by instantiating activities, whereas the resulting activity executions are coordinated. Activities are conducted sequentially or concurrently to each other; their execution is dependent on explicit decisions; and parts of a process may even be repeated multiple times. Yet, execution of activities is coordinated within a specific scope, referred to as a *case*. A case represents an instance of the process and is defined by all activity executions that relate to one specific trigger or input to the system whose behaviour is described by the process. This terminology is summarized in the middle part in Figure 1.2.

Adopting this view, a process is a rather abstract concept that is used to describe phenomena in diverse domains. In finance, a process describes how a loan application is processed; in logistics, a process represents a transportation chain used to ship a good; and in healthcare, a process outlines how a patient is treated.

Considering the example of a bank offering loans in some more detail, a process defines how a loan application is handled once it has been submitted, with the outcome being its positive or negative assessment. Each loan application then represents a case, as part of which the activities of the process are executed. A first activity is a *check* for eligibility of the respective applicant. Subsequently, a decision is taken, which determines whether an application is *declined* or *accepted*. If accepted, the application is *finalised* and an order is *selected* for further processing.

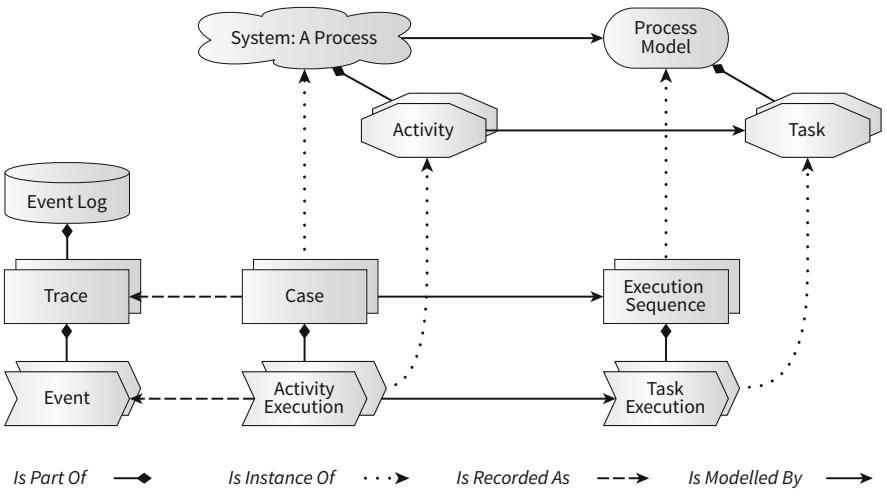


Figure 1.2 Terminology of processes, event logs, and models

The notion of a process per se does not postulate a specific degree of *automation*. The activities of a process can relate to units of work that are conducted manually, or correspond to functionality that is directly available as part of some software infrastructure. The same holds true for the coordination of activity executions as part of a case. There may be a manual assignment of workers to activities (or activity executions, respectively) that is driven by the causal and temporal dependencies of a process. The other extreme are dedicated software systems that handle this coordination, based on a formal specification of a process.

For the above loan application process, as we encountered it in a large European bank, the initial eligibility check is an example for an automated activity—a set of predefined plausibility rules is checked for the applicant's data. While the activities of declining or accepting the application are also conducted automatically, a worker needs to manually finalise the application in case of acceptance.

Process Modelling A process model captures the activities and execution dependencies of a process in a conceptual model. As such, a process model is always an abstraction of a process. It defines a projection and aggregation of a process' properties, dropping information that is considered irrelevant and aggregating information to obtain a general understanding of a process. These abstractions necessarily imply a loss of information, which is intended and motivated by the purpose of the model.

In the above example of a loan application, information on storing the applicant's personal data may be deemed irrelevant for the purpose of defining how the application is processed. In contrast, the creation of an offer in reference to the application is essential to understand which artefacts are created as part of the process.

A process models consists of tasks, each task representing an activity of the process, and their respective execution dependencies. A process model, therefore, aims at an overarching view on a process and generalizes individual cases. As outlined in the right-hand part of Figure 1.2, however, a process model may also be instantiated. That is, an execution sequence of a process model is built of task executions, thereby providing a conceptual representation of cases and activity executions of the process.

The aforementioned abstractions are realized by means of *process modelling languages* that provide a vocabulary for the definition of process models. In essence, such a language comprises concepts to capture tasks and execution dependencies, whereas the concepts may differ drastically from one language to another. The spectrum of concepts for the definition of tasks reaches from a simple identifiers for some unit of work to its full-fledged definition that can be run automatically in an IT infrastructure. Similarly, there is a wide range of approaches to specify execution dependencies, including procedural and declarative characterizations of the process.

A process model that describes how a loan application is handled is illustrated in Figure 1.3. This model, captured in the Business Process Model and Notation (BPMN), serves for illustration in the remainder of this book. In BPMN, tasks

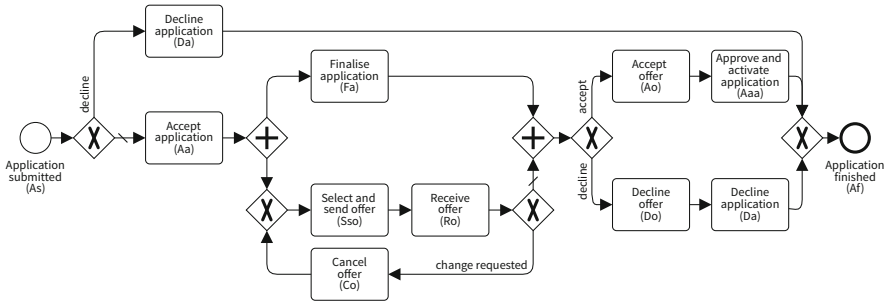


Figure 1.3 Example process model of a loan application process in BPMN

are represented by rectangles; instantaneous events are visualized by circles (in Figure 1.3 they start or end the process); and execution dependencies are modelled by control flow arcs and diamond-shaped nodes, called gateways. The semantics of such a gateway determines the exact behaviour of a process, e.g., whether incoming arcs are synchronised (AND-gateway with a 'plus' symbol) or not (XOR-gateway with a 'cross' symbol); or whether outgoing arcs are enabled concurrently (AND-gateway) or mutually exclusive to each other (XOR-gateway).

According to this model, a submitted application is either accepted or rejected, based on the aforementioned rules to check plausibility of the applicant's data. An accepted application is finalised by a worker, in parallel with the offer process. For each application, an offer is selected and sent to the customer. The customer reviews the offer and sends it back. If the offer is accepted, the process continues with the approval of the application and the activation of the loan. If the customer declines the offer, the application is also declined and the process ends. However, the customer can also request a new offer, in which case the offer is cancelled and a new offer is sent to the customer.

A process model provides an abstraction of a process to serve a specific purpose. Only properties of the process that are relevant with respect to this purpose—also known as the *goal of modelling*—are mapped to the model. The purpose of a process model is commonly rooted directly in the application domain. It spans business purposes such as process documentation for staff training, resource planning, or organizational redesign. However, process models are also employed in the automation of various processes. They serve as a requirements specification that guides software development or system selection, or may even represent implementation artefacts that serve as a blueprint of how a system shall execute a process.

Independent of the specific purpose of a process model, there are a few aspects that differentiate drivers for process modelling on the general level. First and foremost, models deviate in the *perspectives* of a process that are captured. While task and control flow routing are the most prominent building blocks of a process model, further perspectives may be considered. Examples include concepts to specify time-outs, the handling of exceptional behaviour, or the messages a process

exchanges with its environment; data input and output, not only of single activities, but of the process altogether; and resources that are needed for the execution of a process, reaching from organizational entities (staff roles, departments) to technical infrastructure (machines, information systems).

Beyond the question of *what* properties to capture in the model, it is important to identify *when* the process is mapped to a model. The point in time to which the process model shall relate is important as it acknowledges that processes tend to change frequently in many domains. In particular, an *as-is model* models the present, or near past of a process, whereas a *to-be model* describes a hypothetical process, e.g., one that shall be realized in the future.

Process Execution The execution of a process is often supported by various types of information systems. Even if the execution of a process is done completely manually, information that is created or consumed during process execution (e.g., loan applications and corresponding offers) is often kept in some database or document management system. Consequently, the conduct of a process may be reflected in information systems, even if activities and their coordination are not automated.

Beyond such indirect support of a process, dedicated types of information systems exist to foster process automation. A *process-aware information system* (PAIS) aims at supporting the coordination of activity executions based on an explicit formalisation of the respective causal and temporal dependencies, commonly given as a process model. The use of a PAIS thereby represents a common purpose of process modelling, which, as detailed above, determines what to capture in the process model and how to select the appropriate point in time to which the model shall relate. Again, we note that the use of a PAIS does not necessarily mean that activities are automated—they may still be subject to manual execution, whereas a PAIS supports the coordination of the execution, e.g., by selecting activities that could be executed next. That way, the conduct of a process is directly reflected in an information systems.

Unlike traditional *workflow management systems (WfMS)*, process-aware information systems do not necessarily enforce the coordination of activities as defined by the tasks of a process model. A WfMS directly *implements* the causal and temporal dependencies defined by the model, creating cases according to the provided blueprint. Process-aware information systems, in turn, *support* the conduct of a process based on these causal and temporal dependencies, but commonly leave control on which activity to execute to process workers. As a consequence, they enable workers to deviate from the process specification given by the underlying process model, thereby providing a degree of flexibility that is crucial in many application domains. For instance, in the above loan application process, see Figure 1.3, a worker may decide to cancel an offer before the respective customer has sent back the offer. Although this would not be in line with the process as defined in the model, such a deviation may be well-motivated based on contextual factors that are not captured in the process model.

Once a process is supported by information systems, details on how the process is executed are commonly available in the form of event data. Ideally, this data assumes the form of an *event log*, i.e., a collection of *events* that indicates (1) at what point in time, (2) which activity was executed, (3) for which case. As such, an event log represents the recorded behaviour of a process. Events can be separated based on the cases as part of which the respective activities have been executed. This results in event sequences, called *traces*, which represent the behaviour recorded for particular cases of the process. As indicated in Figure 1.2, a trace therefore is a recorded representation of a case of the process, very much like an execution sequence of a process model is a modelled representation of a case. Consequently, the behaviour recorded in an event log can be related to the behaviour captured in a process model.

For illustration, we consider the loan application process as detailed before. Once this process is supported by a PAIS, event data on the execution of this process is collected and may comprise information on the time a particular activity has been executed for a case, the latter being related to a client, the requested amount, and prospective duration of the loan. An example of such an event is given as follows:

Timestamp	Activity	Case	Client	Amount	Duration
9:17, May 2, 2017	Accept Application (*Aa*)	926	Marta Smith	€15,000	2 years

Implementations of such a loan application process produce event logs of considerable size in practice. For instance, taking the data of a large European bank as an example, for the respective scenario and a time frame of 6 months, around 10,000 submitted applications have been recorded. The resulting event log comprises more than 200,000 events, which precludes any manual analysis of the respective data.

While PAIS can be expected to create event logs explicitly, other types of information systems may record data that can be transformed into an event log. For example, functional calls in enterprise systems or transactional logs of database management systems may be the starting point for the construction of an event log. The respective transformations may be non-trivial, though, and potentially have to cope with complex and ambiguous relations between the recorded data and the executions of activities for a particular case.

1.3 Motivation for Conformance Checking

The availability of footprints of a process in terms of an event log, opens the door for relating the actual process, from the perspective of the traces in the event log, and the process model, which describes and abstracts the underlying process. Although both representations, the event log and model, talk about the same thing, the real process,

establishing a relation between them is crucial for understanding how processes are executed, and how far apart the recorded reality and a describing model of a process are from each other. Conformance checking techniques are devoted to automatically compute this relation.

At its core, conformance checking needs to assess the relation between a trace and a process model. For instance, for the model of Figure 1.3, and the trace $\langle As,Da,Af \rangle$, which describes a loan application that has been rejected, an execution sequence of the model exists that completely reproduces the observed trace. This information, which certifies that the model and the trace agree, is of great value: It acknowledges that the trace satisfies the control flow dependencies established in the process model. The consequences for this agreement are diverse:

- It enforces the validity of the process model in describing the recorded reality.
- It classifies the recorded trace as compliant.
- It enables usage of the model to make predictions about the future for running cases, e.g., remaining time or cost.

These consequences denote new knowledge that is available to organizations. Symmetrically, deviations may exist between a trace and a process model. For instance, using again the process model of Figure 1.3, consider the trace $\langle As,Aa,Sso,Ro,\ Do,Da,Af \rangle$, which represents a declined loan application that has not been finalised. There exists a deviation between the process model and the trace, since the model requires to execute the activity to finalise the application (Fa) sometime before the offer is declined (Do). Although important, this deviation is somehow mild in the sense that a formality (finalising the application) has not been performed for a rejected loan application.

However, deviations can also incur a serious problem for an organization; consider now the trace $\langle As,Aa,Sso,Ro,Fa,Do,Aaa,Af \rangle$. This trace represents an application that, in spite of taking the decision of declining the offer, has been finally approved and activated. One can see that the process model of Figure 1.3 cannot reproduce this trace: On the one hand, it requires that the application is declined (Da), and on the other hand, it cannot reproduce that the application is approved and activated (Aaa). Clearly, a root cause analysis of this deviation may be crucial for an organization to detect serious inconsistencies in the way a process is performed. This analysis often goes beyond control flow, i.e., analysing the additional information available in the event log to determine a meaningful explanation for the deviation encountered. There may be different explanations for a particular deviation, such as:

- A lack of coordination between different departments or involved persons.
- Wrong recording of activity executions.
- Corruption in the recorded event data.
- Decisions taken that violate a company's rules.

In fact, these explanations need not to be mutually exclusive, e.g., data corruption (e.g., adding a digit to the salary of applicant when processing loan requests) may cause a wrong decision that violates the company's rules. It is clear that situations like the one illustrated above can incur very high costs for an organization.

In summary, detecting deviations such as the previous one is of paramount importance. The consequences of detecting deviations are:

- The validity of the process model in describing the recorded reality is weakened.
- An alarm is raised for the behaviour observed for a case.

Likewise, this new information may be fundamental for an organization. For instance, deviations may hint at situations for which the system controlling the process is not operating as expected. Yet, there may also be an issue with the way the progress of the process is recorded. Finally, the model may not correctly describe the reality, e.g., due to an evolution of the process that has not been yet incorporated into the process model.

Excursion 1

Deviations: the good, the bad (and the ugly)

From the previous example, one may think that detecting deviations is always a bad sign, either in the recorded behaviour or in the modelled behaviour. This is often the case: deviations imply that something has not been performed as expected, pinpointing a situation that must be resolved (either in the model or in the system). However, deviations can also be positive, representing a sign of flexibility that may be crucial in some contexts. For instance, the steps to follow in a surgery may need to be significantly modified in case of a sudden deterioration of the patient health. These *breaking-the-glass* situations are common in several contexts, and need not to be penalised.

Hence, by analysing deviations that denote weaknesses in the process, its recording, or the process model, an organization can continuously improve its operations. This ambitious goal cannot be achieved without automation; think of a large company having several units across the globe, each one having several departments that need to be coordinated to operate the company's processes. Or think of a popular sales company over the Internet, which stores terabytes of process information daily. Or even think of a hospital that stores the patients' historical processes, and which needs to satisfy strict rules on privacy of the data. For obvious reasons, these organizations cannot relegate conformance checking to a manual task, due to the following features, that resemble the properties that define *Big Data*:

- *Volume*: The size of industrial data representing a set of processes may be huge. For instance, retailers such as Walmart store more than a million customer transactions per hour.
- *Variety*: Event logs often are obtained from several, heterogeneous data sources. This includes databases, sensor data, or phone conversations.
- *Velocity*: Not only process-oriented data may become available at high rates, but also the actions to take for detected deviations typically need to be fast. One cannot rely on the manual analysis of large volumes of event data, while striving for fast response times.

- *Veracity*: Traces in an event log may contain errors that potentially affect conformance checking. For instance, event logs may contain errors due to manual input by process stakeholders.

Conformance checking techniques need to keep up with the trends towards massive availability of event data. Therefore, apart from describing the core algorithmic approaches for conformance checking, an emphazis is put on distributed and online techniques in the last chapters of this book.

Furthermore, novel models for the conduct of process in organizations emerge as drivers of conformance checking. For instance, areas like *collaborative business process management* acknowledge the paramount importance of conformance checking to support the execution of collaborative processes [130].

1.4 Spectrum of Conformance Checking

As detailed above, conformance checking relates event logs and process models, i.e., different representations of a process, to each other. This general setting is illustrated in Figure 1.4, which also outlines a whole spectrum of techniques that can be summarized under the umbrella of conformance checking.

Starting with a broad range of application domains —from finance through logistics to healthcare— conformance checking aims at understanding and improvement of a system, whose behaviour is represented by a process, based on recorded and

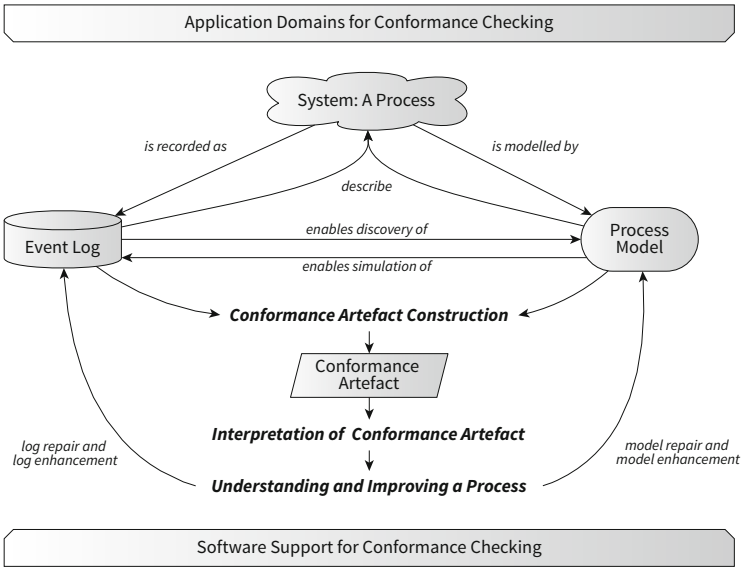


Figure 1.4 Illustration of the spectrum of conformance checking

modelled behaviour. The respective representations, logs and models, do not only describe the process, but are also linked by two complementary sets of techniques:

Discovery aims at the construction of a model of a process based on its recorded behaviour. Common discovery techniques strive for *fitness* and *precision*, i.e., the resulting model shall capture the recorded behaviour, but be limited in how far additional behaviour is allowed. The latter is of importance since an event log per se contains exemplary behaviour of the process, so that some generalization of the recorded behaviour is desirable. We later show how conformance checking enables the quantification of fitness and precision measures that are used to guide and evaluate discovery techniques.

Simulation refers to the generation of an event log from a given model of a process, thereby exemplifying the behaviour that could be observed for the process. An event log that is constructed by simulation techniques is subject to the evaluation dimensions mentioned above for discovery techniques: Simulation that introduces noisy events compromises the fitness of the resulting event log. Completeness of the simulation in terms of coverage of the behaviour of the process model, in turn, translates into the precision of the model with respect to the event log.

Given an event log and a process model, conformance checking techniques yield some explicit description of their consistent and deviating parts, here referred to as a conformance artefact. Examples of such artefacts include information on:

- Behavioural rules such as ordering constraints for activities imposed by the model that are violated by some traces of the event log;
- Events of traces that could correctly be replayed by task executions in the process model, or for which the replay failed;
- An alignment between the events of a trace of the event log and the task executions of an execution sequence of the model.

A conformance artefact then enables conclusions on the relation between the event log and the process model. By interpreting the conformance artefact, for instance, the fitness and precision of the model regarding the given log is quantified. Such an interpretation may further involve decisions on how to weight and how to attribute any encountered deviation. Since the log and the model are solely representations of the process, both of them may differ in how they abstract the process.

Differences in the representations of a process may, of course, be due to inaccuracies. For example, an event log may be recorded by an erroneous logging mechanism, whereas a process model may be outdated. Yet, differences may also be due to different purposes and constraints that guide how the process is abstracted and therefore originate from the pragmatics of the respective representation of the process. Think of a logging mechanism that does not track the execution of a specific activity due to privacy considerations or a model that outlines only the main flow of the process to clarify its high-level phases. Either way, the respective representations

are not *wrong*, but differ because of their purpose and the constraints under which they have been derived.

By linking an event log and a process model through a conformance artefact, the understanding of the underlying process can be improved. That includes techniques for performance analysis. For instance, traces of an event log can be replayed in the process model, while taking into account the deviations between the log and model as materialized in the conformance artefact. Another example includes the inspection of the conditions that govern the decision points in a process. The conformance artefact can be used to derive a classification problem per decision point, which enables discovery of the respective branching conditions. Assuming that the model represents the desired behaviour of the process, the conformance artefact further enables conclusions on how the current realization of the process needs to be adapted.

Improvement of a process based on the result of conformance analysis may also involve changes of the representations of a process. In particular, if deviations between an event log and a process model are considered to stem from inaccuracies and errors in the representation of the process, they may be adapted to achieve conformance. Depending on which representation is considered to be trustworthy, either the log or the model serve as a reference point for the respective changes. *Log repair* considers deviations between the log and model to originate from errors in the log. Therefore, the event log is adapted to make it consistent with the model, thereby realizing some cleansing of the recorded behaviour. *Model repair* represents the mirrored case, where the deviations are deemed to stem from the model. That is, the model is adapted to achieve consistency of the process representations. It should be noted, though, that in many application scenarios, a balanced approach may be needed. Some deviations may indeed be attributed to errors in the log, calling for log repair, whereas others may stem from errors in the model, calling for model repair.

Beyond resolving inconsistency by repairing the representations of a process, the log or the model, improvement may relate to the extension of the representations based on the conformance artefact. *Log enhancement* enriches the events of a log with additional information, which is then exploited by further techniques for an log-driven analysis of the respective process. The added information may stem directly from a process model (e.g., annotations of activities with responsible roles) or some analysis conducted based on a process model (e.g., the probability of ending in a particular state). By establishing a link between an event log and a process model, a conformance artefact guides how such information propagates from one representation to the other one. *Model enhancement*, in turn, refers to the enrichment of the model based on the log, through the conformance artefact. Again, this enhancement enables further types of analysis based on the model. A common example would be the enrichment of a process model with durations of activity execution. To this end, a distribution is fitted to the execution times recorded in the log per activity. These distributions enhance the process model and, for instance, enable performance simulation and prediction.

All of the above techniques are commonly supported by software tools. In order to scale conformance checking to Big Data settings and the complexity of real-world processes, automated conformance checking is of crucial importance. In this book, we therefore include *In the Lab* sections that refer to tasks that can be performed with publicly available conformance checking tools.

1.5 Relevance of Conformance Checking

Conformance checking turned out to be beneficial in a variety of domains. As has been described in the sections before, the two necessary inputs for a conformance checking project are: on the one hand, an event log, a set of recorded traces of a process; and on the other hand, a process model that formally describes the underlying process. Whether this process model was obtained through automated techniques (e.g., process discovery) or through manual modelling efforts is irrelevant for conformance checking. However, it is essential to know the type of model dealt with, e.g., whether it is an as-is or to-be model, in order to interpret the results obtained through conformance checking.

Through conformance checking various questions can be answered, such as:

- Is the process being executed as it is documented in a model?
- Is the model of a process still up-to-date?
- Have there been violations of rules in the recorded instances of a process?
- How much flexibility is allowed for in the execution of a process?
- Is the flexibility offered by a process model actually used during the execution of a process?

In a particular context, the interpretation of the answers to all these questions depends enormously on the type of model considered. For instance, deviations detected for an as-is process are only a sign of low quality of the process model at hand; in contrast, deviations detected for a to-be model may require a careful inspection, often disclosing real problems in the way the process is operated (see for instance the examples illustrated in Section 1.3).

In the remainder of this section, we report on some application domains of conformance checking, to illustrate its great value in helping organizations to exploit event data for analysis and improvement of their operations.

Healthcare In the context of healthcare, several process-oriented initiatives have appeared in the last years where conformance checking plays an important role. The reader can find a summary in a recent publication [67], focused on the wider area of process mining. Processes in the healthcare domain can be of several types, e.g., *the admission process*, *the surgery process*, *the treatment process*, etc. A healthcare information system (HIS) of a hospital may contain hundreds of these processes, and terabytes of information corresponding to their executions. Deviations corresponding to the processes involved in a hospital can reveal important

inefficiencies or errors, that may have drastic consequences. For instance, prior to a surgery for treating a serious disease, all necessary tests regarding the patient resistance and suitability should have been performed. Likewise, processes that regard a treatment should be performed without excessive delays so that important causally related actions are not intertwined at will. Finally, a hospital is a special case of an organization where administrative processes need to be safely executed to guarantee fairness in the way patients are managed. All these requirements have, at their core, the necessity to use conformance checking techniques, so that process models and the reality agree.

Finance Turning to a different domain, the finance sector has seen large interest in conformance checking initiatives in recent years. Again, there is a plethora of processes conducted in this domain, reaching from the aforementioned *loan application process*, through *transaction processing*, to *risk management* and *audit processes*. A driver for this trend are the comprehensive legal frameworks that apply in the finance sector. For instance, the Basel Accords regulate the banking industry in terms of risk-weighting of their assets, minimum capital requirements, and liquidity requirements. Another example are anti-money laundering directives enforced in the European Union or the US. They feature requirements on the identification and verification of a client's identity (aka *KYC, Know Your Customer*). These legal frameworks impose requirements on the conduct of processes, which, at least partially, can be translated into models of the normative behaviour. Since execution of processes in the finance sector is heavily supported by information systems, conformance to these normative models can be assessed by means of conformance checking. For instance, when opening a correspondent (nostro or vostro) account to act on behalf of another financial institution, the respective process shall satisfy various compliance requirements, such as ensuring an evaluation and black-listing institutions upon a negative evaluation result; conducting a due diligence study if this is the first bilateral agreement between the institutions; and ensuring that no account is opened if the study fails or the provided bank certificates are not valid. In the past, the assessment of conforming process execution was primarily done manually, by investigating a small sample of the respective process instances in a labour-intensive manner. Automated conformance checking based on event logs, in turn, promises to automate this procedure. As a consequence, audits and compliance assessments become not only more efficient, but also more effective as the analysis is no longer limited to a small fraction of the process instances.

Transportation Well-coordinated *logistic processes* are a key factor to successful supply chain management. Optimization of material flow and effective management of inventories relies on well-orchestrated logistic processes. This is challenging since such processes may span multiple countries (leading to import/export approval and customs declaration) and diverse transportation modalities, and come with complex service-level agreements and, therefore, risk and penalty structures. Consequently, to enable effective management of various interlinked logistics processes typically relies on conceptual models. These models are the basis of simulation or analytical approaches, e.g., when minimizing the ripple effects incurred by

transportation delays due to missed connections. In this context, conformance checking helps to establish a link between these models and the recorded execution of logistics processes. An assessment of conformance relies on event data from information systems such as Supply Chain Management (SCM) systems, but may also integrate context information, e.g., on delays at customs clearance centres. Conformance checking thereby helps to obtain realistic models of logistics processes and increases the trustworthiness of decision supports that are grounded in these models.

Manufacturing In manufacturing, processes, often referred to as *production workflows*, coordinate the activities needed to build and assemble a product, which may be as simple as a bike lock or as complex as a full-fledged car. Many such workflows are partially automated in the sense that some of the activities involve human intervention. Yet, as part of digitalisation initiatives, even such manual fitting or assembly activities leave digital traces. For instance, sensor-based pick systems record how many parts of a particular type have been retrieved for an activity, while dynamometric screwdrivers record the actual force applied to a screw. Again, the conduct of these workflows is specified by models in order to optimize staff planning and resource utilisation. Against this context, the benefit of conformance checking is twofold. On the one hand, it helps to detect deviations in the recorded behaviour from the specified workflow. This information can be used locally (e.g., notifying a worker to adapt the handling of an individual process instance) as well as globally (e.g., identifying hot-spots of conformance issues and reacting by redesigning the workflow environment or providing targeted worker training). On the other hand, it also helps to increase the trust in the plan and predictions derived from the workflow models.

Bibliographic Notes

The field of conformance checking is relatively new. The definition of the area and a proposal of initial algorithms was presented in the scope of Anne Rozinat's PhD thesis at the TU/e [93] and corresponding publications [95, 97, 116, 117]. Important notions arise from this work, like *fitness* or *appropriateness* between a process model and log. Also, important algorithms, including the techniques to evaluate fitness based on the replay of the traces and the missing/remaining/produced/consumed tokens. Moreover, structural/behavioural techniques to evaluate appropriateness are proposed, that represent the first step towards evaluating precision of a process model with respect to a log.

Also in the scope of the TU/e, the seminal work under the PhD thesis of Arya Adriansyah [1] is crucial for formalizing the notion of *alignments*, one of the main concepts explored in this book. It provides an algorithm for the computation of alignments between an event log and a process model, and it shows preliminary techniques to be used on top of alignments so that further information can be extracted or visualized. Several applications of alignments are explored in the related publications, like performance analysis [2, 115], high-level deviations [7], privacy analysis of user behaviour [8], and alignment-based precision metrics [4].

Another work that has been important for conformance checking is the log conformance analysis presented in the scope of Matthias Weidlich's PhD thesis [131]. The thesis introduces the concept of *behavioural profiles*, as a tailored abstraction for processes that allows comparing recorded and modelled behaviour.

Finally, several books have considered to some extent the area of conformance checking. We include a detailed comparison of these books with the current book. Overall, we believe that the current book, being the first in-depth monograph on conformance checking, complements the books listed below.

The book by Wil van der Aalst [114] is a general book that introduces the field of process mining. It describes the three main dimensions of process mining: discovery, conformance and extension. In the second edition, conformance checking is described more thoroughly with respect to the first edition of the book. In particular, the notion of alignment is now briefly introduced in the book, and a clear emphazis on its importance for the whole field of process mining is acknowledged. In our book, where conformance checking is positioned as a discipline on its own, we aim to complement the book by Van der Aalst so that techniques for the alignment of process models and recorded behaviour can be learned and applied in practice. Due to having a narrower focus, the current book deepens into algorithmic perspectives and multidimensional views, and provides an up-to-date take on applications and software support for conformance checking.

The PhD thesis by Jorge Munoz-Gama (published as a book by Springer LNBIP [71]) contains two main contributions: on the one hand, a technique is proposed to estimate the precision of process models with respect to recorded behaviour; on the other hand, methods are presented to distribute the decisional problem of assessing the fitness of a process model with respect to recorded behaviour. We describe these techniques in this book.

The book by Beheshti et al. [13] gives an overview of basic techniques for process analysis, covering a broad spectrum from methods for process model matching and querying, to infrastructures for data analysis in a process context. The book mentions process mining as a related area, but focuses on the broader field of data-driven techniques for process analysis.

1.6 Exercises

1.A) Processes in your daily life

Give examples of processes that occur in your daily life.

1.B) Deviations

For the processes described before, provide examples of deviations.

1.C) Cases and traces and execution sequences

Explain the difference between a case, a trace and an execution sequence.

1.D) As-is vs. to-be

What differentiates an as-is model from a to-be model?

Chapter 2
The Basics of Processes and Models

Processes play an important role in everyday life and structure organizations in various domains. From the yearly filing of taxes to the daily routine of paying your bills, our lives are governed by (administrative) processes. At the same time, domains such as finance, logistics, and healthcare are structured around processes. In any case, processes form a complex system in which actors, like ourselves, perform *activities* in order to achieve certain goals. Commonly, this is done within a *case* that comprises the *executions of activities* for a specific trigger, such as the due date for tax returns or the receipt of a bill to pay.

When describing a process, a process model provides an abstraction, capturing some of the process' activities by means of *tasks*. A specific instance of a process, i.e., a case, then corresponds to an *execution sequence*, a sequence of *task executions*.

Whether and when an activity is executed is often recorded by information systems. For instance, when filing your taxes, this is recorded by the tax office, whereas payment of a bill is recorded by both the bank and the receiving party. Such recordings of activity executions are called *events*. Events that capture progress of a single case are grouped together in a *trace*, while a collection of such traces is referred to as an *event log*.

Against this background, we note that both process models and event logs represent different conceptualisations of processes. In this chapter, we therefore review in Section 2.1 essential building blocks of a process. Their representation in process models is described in Section 2.2. We then turn to the notion of an event log in Section 2.3. In Section 2.4 we discuss *languages* as the common ground of different process representations.

J. Carmona et al., *Conformance Checking*,
https://doi.org/10.1007/978-3-319-99414-7_2

2.1 Building Blocks of Processes

Next, we cover basic building blocks of processes, commonly known as *workflow patterns*, and discuss how they can be captured in process models.

2.1.1 Tasks

When describing a process and the activities involved in it, we need to start with the basic element: one activity. One activity is a logical unit of action, a piece of work, that we do not want to further split (see Excursion 2 below). In a process, we represent activities with a placeholder in the model. We call this placeholder *task* and draw it in a BPMN model with a rounded rectangle. We also assign a textual label to the task that reflects the activity that is being done at the point when the task is activated. Typically, the convention of labelling is "Verb object" (e.g. "Accept application", "Receive offer").

Excursion 2

Avoid getting lost in details: abstraction as a modelling principle

How much detail we reflect in a process determines the *granularity* of the activities to consider and, thus, the tasks in a model of the process. In general, it is always possible to split an activity into a number of pieces of work. Instead of saying "Finalise application" in our running example, we could zoom into this activity and find sub-activities. For example, we can replace "Finalise application" with: "Select application", "Open application", "Proof-read application", "Fill in necessary categories", "Assign responsibilities", "Save application", and "Close application". And even that would still be an abstraction of what really happens, as we can further zoom into, e.g., "Select application" and see that this activity consists of: "Read the contents in the inbox", "Decide which item to select", "Choose how to select it" (e.g., mouse or keyboard), "Select the item with the chosen means". In most cases, such fine-grained activities are of interest neither to those who created a process model, nor to those who intend to improve a process. Keeping a reasonable abstraction level when referring to a process allows us to choose suitable means of achieving improvement goals. To sum up, activities in processes and thus tasks in process models should be on a reasonable abstraction layer to maintain economic feasibility of both modelling and analysis.

It is important that we are aware of the difference between an activity and one execution of the activity (*activity execution*). While an activity refers to the general concept, an activity execution reflects one particular instantiation of the activity

inside a particular case, an instance of the process. In fact, the same activity can be executed multiple times in one case and it can also be represented in different contexts in the process as different tasks that have the same label. For example, we have two tasks in our running example that refer to the activity "Decline application" in Figure 1.3.

Excursion 3

BPMN—The business process model and notation

The modelling notation we use in our examples is the Business Process Model and Notation (BPMN). BPMN is a feature rich language to describe processes in great detail. It can be considered the de-facto standard for business process modelling. This is not a book about BPMN, so we refrain from an in-depth introduction to the language. A step-by step introduction to the concepts of BPMN can be found for example, in the book "The Process: Business Process Modelling using BPMN" by Grosskopf et al. [51]. More technical details can be found in the specification of the language by the Object Management Group.[1]

[1] http://www.omg.org/spec/BPMN/

2.1.2 Sequence Pattern

When we perform activities to reach a certain goal in a process, we often cannot start an activity before finishing another one. Sometimes, the reason for this phenomenon may simply be that an object needs to be received or created before it can be processed. To illustrate such causal dependencies between activities, let us consider how a sequential process of our running example would look like. Figure 2.1 shows an optimistic version of the loan application. In this model, after the application has been submitted (*As*), it is accepted (*Aa*). After that, an offer is selected and sent (*Sso*) and then received (*Ro*). Consequently, the application is finalised (*Fa*) and accepted (*Ao*). Finally, the application is approved and activated (*Aaa*), leading to it being finished (*Af*). All activities represented by these tasks are performed in sequence, and the arrows indicate the direction of the process control flow. Following the control flow, we can read the possible execution sequences of a process model. Each of these sequences describes a potential case of the process. We can imagine that such an execution sequence is derived by handing over a token from task to task along the control flow.

In imperative modelling languages, where the model tells us what we should do next, the sequence pattern forms a strong relation between the connected tasks. It limits the execution of the captured process to only execute the activities in the sequential order specified for the respective tasks. The model in Figure 2.1 permits

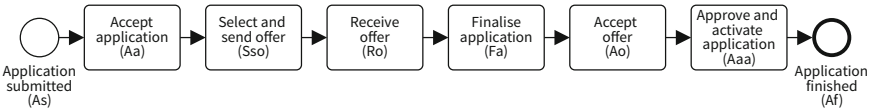


Figure 2.1 Sequential model for a simplified version of the loan application process

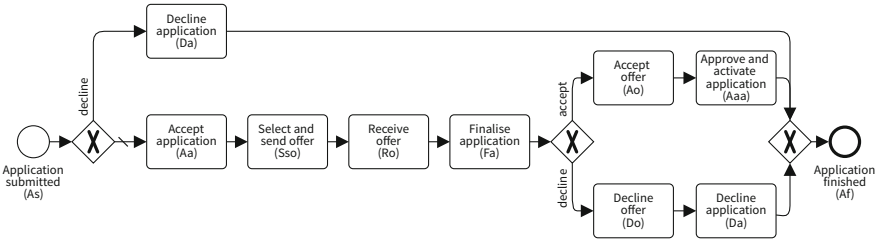


Figure 2.2 Model for a simplified version of the loan application process containing two exclusive choices and one merge

therefore only a single execution sequence to reach the goal of having finished the application (*Af*). That is, the activities of the process captured by this model can only be executed in the depicted sequential order.

2.1.3 Exclusive Choice

The previous example model did not leave us much choice in the execution of the process. In fact, a financial institution implementing the sequential model in Figure 2.1 would end up accepting every loan application—no matter how unrealistic the application is!

To represent a choice in a process, we can use the exclusive choice pattern. Two modelling constructs help to design models that contain choices: i) the *exclusive or (XOR) split* that indicates mutually exclusive alternatives of continuation at a point, and ii) the *XOR join* that merges alternative branches in the model to continue from there onwards on a shared path. Figure 2.2 shows an extended version of our running example, where we have a choice how to continue immediately after the application has been submitted (*As*). The exclusive choice is represented in the figure as a gateway (a diamond shape) marked with an "X" (for exclusive). It allows us to continue with accepting the application (*Aa*), which is the default choice, or declining the application (*Da*). When the application is accepted in the beginning, we see that the model at a later point allows us to further decide whether to accept the offer (*Ao*) or decline it (*Do*) and continue accordingly. Finally, the process is merged with the XOR join gateway that shows that all three exclusive branches in the model lead to a finished application (*Af*).

2.1.4 Parallel Execution

Sometimes we have activities in a process that all need to be done, but there are no dependencies between them. This means that the activities can be done in parallel in any order. In our example, a manager might see that the application can already be finalised (*Fa*) independently from the offer that is being selected and sent (*Sso*) and later received (*Ro*). With this observation, the model in Figure 2.1 can be redesigned to allow this additional flexibility in the process. A simple process model with this parallel execution pattern is depicted in Figure 2.3. The parallel execution starts at the parallel (AND) gateway that models this particular relation and is denoted with the "+" sign. It shows that all outgoing branches are executed independently from one another. We also call this gateway a *parallel split* or *AND split*. Inside the branches that leave the AND split, we are still allowed to rely on other workflow patterns. In the model in Figure 2.3 we see that select and send offer (*Sso*) is in a sequence with receive offer (*Ro*).

When execution of parallel branches comes to an end, and the next activity in the process sequentially depends on these branches, we need to wait for all these branches to finish before we can continue. To capture this synchronization in a process model, we use again the AND gateway. At the AND join gateway, we have multiple incoming branches and one outgoing branch. The AND join ensures that the process can only continue once all incoming branches are finished.

2.1.5 Loop

In our running example process of the loan application, it can happen that a customer is not satisfied with an offer and requests a change. In this situation, another offer is selected and sent. When the offer is received, there might still be open questions left, resulting in another change requested. This leads to a second cancellation of the offer and another selection and sending of the offer. When we realize that we cannot

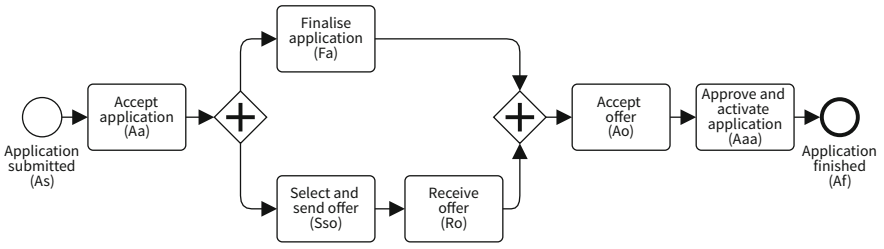


Figure 2.3 Model for a simplified version of the loan application process containing two parallel branches between accept application and accept offer

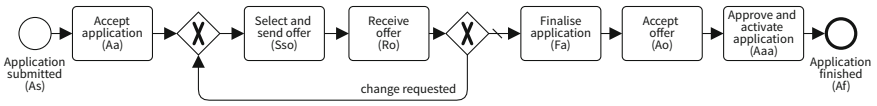


Figure 2.4 Model for a simplified version of the loan application process that allows us to repeatedly send and receive offers upon change requests

anticipate how often certain activities are repeated in a process, we rely on the *loop* pattern (sometimes also *iteration*, or *cycle*).

The loop pattern in our model can be realized through a combination of the exclusive gateway. In contrast to the normal exclusive pattern, we can decide to go back to an *earlier* point in the process to indicate that we want to repeat the activities represented by the respective tasks from that point forward. This way, the control flow forms a loop that can be traversed an *arbitrary number* of times (iterations). Each time we reach the same decision point, we can decide whether to go back and perform another iteration, or we can leave the loop and continue the process. Figure 2.4 shows the model that allows us to repeat the tasks *Sso* and *Ro*. Note that the loop begins with a XOR join that merges the current control flow and a branch that is potentially visited in the future.

Excursion 4

Events and activities

In this book, we are mostly interested in events that are recorded in event logs. These recorded events mostly represent progress in a process in terms of activity executions. The term event is ambiguous, however. We can also distinguish between events and activities in a process and also distinctly capture them in a process model. Activities are the building blocks of a process and constitute *active* behaviour in a process, where customer value is generated and certain manual or (semi-)automated actions are performed. Things happening beyond the control of an organization represent *passive* behaviour that can be captured as events. Following the BPMN standard, such events are represented explicitly as circles, in contrast to the rounded rectangles of tasks that denote activities.

Typically, events in a process model do not take time, but are instantaneous (e.g., *"Customer arrived"*, *"Form received"*) and signal that the process can continue after their occurrence. The activities represented by tasks are performed by resources and their execution typically takes time and incurs a certain cost.

2.2 Process Models

We can compose more complex process models by hierarchically nesting models that capture workflow patterns, i.e., the basic process building blocks as discussed above. Using these patterns and hierarchical composition, we can describe a large share of the actual processes that are performed in organizations. Yet, to also capture more exotic behaviour, we can abandon the clear structure of these blocks and consider processes that are built of an arbitrary control flow between activities. Such processes are represented by unstructured models.

2.2.1 Structured Models

Processes built of basic workflow patterns can be described with a process model by simply nesting their respective representations. In fact, our running example in Figure 1.3 contains all introduced patterns. It comprises the sequence, choice, parallel, and loop patterns, which are highlighted in Figure 2.5. We see that through various arrangements of these patterns, we are able to express a large diversity of behaviour.

The models we have looked at so far had a certain *structure*. That is, there are always pairs of the same gateway type that encapsulate the branches of the respective process. For example, all branches between an XOR split and an XOR join are exclusive to one another. Furthermore, these blocks are entered at one single point and left at another single point, that is, before and after there is only one branch going in and out. These blocks are also called single-entry single-exit (SESE) regions. If a model consists only of SESE regions, we call the model block-structured (or well-structured). Our example process model in Figure 2.5 is block-structured and the blocks are nested and form a hierarchy. Models that do not have this property are called unstructured.

2.2.2 Unstructured Models

An example unstructured model is depicted in Figure 2.6. Here, it is possible to exit the loop at two positions: (1) when no change is requested, one continues with the activity to finalise application, and (2) there is a new option that becomes available after a timeout of 1 week, if no response to the offer came from the applicant. The new gateway type with two circles enclosing a pentagon is a so called event-based gateway that indicates that external events dictate the continuation of the process. The two events that can happen here, are a timer event (indicated by the little clock symbol) and the receive task (envelope symbol in corner). Simply put, the process waits for these two events and continues with whichever arrives first. In this regard,

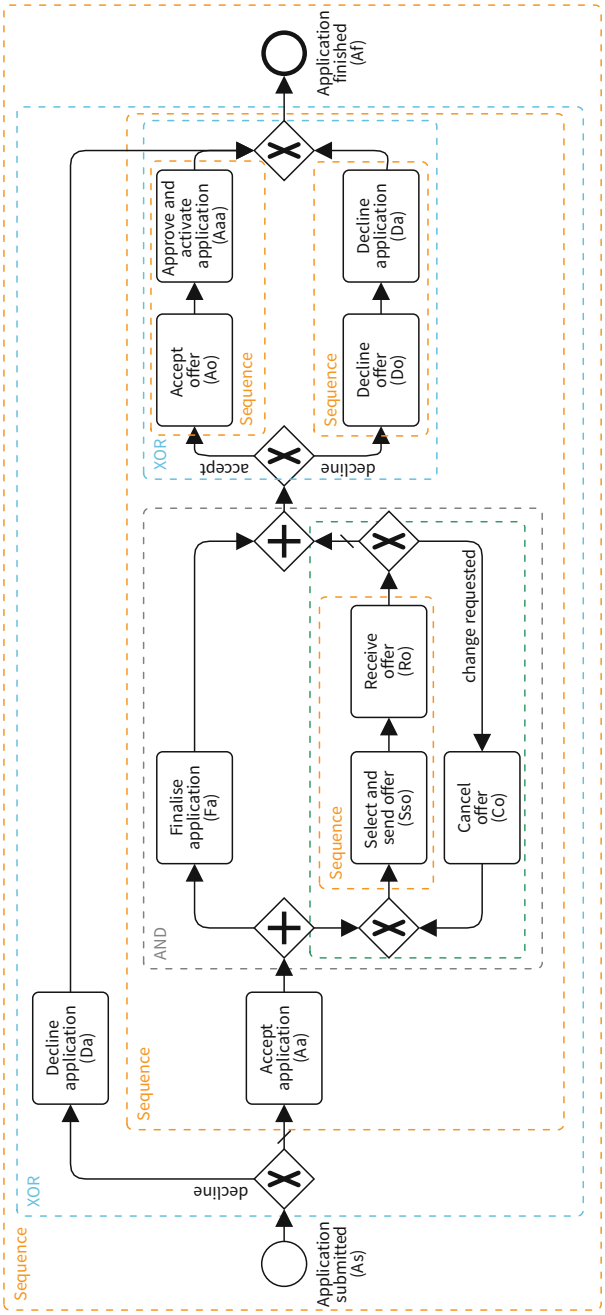


Figure 2.5 Example process model of a loan application process in BPMN with annotated building blocks; cf. Figure 1.3

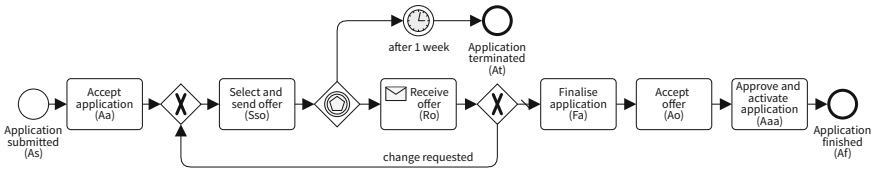


Figure 2.6 Unstructured model for a simplified version of the loan application process that allows us to repeatedly send and receive offers upon change requests

this gateway is to be treated as an XOR split as far as the control flow is concerned. The point is that the loop, which was previously a SESE region (see Figure 2.4), is now no longer a SESE loop. It has one entry and *two* exit points, rendering this model unstructured.

2.2.3 Semantics of Process Models

The execution dependencies imposed by a process for the activities induce a set of states and state transitions, jointly referred to as the *state space* of the process. Intuitively, a state of a process can be thought of a specific situation during the execution of a process. It is reached via the execution of activities starting in a well-defined initial state and induces activities that may further be executed to advance to a next state.

The notion of a state space carries over directly from processes to process models. That is, state also represents a specific situation reached during the execution of tasks of a process model. Inspired by the formalism of Petri nets that are introduced in the second part of this book, procedural process modelling languages such as BPMN describe the execution semantics of their language constructs as a token-flow game. A state of a BPMN model is then a distribution of tokens over the control flow arcs of the model.

For illustration, we assume that all tasks in a process model have exactly one incoming and one outgoing control flow arc, which is also considered to be good modelling practice. Then, a task is enabled in a state, i.e., can be executed in a particular state, if its incoming control flow arc is assigned a token by the respective distribution. If it executes, this token is *consumed*, i.e., no longer assigned to the arc. Moreover, a token is *produced* on the outgoing control flow arc of the task, i.e., the number of tokens of this arc in the current state is increased by one. Following this line, execution semantics of language constructs other than tasks are defined in the same vein: An exclusive gateway is enabled in a state if at least one of its incoming arcs is assigned a token and a token is produced on solely one of its outgoing arcs (as determined by the branching conditions of these arcs). A parallel gateway, in turn, is enabled only once all incoming arcs carry at least one token and its execution produces a token on all outgoing arcs.

Using the above notions, a model induces a set of possible sequences of task executions, also referred to as *execution sequences* of the model. They are implied by the sequences of state transitions that are reachable from a given *initial state*, i.e., an initial distribution of tokens to arcs. In a BPMN model, however, this initial state is commonly derived from explicit start events that do not have any incoming control flow arcs and are denoted by a circle with a regular border. Upon enabling, typically through an external trigger such as a received message or the expiration of a timer, such a start event produces a token on its outgoing control flow arc, thereby starting an execution sequence that represents a new case of the process. Then, the behaviour of this case is represented by the token *flow* through the model, where each execution of a task represents a state transition. In addition, a process model may define a *final state*, i.e., a distribution of tokens to arcs that represents a valid end of an execution sequence and, thus, a case of a process. In BPMN, again, such final states are modelled explicitly by means of end events that have only an incoming control flow arc, but no outgoing control flow arc, and are denoted by a circle with a bold border. An end event that is enabled consumes a token from its incoming arc upon execution. Consequently, the final state of a process model in BPMN is assumed to be the empty state that is reached after all tokens have been consumed by some end event.

Excursion 5

Let's get the process started (and ended)

As discussed in Excursion 4, we are primarily concerned with events that represent the recorded executions of activities of a process. However, modelling languages such as BPMN (see Excursion 3) also include event concepts to distinguish how activities and instantaneous signals that impact a process are captured. This is of particular relevance for the start and end of a process.

Tasks that have no incoming control flow arc can be seen as being enabled immediately, so that their execution indicates the start of an execution sequence representing a new case of the process. However, it is typically better to model the trigger that instantiates a process explicitly. For this purpose, BPMN defines various types of start events, some of which are depicted in Figure 2.7

Figure 2.7 Different types of start events as defined by BPMN

Here, a plain start event represents models for which the trigger to instantiate a new case is not defined, which may correspond to manual

(continued)

instantiation by a user. The other events, in turn, define explicitly that a case is started upon receiving a message, observing the expiration of a timer, satisfying specific data conditions, or being notified about an error (e.g., thrown by a different process). Note that instantiation semantics quickly become complex and potentially ill-defined when using multiple start events within a single process model.

Similarly, BPMN defines different types of end events shown in Figure 2.8. While the plain end event only consumes a token from the incoming control flow arc upon execution, end events may involve further actions. For instance, a message may be sent or an error may be thrown. A terminate end event has even non-local semantics: upon execution it consumes *all* tokens from *all* arcs in the process model, thereby modelling an immediate and complete termination of the case, e.g., due to an error that precludes any recovery.

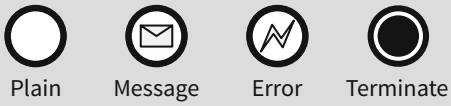


Figure 2.8 Different types of end events as defined by BPMN

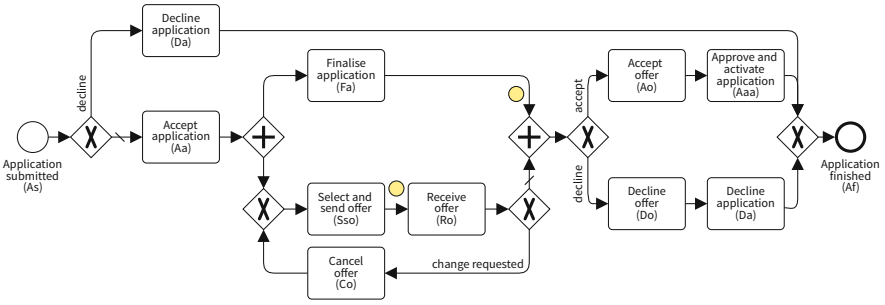


Figure 2.9 Loan application process model with two tokens that represent the state reached after executing the sequence of activities $\langle As, Aa, Sso, Fa \rangle$

As an example, Figure 2.9 illustrates a specific state reached during the execution of the aforementioned loan application process. Once an application has been submitted (As) and accepted (Aa), the offer has been selected and sent (Sso), and the application has been finalised, the process is in a state that is represented by two tokens assigned to the arcs originating from tasks Sso and Fa, respectively. Note that the same state would have been reached, if for a submitted (As) and accepted application (Aa), the application would have been first finalised (Fa) before the order

had been selected and sent (*Sso*). The reason being that tasks *Sso* and *Fa* can be executed in parallel.

In the state visualized in Figure 2.9, only the task to receive an offer (*Ro*) can be executed. Once this is done, the token from the incoming arc of this task is consumed and a token is produced on its outgoing arc. Then, the decision on whether a change is requested is taken. If so, the offer is cancelled (*Co*). If not, the AND gateway has both of its incoming arcs assigned a token, which are consumed subsequently to synchronise the parallel branches of execution. Figure 2.10 illustrates these two possible execution alternatives.

2.3 Event Logs

In the previous section, we introduced *activities* as the primary element of processes and tasks as modelling constructs to represent activities that should be executed. When processes then are executed in an organization, these executions are typically registered by means of information systems that support the conduct of the process. To stay with the loan application process, the execution of the task referring to the "Accept application" activity is recorded in a database in which all information relevant for the application is stored, as well as data about who (i.e. which employee) accepted the application and when this happened.

The recording of a single execution of an activity is called an *event*. An event refers to an *atomic* execution of a specific activity. And typically, events are performed in a certain *context*, such as for example a specific loan application. This context is commonly given by the *case* as part of which an activity was executed. The notion of a case, therefore, binds together events, thereby allowing us to track the evolution of a case over time. The events related to a single case are called a *trace*. Very much as each activity execution is part of a case, each event representing a recorded activity execution is part of a trace.

The notion of a trace is fundamental for *event logs*. In essence, an event log is a collection of traces, each trace comprising events that can be sorted by their occurrence time. However, event logs can also be represented as a collection of events, where all events carry information about the respective trace as part of their payload. Consider for example our loan application process of Figure 1.3. Table 2.1 shows a number of events that correspond to a single loan application. The application with id *A*5634 is accepted by the system on January 1st at 12:32 and the customer asks for a €2000 loan. On January 3rd the application is finalised and two days later, an offer is made to the customer for a €1500 loan. The offer is received back on January 10th and the customers did not sign it, nor did they indicate they want any changes. Therefore, a few minutes later, the offer is declined, which is also done for the application as a whole.

This simple example shows the richness of data available in a single application. Now imagine the data available for the entire process, i.e., data about multiple

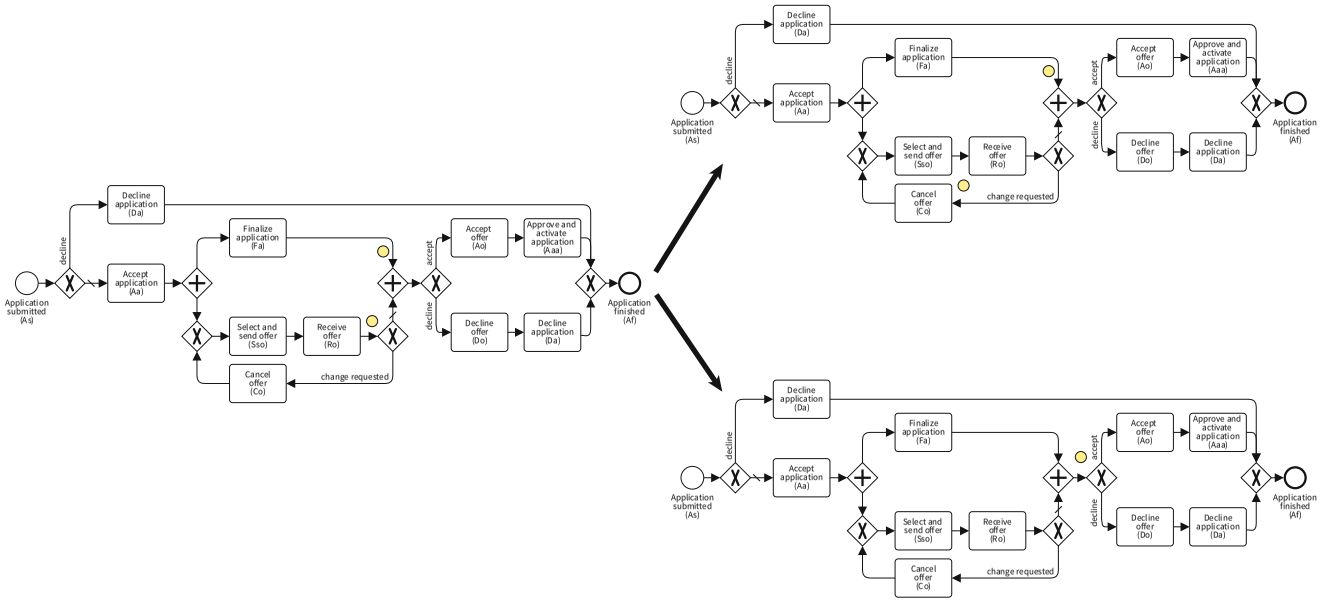


Figure 2.10 Loan application process model with the state reached after executing the sequence of task $\langle As, Aa, Sso, Fa, Ro \rangle$ (left), and the two possible next states

Table 2.1 Example of a trace in the loan application process

Event	Application	Offer	Activity	Amount	Signed	Timestamp
e_{13}	$A5634$		Application submitted	€2000		Jan 01, 12:31
e_{14}	$A5634$		Accept application	€2000		Jan 01, 12:32
e_{22}	$A5634$		Finalise application			Jan 03, 09:00
e_{37}	$A5634$	$O3541$	Select and send offer	€1500		Jan 05, 12:32
e_{42}	$A5634$	$O3541$	Receive offer		NO	Jan 10, 10:00
e_{54}	$A5634$	$O3541$	Decline offer			Jan 10, 10:04
e_{64}	$A5634$		Decline application			Jan 10, 10:05
e_{65}	$A5634$		Application finished			Jan 10, 10:06

applications, offers, etc. Such an event log would grow considerably and Table 2.2 shows an excerpt of such a log.

Table 2.2 shows what is commonly called an event log. Each row in this table is an event and each event refers to a trace (in this example the loan application) and to an activity. Furthermore, events are assumed to be totally ordered (in this case by the timestamp) and there may be other data relevant to the process, such as the amount requested, the offer ID, the amount offered and whether the offer was signed or not.

In practice, one can look at these event logs from various perspectives. So far, we considered the application to be the leading perspective, or the trace. However, we could also look at the same data and select the offer as the trace identifier. If so, many events would not be included and events that currently belong to a single trace become separate traces. This is shown in Table 2.3, where the offer perspective was chosen for the same data as shown in Table 2.2.

Whenever processes in organizations are executed precisely as captured by a respective process model, any case of a process should be represented by an execution sequence in the model and a trace in the event log. In fact, traditional *workflow systems* are available to control the execution of a process: They take a process model as input and do not allow deviations from it during the execution, meaning that the modelled behaviour is always in line with the recorded behaviour. However, in many cases, organizations do not strictly enforce their processes to be executed according to models. Instead, they enable a certain degree of flexibility, so that there may be deviations between the modelled and recorded behaviour in general, and execution sequences and traces in particular.

Consider again the loan application process of Figure 2.6 which specifies that, if an offer is not received back within one week, the process ends. Now suppose the applicant calls the company after this week and explains that they have been on holiday, but they send back the signed offer as soon as possible. According to the model, this is not allowed. However, in most cases, the company still accepts the signed offer and approves the application. In an event log, the acceptance of the offer would be recorded and it is up to the company to identify these deviations.

In the next section, we establish a common ground for event logs and process models in such a way that such deviations can indeed be identified.

Table 2.2 Example of a log of the loan application process

Event	Application	Offer	Activity	Amount	Signed	Timestamp
...	...	...	...	...	...	...
e_{13}	$A5634$		Application submitted	€2000		Jan 01, 12:31
e_{14}	$A5634$		Accept application	€2000		Jan 01, 12:32
e_{15}	$A5635$		Application submitted	€5000		Jan 02, 04:31
e_{16}	$A5635$		Accept application	€5000		Jan 02, 04:32
e_{17}	$A5636$		Application submitted	€200		Jan 03, 06:59
e_{18}	$A5636$		Accept application	€200		Jan 03, 07:00
...	...	...	...	...	...	...
e_{22}	$A5634$		Finalise application			Jan 03, 09:00
e_{23}	$A5636$		Finalise application			Jan 03, 09:01
e_{24}	$A5635$		Decline application			Jan 03, 09:02
e_{25}	$A5635$		Decline application			Jan 03, 09:03
...	...	...	...	...	...	...
e_{30}	$A5636$	$O3521$	Select and send offer	€500		Jan 04, 16:32
...	...	...	...	...	...	...
e_{37}	$A5634$	$O3541$	Select and send offer	€1500		Jan 05, 12:32
e_{38}	$A5636$	$O3521$	Receive offer		NO	Jan 05, 12:33
e_{38}	$A5636$	$O3521$	Cancel offer			Jan 05, 12:34
e_{39}	$A5636$	$O3542$	Select and send offer	€500		Jan 05, 13:29
e_{40}	$A5636$	$O3542$	Receive offer		YES	Jan 08, 08:33
e_{41}	$A5636$	$O3542$	Accept offer			Jan 08, 16:34
e_{42}	$A5634$	$O3541$	Receive offer		NO	Jan 10, 10:00
...	...	...	...	...	...	...
e_{54}	$A5634$	$O3541$	Decline offer			Jan 10, 10:04
...	...	...	...	...	...	...
e_{64}	$A5634$		Decline application			Jan 10, 10:05
e_{65}	$A5634$		Application finished			Jan 10, 10:06
e_{66}	$A5636$		Approve and activate application			Jan 10, 10:07
e_{67}	$A5636$		Application finished			Jan 10, 10:08
...	...	...	...	...	...	...

Excursion 6

Deviations: Who's right, the model or the log?

Conformance checking aims at relating modelled behaviour and recorded behaviour of a process to each other, whereas reality in the sense of the actual process, the behaviour of the *system* under consideration, is typically unknown. Whenever deviations between a process model and an event log are detected, therefore, the question of the source of this deviation pops up.

(continued)

This question, simply put as *"who is right?"*, is delicate, since the relation between the process and the process model, as well as between the process and the event log, is uncertain. As such, deviations between a model and a log may have two origins, since either representation of the process may not be accurate.

In practice, it is hard to tell whether a certain deviation between a model and a log is due to a modelling error (i.e., the log is right) or anomalies in how the events have been recorded (i.e., the model is right). In the end, this issue can only be resolved by explicitly reflecting on the trust that can be put into either process representation. For instance, a process model created very recently by analysts interviewing process stakeholders may be more trustworthy than a model that has been designed for staff training a few years back. On the other hand, event logs generated by a single enterprise system may be more trustworthy than those derived by fusing data from multiple sources, which may partially be inserted manually by users.

Table 2.3 Example of log of the loan application process, from the perspective of the offers

Event	Application	Offer	Activity	Amount	Signed	Timestamp
...	...	...	...	...	...	...
e_{30}	$O3521$	$A5636$	Select and send offer	€500		Jan 04, 16:32
...	...	...	...	...	...	...
e_{37}	$O3541$	$A5634$	Select and send offer	€1500		Jan 05, 12:32
e_{38}	$O3521$	$A5636$	Receive offer		NO	Jan 05, 12:33
e_{38}	$O3521$	$A5636$	Cancel offer			Jan 05, 12:34
e_{39}	$O3542$	$A5636$	Select and send offer	€500		Jan 05, 13:29
e_{40}	$O3542$	$A5636$	Receive offer		YES	Jan 08, 08:33
e_{41}	$O3542$	$A5636$	Accept offer			Jan 08, 16:34
e_{42}	$O3541$	$A5634$	Receive offer		NO	Jan 10, 10:00
...	...	...	...	...	...	...
e_{54}	$O3541$	$A5634$	Decline offer			Jan 10, 10:04
...	...	...	...	...	...	...

2.4 Languages: The Common Grounding of Process Models and Event Logs

Above, we explained that process models capture the behaviour of a process by means of execution sequences, i.e., sequences of task executions, each representing a potential case of a process. There are various ways to specify the behaviour of a

process model in terms of its execution sequences. In this part of the book, we look into more detail of the *language* of a process model.

The language of a process model is a (possibly infinite) set of possible execution sequences. If a model, for example, contains a loop, then the language of the model is infinite and contains all sequences where the loop is executed once, twice, three times, but also 10,042 times. If a model contains parallel paths, then all interleavings of these parallel paths are part of the language of a model.

Consider the model in Figure 2.3. This model allows for three different sequences of task executions, i.e., its language comprises: $\langle Aa, Fa, Sso, Ro, Ao, Aaa\rangle$, $\langle Aa, Sso, Fa, Ro, Ao, Aaa\rangle$ and $\langle Aa, Sso, Ro, Fa, Ao, Aaa\rangle$. These three sequences together form the language of that model.

To reason about a process model's expressiveness, in this book , we use a compact representation of a language, called *regular expressions*.

The basic concept in a regular expression are atoms, that correspond to one or more letters that form a string. These atoms form words according to a set of rules. In this book, one should consider atoms to correspond to activity executions and words to cases of a model. Based thereon, we use these regular expressions to compactly represent a model.

Given four atoms a, b, c, and d, a regular expression denoting a sequence of these atoms is simply the concatenation of these atoms. For example $a.b.c.d$ is one expression that represents the sequence of a followed by b followed by c, and then d. Another example is $b.a.d.c.c$, where after the sequence of b, a, and d, c occurs twice. Also $a.a$ is a valid sequence that denotes that a is occurring exactly twice, i.e. the execution of a is represented twice.

Turning to process models, atoms denote task executions, while words represent execution sequences. The language of the model shown in Figure 2.1 can be described as the following regular expression: $As.Aa.Sso.Ro.Fa.Ao.Aaa.Af$, i.e. this model has a language consisting of a single execution sequence.

To denote choices, regular expressions use the | operator. Using this operator, we can write the language of the model in Figure 2.2 as the regular expression $As.(Da|(Aa.Sso.Ro.Fa.((Ao.Aaa)|(Do.Da))).Af$ to represent the choices that we have introduced in our running example. Note that we use parentheses to group elements and treat them as one element. This allows us to hierarchically compose regular expressions. The language of this model contains three execution sequences, namely: $\{\langle As, Da, Af\rangle, \langle As, Aa, Sso, Ro, Fa, Ao, Aaa, Af\rangle, \langle As, Aa, Sso, Ro, Fa, Do, Da, Af\rangle\}$.

We denote parallel execution, which leads to all possible interleavings of atoms, with the double pipe || symbol. Referring to Figure 2.3, the language of this model can be written as $As.Aa.((Sso.Ro)||Fa).Ao.Aaa.Af$. Again, this language has three sequences, namely: $\{\langle As, Aa, Sso, Ro, Fa, Ao, Aaa, Af\rangle, \langle As, Aa, Sso, Fa, Ro, Ao, Aaa, Af\rangle, \langle As, Aa, Fa, Sso, Ro, Ao, Aaa, Af\rangle\}$.

Loops are included in regular expressions using the $^+$ and * operators. The language of the model in Figure 2.4 is given as $As.Aa.(Sso.Ro)^+.Fa.Ao.Aaa.Af$. Since the subsequence $\langle Sso, Ro\rangle$ can be repeated multiple times, the language described by this regular expression is infinite. The $^+$ operator indicates that a

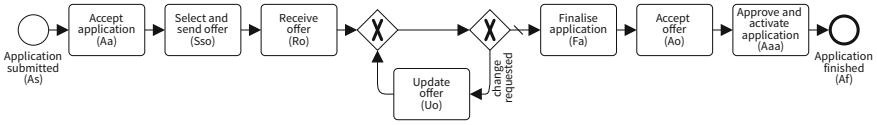


Figure 2.11 Example of a loop that can be skipped

subsequence must be repeated one or more times. The * operator is used when it can be repeated zero or more times, as indicated by the example in Figure 2.11. The infinite language of this model is $As.Aa.Sso.Ro.Uo^*.Fa.Ao.Aaa.Af$.

Using composition of regular expression operators, the language of the model of our running example in Figure 1.3 is given by the following regular expression: $As.(Da|(Aa.(Fa||(Sso.Ro.(Co.Sso.Ro)^*)).((Ao.Aaa)|(Do.Da)))).Af$.

Next, we turn the focus from the behaviour described by models to the behaviour recorded in an event log. Event logs can also be seen as languages. While technically, an event refers to an execution of an activity, it is not uncommon to represent an event simply by the identifier of the executed activity. An event log can therefore be seen as a finite set of sequences of activities. For example, the trace in Table 2.1 can be described as the language $As.Aa.Fa.Sso.Ro.Do.Da.Af$. Table 2.2 on the other hand contains three traces, i.e. the corresponding language is $(As.Aa.Fa.Sso.Ro.Do.Da.Af)|(As.Aa.Da.Af)|(As.Aa.Fa.Sso.Ro.Co.Sso.Ro.Ao.Aaa.Af)$.

So far, the traces we have seen all correspond to the model, i.e., each trace corresponds to an execution sequence of the model. However, we already elaborated on possible deviations between modelled and recorded behaviour. Consider for example the following trace $\langle As, Da, Aaa, Af\rangle$, i.e. an application is submitted, subsequently declined, then approved and activated and finally finished. This behaviour does not correspond to the model of our running example. Clearly, approval and activation of the application (Aaa) should not have happened. Similarly, consider a trace $\langle As, Aa, Sso, Ro, Do, Da, Af\rangle$. In this sequence, the application was not finalised (Fa), which is not in line with the model. Adopting the view of languages, both example traces lead to a situation where the sequence of activities given by the events of a trace is not part of the language of the process model.

It is important to realize that our language-based view on event logs is somewhat limited. We can, for example, no longer refer to the most frequent trace, since the respective language contains each sequence only once. However this language-based view on models and logs suffices for Part I of this book to outline the main ideas behind conformance checking.

Bibliographic Notes

There exist several monographs that introduce modelling notations similar to the one used in this book. A good reference for the interested reader is the book by Mathias

Weske [135], where not only the modelling of processes but also their execution is considered. A more technical reference to dive into the semantics of BPMN 2.0, the reference language used in this book for describing processes, can be found in [57] or [36].

Event logs are the principal input for process mining algorithms. For a detailed definition, we refer to the seminal book about process mining [114]. There exists now an approved XML-based IEEE Standard for event logs, which was developed in the scope of the IEEE Task Force of Process Mining[2].

2.5 Exercises

2.A) Define a process model from a textual description

Given the following textual description of a process, provide the corresponding process model using the constructs explained in this chapter.

The examination process can be summarized as follows. The process starts when the female patient is examined by an outpatient physician, who decides whether she is healthy or needs to undertake an additional examination. In the former case, the physician fills out the examination form and the patient can leave. In the latter case, an examination and follow-up treatment order is placed by the physician who additionally fills out a request form.

Beyond information about the patient, the request form includes details about the examination requested and refers to a suitable lab. Furthermore, the outpatient physician informs the patient about potential risks. If the patient signs an informed consent and agrees to continue with the procedure, a delegate of the physician arranges an appointment of the patient with one of the wards. The latter is then responsible for taking a sample to be analysed in the lab later. Before the appointment, the required examination and sampling is prepared by a nurse of the ward based on the information provided by the outpatient section. Then, a ward physician takes the sample requested. They further send it to the lab indicated in the request form and conduct the follow-up treatment of the patient.

After receiving the sample, a physician of the lab validates its state and decides whether the sample can be used for analysis or whether it is contaminated and a new sample is required. After the analysis is performed by a medical technical assistant of the lab, a lab physician validates the results. Finally, a physician from the outpatient department makes the diagnosis and prescribes the therapy for the patient.

[2] http://www.win.tue.nl/ieeetfpm

2.B) Where did the process go wrong?

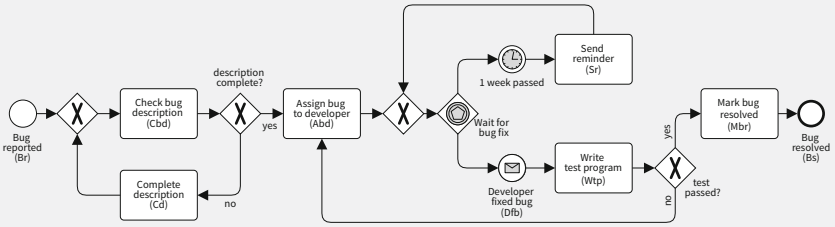


Figure 2.12 Example bug fix process

Given the process model in Figure 2.12 and the traces:

a) $\langle Br, Cbd, Cd, Cbd, Adb, Sr, Dfb, Wtp, Mbr, Bs\rangle$
b) $\langle Br, Cbd, Adb, Sr, Sr, Adb, Dfb, Wtp, Mbr, Bs\rangle$

Decide whether the traces are correct according to the model. If not, where is the error? Can the model be changed such that the two traces fit?

2.C) How to make the log match the model?

Given again the process model in Figure 2.12, the following two traces were recorded, i.e. they are the observed behaviour:

i) $\langle Br, Cbd, Sr, Abd, Dfb, Wtp, Abd, Dfb, Mbr, Bs\rangle$
ii) $\langle Br, Cd, Cbd, Adb, Sr, Dfb, Mbr, Bs\rangle$

What went wrong here? Correct the traces such that they fit the process model. Is there only one correct solution?

2.D) Regular expressions vs. process models

Find a regular expression that captures the traces that are allowed by the process model in Figure 2.13.

Hint: The sequential operator "." can be omitted in this exercise.

(continued)

2.D) (continued)

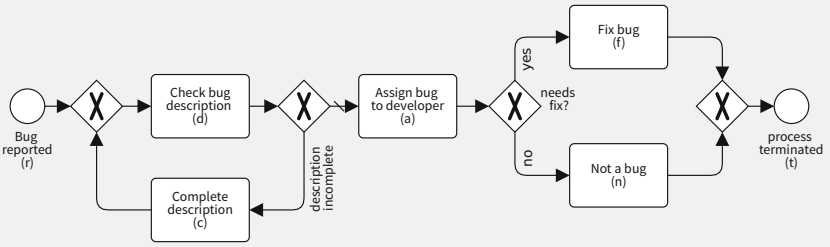


Figure 2.13 Shorter bug fix process

Chapter 3
Quality Dimensions for Relating Processes and Models

In the previous chapter, two important perspectives have been reported: First, process models were introduced as conceptual descriptions of the underlying process. Second, event logs were presented as the footprints recorded by information systems during the executing of a process. The behaviour represented by a process model may be either descriptive (the goal of the model is to show the reality), or prescriptive (the model defines how the reality should be). In any of these two possibilities, it is of paramount importance to relate the two perspectives of modelled behaviour versus recorded behaviour to obtain insights on the capability of the model to describe what is observed in the information systems supporting a process.

In this chapter, we focus on the two main high-level dimensions to assess the relation between process models and event logs. These dimensions are inspired by the well-known notions of *recall* and *precision* from classification, where they measure the cost of different mistakes that can be made in relating observations with a model. Although the underlying motivation is the same, in this chapter we show that in the context of conformance checking, we need a tailored definition to accommodate the particularities of process models and event logs.

Overall, the intuition between the two quality dimensions that is presented in this chapter is depicted in Figure 3.1. The fitness dimension focuses in assessing the capability of a process model in reproducing the recorded behaviour, thus the direction of the arc from the event log to the process model: for each trace in the event log, the process model is queried to identify the model's capability to reproduce it. Symmetrically, the precision dimension targets the amount of behaviour that exists in the process model that has actually been recorded, thus the direction of the arc from the process model to the event log. One can see the similarity between the intuitive idea of fitness/precision introduced here and the established notions of recall/precision in classification or information retrieval.

J. Carmona et al., *Conformance Checking*,
https://doi.org/10.1007/978-3-319-99414-7_3

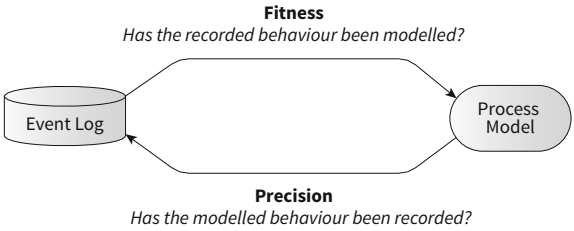


Figure 3.1 General view on the two quality metrics considered: fitness and precision

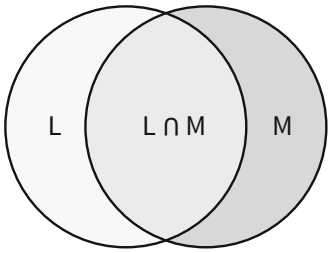


Figure 3.2 Visualization of recorded (L) and modelled (M) behaviour and their overlap ($L \cap M$)

In Section 3.1 we reflect on the conceptualization necessary to relate recorded and modelled behaviour. Then in Section 3.2 we focus on the fitness dimension, while in Section 3.3 we overview the precision dimension. Then we turn the attention to explain deviations in Section 3.4. Finally, a method to estimate precision is presented in Section 3.5.

3.1 Behaviour of Processes

When trying to confront event logs and process models, an important consideration should be taken into account: The size of the behaviour described. An event log describes finite behaviour, that corresponds to the set of traces recorded throughout the execution of a process. For instance, for the running example of a loan application process, as implemented in a large European bank, we obtain a log comprising 9629 traces. In contrast, a process model may describe infinite behaviour, in case the process model contains iterative constructs. For instance, the process model of Figure 1.3 defines, among others, any execution sequence that satisfies the following regular expression: $As.Aa.Sso.Ro.(Co.Sso.Ro)^*.Fa.Ao.Aaa.Af$. Thus, although having a compact representation, it represents an infinite behaviour.

The rest of this chapter presents metrics to evaluate the quality of a process model with respect to an event log. The Venn diagram of Figure 3.2 shows the conceptual picture. Process models and logs can be seen as sets of behaviours, and by analysing

their commonalities and differences, one can inspect the two quality dimensions considered in this chapter: fitness (Section 3.2) and precision (Section 3.3).

Excursion 7

The underspecified process

Sometimes, it is not desired to specify the control flow in a model rigidly. The BPMN language has a construct to express that at some point the process can arbitrarily continue with a set of activities. The so called ad hoc subprocess allows us to execute any task in the given set in any order arbitrarily often. It is denoted with the tilde "~" sign at the bottom.

This way, a model that allows us to execute any task within our running example would look as in Figure 3.3.

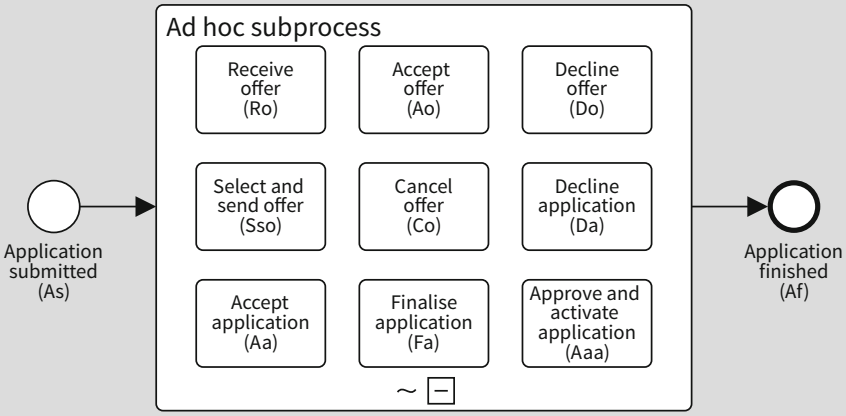


Figure 3.3 The underspecified version of the running example using the ad hoc construct

Note that such underspecified processes can always reproduce any behaviour. If you think of the model as a form that captures the behaviour of the process, the ad hoc subprocess is indeed a very flexible form, into which you can fit any recorded behaviour of the given activities. On the one hand, this provides us with the sometimes required flexibility in a process. On the other hand, however, the model does not really assist in specifying what should be done in what order. All control is in the hands of the (hopefully experienced) process participant.

The language produced by the model in Figure 3.3 can be expressed as $As.(Ro|Ao|Do|Sso|Co|Da|Aa|Fa|Aaa)^*.Af$.

3.2 Fitness

Fitness measures the ability of a model to explain the execution of a process as recorded in an event log. It is the main measure to assess whether a model is well-suited to explain the recorded behaviour. To explain a certain trace, the process model is queried to assess its ability to replay the trace, taking into account the control flow logic expressed in the model.

In general, fitness is the fraction of the behaviour of the log that is also allowed by the model. With Figure 3.2, fitness can be expressed as follows.

$$\text{fitness} = \frac{|L \cap M|}{|L|} \tag{3.1}$$

Let us have a look at this fraction in more detail by examining the extreme cases first. Fitness is 1, if the entire behaviour that we see in the log L is covered by the model M. Conversely, fitness is 0, if no behaviour in the log L is captured by the model M. Figure 3.4 conceptually shows these extreme cases side by side.

For the following discussion, we define a trace to be either fitting (it corresponds to an execution sequence of the model) or non-fitting (there is some deviation with respect to all execution sequences of the model). This simplification is made to illustrate the problem, and refinements of the measure to be more sensitive to smaller or larger deviations are discussed in Part II of this book.

With this basic definition of fitness, let us next illustrate fitting and non-fitting behaviour with particular traces and our running example model in Figure 1.3. This helps to separate the behaviour that we see in log L and that is represented in the model (i.e., in $L \cap M$) from the behaviour that is outside the model (i.e., in $L \setminus M$). With this distinction, computing the fitness according to Equation 3.1 is reduced to simply counting the number of traces that fit the model and dividing that number by the number of all the traces in the log.

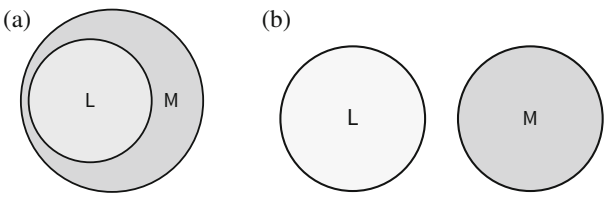


Figure 3.4 Conceptual depiction of extreme fitness levels. (**a**) Visualization of recorded (L) and modelled (M) behaviour with fitness 1 ($L \subseteq M$). (**b**) Visualization of recorded (L) and modelled (M) behaviour with fitness 0 ($L \cap M = \emptyset$)

3.2.1 Fitting Behaviour

Let us first look at a short and simple trace. Given the trace $\langle As, Da, Af \rangle$, we can track the path in the model in Figure 3.5 and identify a corresponding execution sequence. First, the application is submitted (As), then the (exclusive) choice leads to the application being declined (Da). Eventually, through an exclusive join gateway, we reach the final step in which the application is finished (Af). Since this path denotes an execution sequence, the trace of the log is considered fitting, i.e., it is part of the intersection of the log's and the model's language $L \cap M$.

Let us consider trace $\langle As, Aa, Sso, Fa, Ro, Do, Da, Af \rangle$, for which the respective path is highlighted in Figure 3.6. This trace is also fitting, and leads to the application being declined in the end. After submission (As), the decision is made to accept the application (Aa). Then, the process is split into two parallel branches: finalising the application and the potential iterative refinement of the offer. Here, the offer is selected and sent (Sso), before the application is finalised in parallel (Fa). After that, the offer is received without requested changes. This closes the parallel part. However, at this point the offer is declined (Do) leading to the application being declined (Da) and finally, the application being finished (Af). Again, this

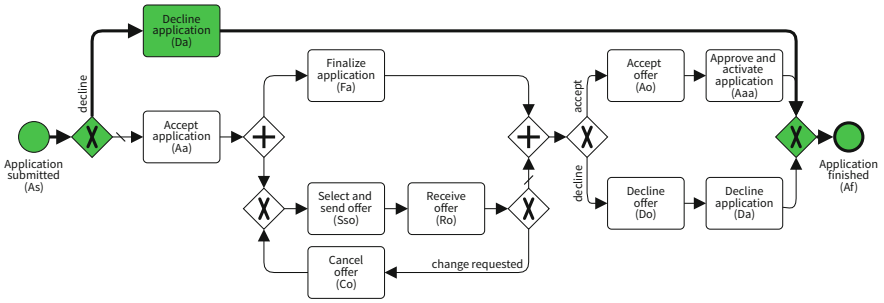


Figure 3.5 Loan application process model with highlighted path corresponding to one trace

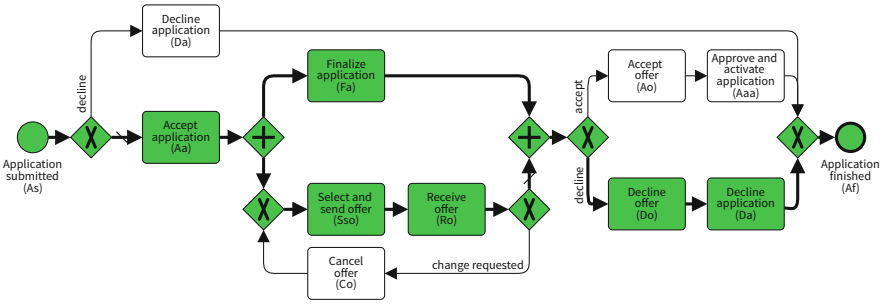


Figure 3.6 Loan application process model with highlighted path corresponding to another trace

corresponds to an execution sequence of the model, which renders the trace a fitting one. Finding many such traces that fit the model increases the value of fitness of the model. Next, we shall have a look at traces that are not reproducible in the model.

3.2.2 Non-fitting Behaviour

Let us assume that our log L also contains the following trace $\langle As, Aa, Sso, Ro, Do, Da, Af \rangle$, which is similar to the previous one. Looking more closely, we can see that in comparison to the previous trace, it does not contain any event that signals that the application has been finalised (Fa). Figure 3.7 highlights the part where the model does not fit to the trace. In fact, the model prescribes that the application must be finalised (Fa) after it has been accepted (Aa) and before either accepting the offer (Ao) or declining the offer (Do). Thus, when we try to replay this trace in the model, we eventually see that the model requires to first finalise the application before declining the offer. This mismatch between the trace and the model means that the there is no corresponding execution sequence for the trace, i.e., the trace is non-fitting. In the Venn diagram, this trace belongs to the part of the log L that is outside of the model M. The more of these non-fitting behaviour we encounter in an event log, the smaller is the intersection of log and model and the less fitting is the corresponding model.

Here, we saw non-fitting log behaviour where modelled behaviour is not in the trace. In general, however, there can be also excess behaviour in a trace, which means that at some point during replay, we find (at least) one event in the trace that signals an execution of an activity, whereas the corresponding task is not enabled in the current state of the model. To this end, let us assume that the log also contains the non-fitting trace $\langle As, Da, Aaa, Af \rangle$. After the application was declined, it was approved and activated afterwards. The path in the model is shown in Figure 3.8. There is a deviation, as an event signalled an execution of an activity (after decline application) that is not in line with what is captured in the model.

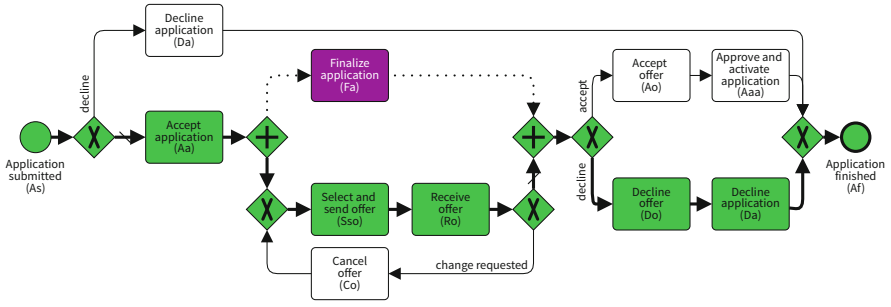


Figure 3.7 Loan application process model with highlighted path corresponding to one trace, which does not include an event to signal that the application has been finalised (Fa)

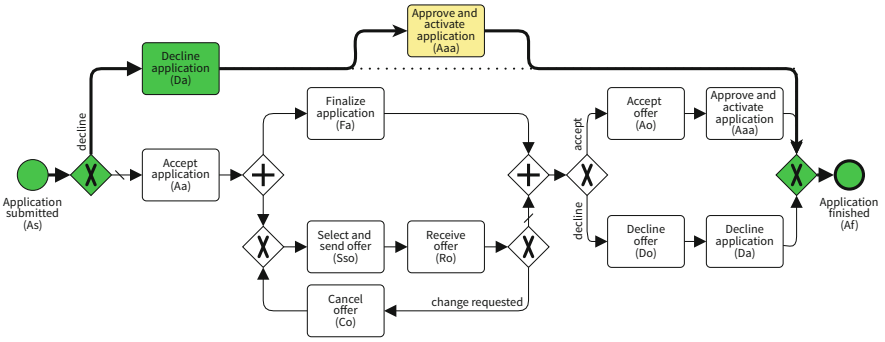


Figure 3.8 Loan application process model with highlighted path corresponding to one trace, where an event that signals the approval and activation of the application (*Aaa*) is present after the application has been declined (*Da*), which is not in line with any execution sequence of the model

A trace can have more than one deviation, and the deviations can be of either missing or excess types. The decision about which part of a trace is non-fitting can be less obvious than in the examples we have seen here. Alignments provide a solution to this decision, which we discuss in detail in the next chapter.

To sum up, the notion of fitness measures how well the behaviour in the log is captured by the model. When the model is *prescriptive*, that is, it contains guidelines that should not be violated, a high value of fitness is an indicator that, based on the recordings in an event log, the process is correctly executed (i.e. according to plan). Conversely, a low fitness value indicates that there is a problem, and the behaviour recorded in the log cannot be correctly mapped to the prescribed behaviour. If so, however, we cannot immediately conclude that the process is not executed according to the guidelines or regulations. It is also possible that there are other deviations between model and log which do not violate the guidelines. Therefore, we need to carefully analyse the deviations and their causes.

Next, we look at the behaviour in the model and ask the conformance question from the model's viewpoint.

3.3 Precision

In the previous section, we discussed the fitness metric that measures how much of the recorded behaviour is captured by the model—and how much of it deviates. Now, we turn this question around and ask how much of the modelled behaviour is actually recorded in an event log. We already saw one example model—the under-specified model in Figure 3.3—that is perfectly fitting for all the traces in the event log L, but does not explain how the process shall actually be conducted. The ad hoc model allows for any behaviour between the respective start and end of the process.

Precision helps us to quantify the behaviour of the model with respect to the log. Thus, an underspecified model that allows us to do anything scores low on precision.

In Figure 3.2, precision is the counterpart of fitness. It can be calculated by looking at the fraction of the model behaviour that is covered in the log.

$$\text{precision} = \frac{|L \cap M|}{|M|} \tag{3.2}$$

We see that precision shares the numerator in the fraction with fitness from Equation 3.1. This implies that if we have a log and a model with no shared behaviour (cf. Figure 3.4b), fitness is zero, and by definition also precision is zero. However, the denominator is replaced with the amount of modelled behaviour.

3.3.1 Precise Behaviour

Precise behaviour in the model is given by execution sequences of the model for which a corresponding trace is part of the event log. This coincides with the fitting behaviour that we illustrated in Section 3.2.1. Therefore, when we compute fitness, we can keep track of the fitting traces for future reference in order to compute precision.

3.3.2 Imprecise Behaviour

Let us recall the underspecified model in Figure 3.3. This model has the following execution sequences, among others:

$$S_1 : \langle As, Aa, Af \rangle$$

$$S_2 : \langle As, Sso, Sso, Sso, Sso, Af \rangle$$

The first sequence S_1 is simply finishing the application (Af), even though it was not finalised (Fa), no offer was sent to the customer (Sso), and the application is neither approved (Aaa), nor declined (Da). The second sequence S_2 is spamming the customer with offers without waiting for any response. These examples exhibit undesired behaviour, and do not represent valid behaviour of the loan application process. Also any other permutation of the tasks in the ad hoc subprocess are encoded in the model. This is an extreme example of very low precision. If we think about it, this model does not contain any meaningful information except the list of possible activities of the process. Their number and order of execution is not constrained by the model. Specifically, the allowed behaviour is much more than what we observe, i.e., $|M| >> |L|$. In the spectrum of precision shown conceptually in Figure 3.9, this corresponds to the latter case, illustrated in Figure 3.9c.

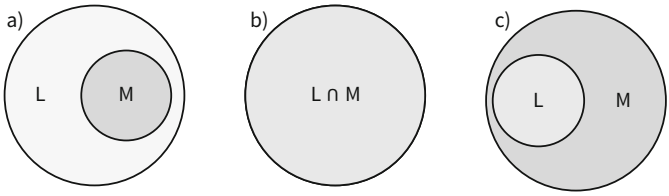


Figure 3.9 Spectrum of (**a**) precise, but not fitting, (**b**) precise and fitting (L = M), and (**c**) imprecise and fitting

In fact, the behaviour in models that allow recurrence is infinite. The reason is that we can repeat the iterative part once, or twice, or *any number* of times. How can we compute precision then? If the behaviour of M is infinite (i.e., $|M| = \infty$), the fraction of precision reduces to 0 by definition. This is a problem, because models with iterations are valid models, as there can be repetitive tasks, which we do not want to spell out by making each iteration explicit.

To have a meaningful comparison of precision between event logs and models with iterative constructs, precision is approximated. The approximation can be done in different ways; at the end of this chapter we informally explain the simplest (yet the most widely used) technique to approximate precision. We refer the interested reader to the second part of this book (in particular, Section 8.5.1), for a detailed and formal explanation on techniques to approximate precision. Bibliographical notes on this particular topic are reported at the end of this chapter.

Excursion 8

Filling the precision spectrum

Let us have a look at three additional process models in comparison to the example process model shown in Figure 1.3.

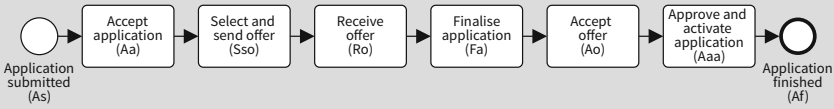


Figure 3.10 Simple model M_1

(continued)

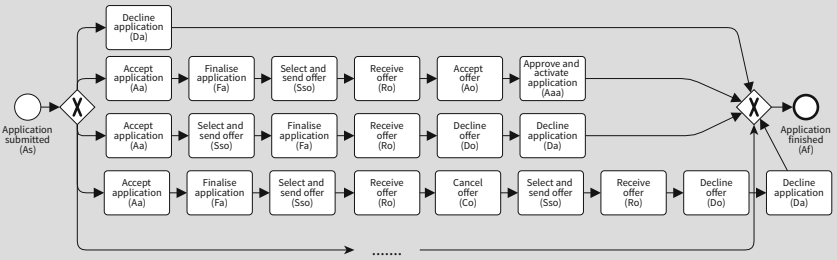


Figure 3.11 Enumerated model M_2

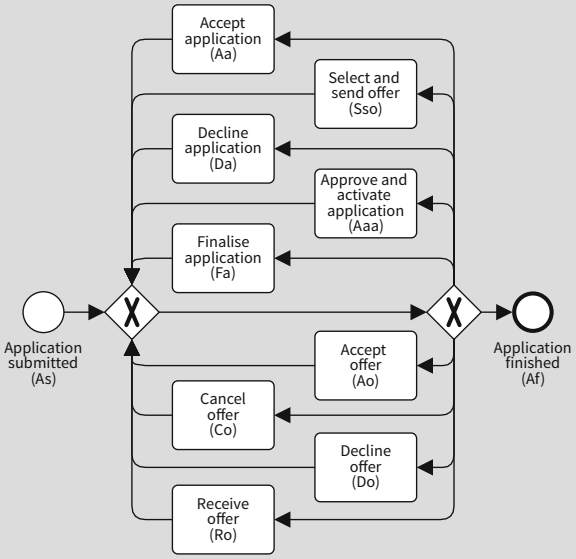


Figure 3.12 "Flower" model M_3

First, let us have a look at model M_1 in Figure 3.10. This is a simple model that defines a single execution sequence $\langle As, As, Aa, Sso, Ro, Fa, Ao, Aaa, Af \rangle$, which denotes a successful loan application. Although M_1 can represent an important case of the process model, it is unable to describe other cases, like the execution sequence $\langle As, Da, Af \rangle$, which denotes immediate rejection of a loan application. The latter situation often happens in reality, as recorded in the event log of this running example. Hence, model M_1 is not fitting event log L.

(continued)

Now, consider the model M_2 on Figure 3.11: it contains corresponding execution sequences for all traces of the event log (we only show four of them in the figure). According to the precision direction, Figure 3.1, that requires that traces of the log be matched by execution sequences of the model, M_2 is perfectly precise. Moreover, all execution sequences of model M_2 correspond to traces recorded in the event log, and therefore M_2 is fitting. Let us have a look at a potential execution sequence $\langle As, Aa, Sso, Ro, Co, Sso, Ro, Fa, Do, Da, Af \rangle$, which represents a failed application that required us to modify the offer twice. Clearly, this option should be possible, but M_2 does not allow for it, since the task to finalise an application (Fa) can only be executed before the tasks to select and send an offer (Sso), receive an offer (Ro), and cancel an offer (Co); see the bottom branch of the model in Figure 3.11. By capturing only execution sequences that correspond to traces that appear in the event log, and not more, M_2 is *overfitting* the event log L.

Consider now the model M_3 in Figure 3.12, that we informally call the *flower model*. This model has the same language as the model in Figure 3.3. In this model, every execution sequence starting with the application being submitted (As) and ending with the application being finished (Af) is possible. Thus, for every trace in the log L, there is a corresponding execution sequence in the model, making the model M_3 fitting for L. However, the precision of M_3 with respect to L is poor, since many execution sequences of this model, like $\langle As,Do,Ao,Fa,Af \rangle$, are allowed for by M_3, but corresponding traces are never observed in the log. Hence, although being fitting, M_3 is not able to explain the recorded behaviour, but, instead, denotes an *underfitting* model.

In terms of the spectrum depicted in Figure 3.9, model M_1 in Figure 3.10 belongs to the category (a) precise, but not fitting; M_2 is (b) fitting and precise; whereas model M_3 is (c) imprecise and fitting.

3.4 Reasons for Deviations

To reason about causes of deviation between modelled and recorded behaviour, we can classify the non-fitting behaviour that is outside the model into possible explanations. There are two obvious explanations. On the one hand, we can assume that for an event that is missing, even though a corresponding task would have to be executed according to the model, the respective activity was actually executed in the particular case of the process. If so, this is an issue of recording the activity execution, leading to the absence of the event from the log. On the other hand, we can assume that the activity execution did not happen in the case.

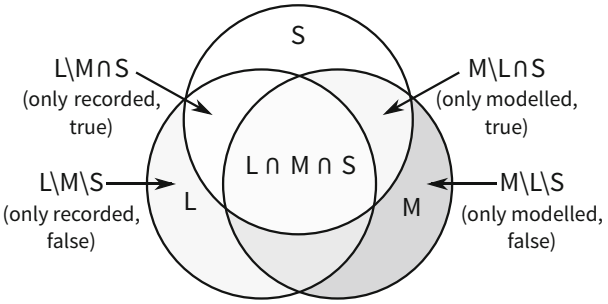


Figure 3.13 Visualization of recorded (L), modelled (M), and real behaviour (S) and their overlaps; cf. [18]

The same reasoning can be applied to explain excess behaviour in the event log with respect to the model. We can assume that there was an activity executed, which is simply not modelled, or we can assume that the event is erroneous. For example, when recording is done manually or based on sensed data that is not trustworthy, these false positives can occur.

We lean on the view, as introduced by Buijs and colleagues [18], that includes the *system S*, for which the behaviour is defined by the process, in addition to the dual perspective of modelled behaviour and recorded behaviour, as introduced before. We depict this view with a Venn diagram in Figure 3.13. In the figure, the four arrows point to the respective behaviours.

3.4.1 Interpreting Non-fitting Behaviour

Conceptually, we can separate the *non-fitting* behaviour that is recorded but not captured in the model, $L \setminus M$, from the behaviour that really occurred, $L \setminus M \cap S$, and the behaviour that was recorded but did not occur in reality, $L \setminus M \setminus S$.[1] We see this on the left-hand side of Figure 3.13.

The non-fitting behaviour that occurred in the system represents the situation that the log is correct, but the model is wrong, as it does not capture the true behaviour. Such a situation is best treated by refining the model at the position of deviation, if our goal is to have models that correctly represent the behaviour. There exists a trade-off, however, as "repairing" the model at many of these deviations can lead to overly complex models that become no longer usable.

[1] In the set expressions, we assume operator $\setminus$ has higher precedence than $\cap$, i.e., $L \setminus M \cap S$ is equivalent to $(L \setminus M) \cap S$. Also, we assume operators to be left-associative, i.e., $L \setminus M \setminus S$ is equivalent to $(L \setminus M) \setminus S$.

The non-fitting behaviour that did not occur in the system represents the situation that the log is incorrect or noisy. These anomalies in the log indicate that the generation of the log contains erroneous sources. This may stem from manual logging of events (possibly even scanned with optical character recognition), or when we try to make sense of the process through noisy sensed data.

3.4.2 Interpreting Imprecise Behaviour

On the right-hand side of this figure, we can distinguish the *imprecise* behaviour that is only modelled and not recorded ($M \setminus L$) into modelled behaviour that actually occurs in the system, $M \setminus L \cap S$, and modelled behaviour that does not occur, $M \setminus L \setminus S$. The former denotes a situation in which we truthfully capture behaviour in the model, which is not recorded—it is done and modelled, but not recorded. The latter represents excess behaviour that is either just wrong (imprecise model), or is caused by the modelling language that can allow infinite repetitions, even though in reality the process is bounded to a given number of repetitions.

Imprecise behaviour that occurs in the system hints at a documentation issue in the process, as the log does not capture this part. Possibly, one could decide to implement additional logging/monitoring for these parts in the process, to also be able to monitor and control it. But economic feasibility should always be kept in mind, as the gain in transparency might not justify the additional overhead to capture these parts in the event log of the process.

Imprecise behaviour that is allowed and represented in the model, but occurs neither in the log, nor in reality is a candidate for removal from the model. Sometimes this excess behaviour is due to well-justified generalization. For example, the model might allow us to execute two tasks in parallel at a point (thus allowing any order), because there is no dependency between the respective activities. If, however, this process is performed by a single person, who always executes these activities in the same order, the log does not contain corresponding traces for all possible execution sequences allowed by the model. The imprecision in the model is then a justified generalization of the process, even though in reality, the flexibility of the process is not fully exploited. A similar situation is obtained with a loop in a process model that indicates arbitrary repetition of a part of the model. In reality, we might always only see a handful of repetitions, although the model in theory allows us to do a hundred, a thousand, or infinite iterations. Nevertheless, we would rather choose a simple model that contains one loop, instead of a model that explicitly captures one, two, three, four, or five iterations of a part of the model.

3.4.3 Completing the Picture

For the sake of completeness, we briefly mention the other parts in Figure 3.13. The top white part, $S \setminus L \setminus M$, is the behaviour that occurs in reality, but is neither recorded, nor modelled. In general, there is usually a lot of behaviour in that region, as models capture only a specific perspective and an abstraction of the process that describes the behaviour of a system. The central area in the figure, $L \cap M \cap S$, represents the behaviour that is recorded, modelled, and actually occurred in reality. In conformance checking, this is the part that we can try to maximize by correcting the model, or reducing causes of errors in documentation and logging.

Directly below the centre is the behaviour that is both modelled and logged, but did not happen in reality, $L \cap M \setminus S$. This is a rather theoretical construct, as the coincidence of falsely recording some behaviour which did not occur, while at the same time also modelling it, is expected to be low. There is one exception to this, though. When the model is automatically created from the log by process discovery techniques in process mining, noisy behaviour that affects the log but did not actually happen could be represented in the model as well.

Excursion 9

The overfitting problem

This additional perspective of the *system* helps us to better understand the problem of overfitting. The problem occurs when our model fits the recorded behaviour, but the recorded behaviour is only representing a fraction of the real process of the system. Figure 3.14 depicts this situation. We see that the model and the log are perfectly aligned in their behaviour ($L = M$), but they are only representing a part of the actual behaviour of S. The problem occurs in practice when we know only little of the process, because we only observed a part of it in our log L, or we are novice process modellers, and try to create a model that *exclusively* represents the recorded part. In this case, we do not generalize at all.

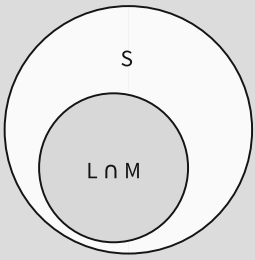


Figure 3.14 Visualization of an overfitting model that only captures behaviour recorded in the log

(continued)

For example, the model M_2 in Figure 3.11 suffers from the problem of overfitting. It defines only execution sequences that represent exactly all traces recorded in the log. However, if we assume that the real process is the one depicted in Figure 1.3, many plausible cases of the process are not represented in M_2.

3.5 Precision Approximation

The reader may have noticed that we left some issue open in Section 3.3, when presenting the formula (Equation 3.2) for computing precision. In this section, we provide an intuitive explanation on how precision can be approximated. The second part of this book completes the picture, providing a detailed description of algorithms for the computation of this important metric.

Precision can be approximated by exploring the behaviour of the model using as a reference the traces of the log, and stopping the exploration each time modelled behaviour deviates from recorded behaviour. Figure 3.15 illustrates the main idea. To simplify the explanation, we focus on the fitting part of the model behaviour for precision approximation. In the second part of this book the reader can find a general description, where this restriction is dropped.

The core idea to approximate precision is to focus only on finite prefixes of the execution sequences of a model. Specifically, by focusing on those finite prefixes that match prefixes of traces in the log, a precision metric can be obtained. For each possible prefix σ of a trace, for which there is a corresponding execution sequence

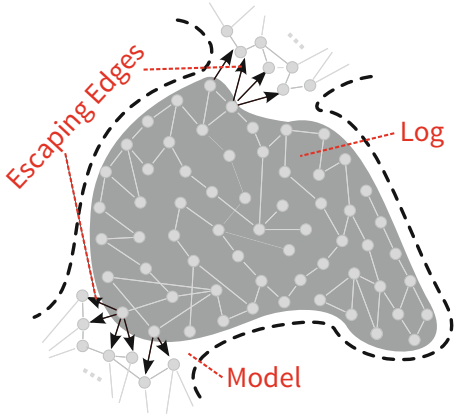


Figure 3.15 Idea on approximating precision (figure taken from [72])

in the process model, the model can be queried to determine those tasks a for which σ followed by a is *not* a prefix of a trace. These tasks, called *escaping arcs*, denote states in the model that start to deviate from the behaviour seen in the event log. Importantly, since both the event log L and the set of tasks of a model A are assumed to be finite sets, the set of escaping arcs is finite, as illustrated in Figure 3.15.

If Pref(X) denotes the set of prefixes of the words in language X, examples of elements in Pref($L \cap M$) for the model of Figure 3.12 are:

$$\langle\rangle$$
$$\langle As\rangle$$
$$\langle As, Aa\rangle.$$

An example of an element in Pref($M \setminus L$), in turn, is:

$$\langle As, Aa, Af\rangle$$

Here, the last entry of the sequence (Af) corresponds to the escaping arc.

The more sequences exist in Pref($M \setminus L$), the less precise is the model with respect to the log. Hence, by computing all the escaping arcs leading to sequences in Pref($M \setminus L$), a simple metric for approximating precision can be obtained:

$$\text{precision}_a = 1 - \frac{|\text{Pref}(M \setminus L)|}{|\text{Pref}(M \setminus L)| + |\text{Pref}(L \cap M)|} \tag{3.3}$$

Extreme cases of the formula above correspond to corner cases of precision; for instance, if no escaping arcs exist, then the set Pref($L \setminus M$) is empty, and so is the fraction, which yields a precision of 1. Conversely, if Pref($L \setminus M$) $>>$ Pref($L \cap M$), then the fraction tends to evaluate to a number near 1, and therefore the metric tends to zero.

Bibliographic Notes

Metrics for the evaluation of fitness have appeared in the last decade. In [93], the notion of *token replay*-based fitness is introduced, where traces in the event log are replayed and deviations are estimated over the missing or remaining tokens that arise during the replay. In presence of non-deterministic process models, replay cannot guarantee the most similar model trace that can replay the recorded behaviour, and therefore the methods in [93] present heuristics to look ahead in the replay so that non-deterministic choices are decided with more information about next possible steps of the process. As we detail in the next chapter, wrong choices in the replay may affect the accuracy of the diagnosis extracted from replay-based fitness.

The work in [1] overcomes the aforementioned limitations by incorporating the notion of alignment, and grounding the computation of fitness on top of the alignments. Since an alignment provides a best execution sequence of the model that can reproduce the recorded trace, alignment-based fitness is an accurate metric. Fitting or unfitting traces, as has been shown in this chapter, are aligned to a process model so that an explanation of the recorded behaviour in reference to the model (like the ones in Figure 3.5, Figure 3.6, Figure 3.7 or Figure 3.8) can be extracted. The approach can easily incorporate costs assigned to deviations, so that unimportant or less-likely deviations are avoided by assigning suitable costs.

The first metric for precision was presented also in the scope of [93], denoted *behavioural appropriateness*. This metric is based in comparing the ordering relations between events (*sometimes/always/never* follows and precedes relations) in the model with the ones from the log. Since an exhaustive exploration of the model state space is required to compute the relations corresponding to the process model, this approach suffers from the well-known state space explosion problem.

The metrics for precision described in this chapter are based on the PhD thesis of Jorge Munoz-Gama, where the notion of *escaping arcs* first appeared [71]. By bounding the exploration of the model state space to those parts that are visited by the traces in the log, this precision metric represents an efficient technique to estimate precision. As it was done with fitness, the escaping arcs technique can be defined on top of alignments, so that an alignment-based precision metric can be provided [4]. Also, further information can be incorporated so that confidence and stability analysis is provided on top of the precision metric [73].

The Venn diagram view of a system (process) vs. process model vs. event log in Figure 3.13 was first introduced by Joos Buijs et al. in [18], to show that it is possible to use multiple dimensions to quantify the quality of process models with respect to event logs.

3.6 Exercises

3.A) Report a fitting trace from a model

Given the model of Figure 1.3, define a fitting trace.

3.B) Report an unfitting trace from a model

Given the model of Figure 1.3, define an unfitting trace.

3.C) Which model is more fitting?

In the following figures, there are three examples of process models, namely M_1, M_2 and M_3 (Figures 3.16, 3.17, and 3.18).

(continued)

3.C) (continued)

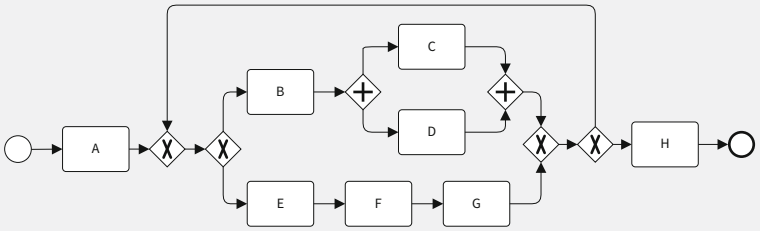


Figure 3.16 M_1

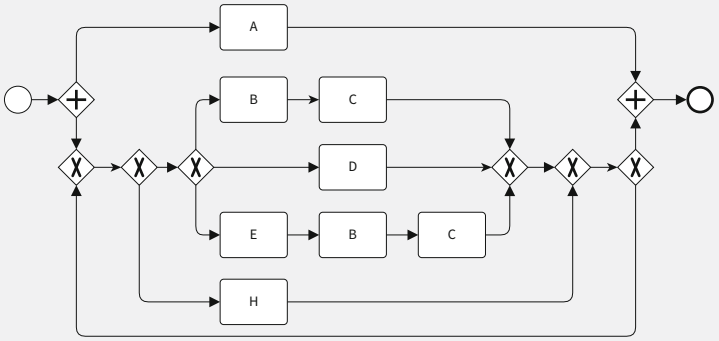


Figure 3.17 M_2

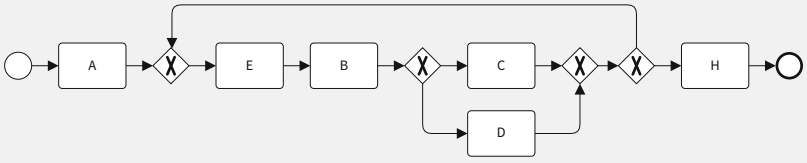


Figure 3.18 M_3

Given the trace $\langle A, E, B, C, B, D, H \rangle$, choose which model fits best to the trace.

3.D) Analysing fitness

Given the models M_1, M_2 and M_3, evaluate their corresponding fitness (using Equation 3.1) on the following event log: $\{\langle A, E, B, C, B, D, H\rangle, \langle A, E, F, G, H\rangle, \langle A, B, C, B, C, H\rangle, \langle A, E, F, G, B, C, B, D, H\rangle\}$.

3.E) Analysing precision

Given the model M_3, evaluate its corresponding precision (using Equation 3.3) regarding the following event log: $\{\langle A, E, B, C, B, D, H\rangle, \langle A, E, F, G, H\rangle, \langle A, B, C, B, C, H\rangle, \langle A, E, F, G, B, C, B, D, H\rangle\}$.

Chapter 4
A First Take on Conformance Checking

In the previous chapter, we discussed fitness and precision as two important dimensions when relating process models and event logs. By referring to the behaviour of a model and a log in abstract terms, these notions are independent of any specific operationalisation. That is, conformance checking in general, and any assessment of fitness and precision in particular, is based on a choice of a specific formalism for the behavioural comparison of models and logs. This chapter reviews three such formalisms, thereby outlining different avenues on how to ground conformance checking. As illustrated in Figure 4.1, conformance checking can be approached by means of rule checking, token replay, or alignments.

In Section 4.1 we begin with conformance checking based on rules that capture the behaviour of a model by means of binary relations over the activities represented by the tasks of the model. In that case, conformance checking is interpreted as the verification of consistency of the rules derived from the model with the traces of the event log. While this approach can be instantiated for different sets of rules, of which many can be computed efficiently, there is a general issue in terms of completeness of the conformance checking result.

A second approach, therefore, based on a replay of the traces of an event log in the process model, is described in Section 4.2. In essence, this approach verifies to what extent a trace qualifies as an execution sequence of the model. The trace is replayed in the model event by event, checking in each replay step whether the conditions for the execution of the corresponding task are satisfied by the current state of the model. As we discuss in the remainder of this chapter, this approach overcomes some of the completeness issues of rule checking. Yet, in the general case, replay may still suffer from incompleteness that is due to a lack of a systematic exploration of non-deterministic choices in the replay procedure.

Against this background, in Section 4.3 we then turn to conformance checking based on alignments. The main idea of alignments is to link each event of a trace to at most a single task execution of an execution sequence of the model, and vice versa. This takes into account that both, events and task executions, may be without

J. Carmona et al., *Conformance Checking*,
https://doi.org/10.1007/978-3-319-99414-7_4

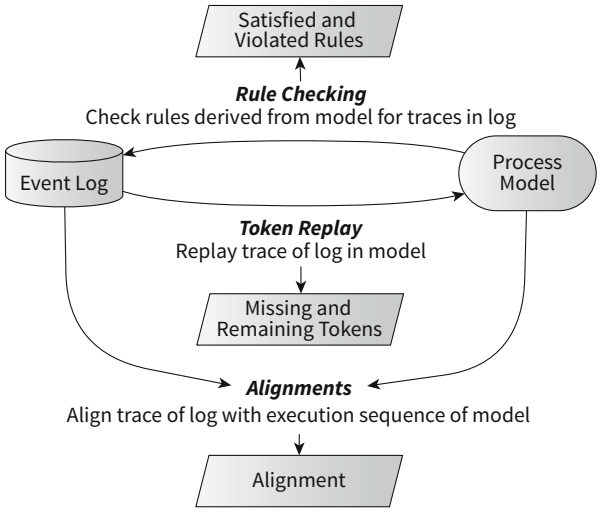


Figure 4.1 General approaches to conformance checking and resulting conformance artefacts

counterpart. As such, alignments can be seen as an evolution of the aforementioned approaches to conformance checking that overcome their asymmetric view on conformance. Instead of verifying rules induced by a supposedly correct process model or replaying a supposedly correct trace in the model, alignments follow a symmetric approach of balancing deviations between an event log and a process model.

When considering their application in reality, the three methods described in this chapter are complementary. Rule checking provides a simple yet effective way of analysing the fulfilment of a desired set of control flow rules, enabling us to focus only in certain parts of the process. Likewise, token replay may be applicable when the interest is on evaluating early deviations, or just to have an intuition of the replayability of traces. Alignments are applied when a detailed, fine-grained and complete analysis is necessary, or when further enhancement of the model is performed.

In the spirit of a *first take* on conformance checking, this chapter explains the above techniques for conformance checking from an intuitive point of view, focusing on the ideas behind rather than comprehensive formalisation. To this end, we rely on the running example of a loan application process, which was discussed already in the previous sections. For the sake of easy reference, the respective process model is repeated in Figure 4.2.

The following overview of conformance checking techniques concludes the first part of this book. Alignments as discussed towards the end of this chapter, however, provide a generic formalism for all kinds of further conformance checking techniques. The second part of this book, therefore, is devoted to an in-depth discussion of alignments.

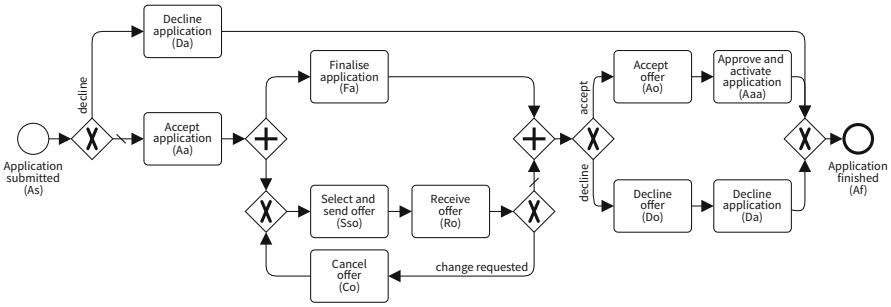


Figure 4.2 Recap of the process model of a loan application process, illustrated already in Figure 1.3 on page 8

4.1 A Gentle Introduction to Rule Checking

Rule checking is a simple, first approach to conformance checking. Below, we reflect on the basic idea behind this approach and discuss common formalisms for rule checking. Then, we illustrate rule checking with our running example and discuss the provided feedback on non-conformance.

4.1.1 Foundations of Rule Checking

A process model defines a set of tasks along with causal dependencies for their execution. As such, a process model constrains the possible behaviour of a process in terms of its execution sequences. Instead of considering the set of possible execution sequences of a process model, however, the basic idea of rule-based conformance checking is to exploit rules that are satisfied by all these sequences as the basis for analysis. Such rules define a set of constraints that are imposed by the process model. Verification of these constraints with respect to the traces of an event log, therefore, enables the identification of conformance issues.

Considering the running example of our loan application process as depicted in Figure 4.2, rules derived from the process model include:

R1: An application can be accepted (*Aa*) at most once.
R2: An accepted application (*Aa*), that must have been submitted (*As*) earlier, and eventually an offer needs to be selected and sent (*Sso*) for it.
R3: An application must never be finalised (*Fa*), if the respective offer has been declined (*Do*) already.
R4: An offer is either accepted (*Ao*) or declined (*Do*), but cannot be both accepted and declined.

By assessing to what extent the traces of a log satisfy the rules derived from a process model, rule-based conformance checking focuses on the fitness dimension,

i.e., the ability of the model to explain the recorded behaviour. Traces are fitting, if they satisfy the rules, or non-fitting if rules are violated. The dimension of precision—how much of the modelled behaviour is actually recorded—is not targeted by rule-checking. That is, specificity of rules is neglected; there could be many more traces, not contained in the log, that would also satisfy the rules.

A specific operationalisation of this idea first requires the choice of a formalism for the rules. A common approach is to rely on *unary* or *binary* rules over activities. A rule thereby specifies how a *single* activity is executed or how the executions of a *pair* of activities relate to each other. Note that either type of rule may still be related to the executions of multiple tasks, since a process model may comprise multiple tasks that represent the same activity. An example for that is given in Figure 4.2, where two tasks represent the activity to decline an application (*Da*). The reason being that this activity may happen in different contexts, either immediately after the submission of an application (*As*) or after it has been finalised (*Fa*) and an offer has been received (*Ro*).

The restriction to unary and binary rules is a pragmatic one, as the number of n-ary rules that could be considered is exponential in the number of activities that are captured in the process model. However, even when limiting conformance checking to unary and binary rules, various different types of rules could be considered, such as:

- Cardinality rules: A rule defines an upper and lower bound for the number of executions of an activity. Any complete execution sequence of the model adheres to this rule in terms of containing at least the minimal number of executions and at most the maximal number of executions of tasks representing this activity, respectively. Rule R1 as defined above is an example of such a rule, defining an upper bound of 1 for the execution of activity *Aa*.
- Precedence and response rules: A precedence rule is derived for two activities, if in any complete execution sequence of the process model, an execution of a task representing the second activity is preceded by at least one execution of a task representing the first activity. A response rule is the counterpart of a precedence rule. Defined for two activities, it requires that any execution of a task modelling the first activity be eventually followed by an execution of a task representing the second activity in any complete execution sequence. Rule R2 is a combination of a precedence rule, requesting that applications are submitted (*As*) before they can be accepted (*Aa*), and a response rule, enforcing that an offer is selected and sent (*Sso*) afterwards.
- Ordering rules: Two activities may be executed independently of each other, but if they are executed for a single case, their executions are always ordered. In a process model, executions of the tasks representing these activities always occur in a particular order in any execution sequence. Rule R3 gives an example for such a rule that essentially prohibits the finalisation of an application (*Fa*) after the respective offer has been declined (*Do*).
- Exclusiveness rules: There may also be pairs of activities that are never jointly executed in any case, i.e., a process model does not contain an execution sequence

with tasks execution related to both activities. For those pairs of activities, an exclusiveness rule is derived. Rule R4 is an example for such an exclusiveness rule.

Once particular types of rules have been chosen, rules are instantiated for activities based on the tasks and their execution dependencies, as captured by a given process model. Aiming at a comprehensive assessment of conformance, rules are instantiated for all activities or pairs thereof, respectively, which are modelled by tasks. These rules are then used to check conformance of each individual trace of an event log: a specific rule may be satisfied or violated by the events of a trace.

Excursion 10

Processes as a set of rules

So far, we discussed rules as a means to extract some behavioural information from a process model. However, rules may also be used as a first class citizen in process modelling.

It has been argued that procedural models, as discussed in this book, are not well-suited to capture processes that are loosely structured and not entirely standardized. Examples for such processes can be found in domains where process control is done by people rather than systems, as it requires insights regarding the physical context of process execution and a balanced consideration of conflicting goals. Consider the clinical pathways of patients in a hospital: While there is a general ordering of examination and treatment activities, there is typically also some flexibility (e.g., whether to do a blood draw before or after an MRT screening), which enables medical staff members to react to clinical emergencies or the non-availability of particular resources. Yet, capturing all examination and treatment activities as an ad hoc subprocess, as discussed in Excursion 7, would not yield a truthful representation either, as certain general constraints on the execution of activities would be neglected.

Against this background, declarative process models have been suggested as an alternative modelling paradigm. Unlike procedural models that realize an *inside-out* approach of specifying all valid execution sequences explicitly, declarative models follow an *outside-in* approach: any sequence of task executions that is not explicitly forbidden is valid. A process model is then given as a set of rules, or constraints. Instead of explicitly modelling all valid cases of a process, any case that obeys these rules is considered to be valid.

DECLARE [79] is an example for a declarative process modelling language. It defines a set of templates for unary and binary rules for the execution of activities that are based on Linear Temporal Logic (LTL). Examples for such rules are: an activity may be executed at most n times, two activities must never be executed as part of a single case, or for each execution of one

(continued)

activity another activity needs to be executed at a later stage. To illustrate these rules, a simplified version of our loan application process is given as a DECLARE model in Figure 4.3.

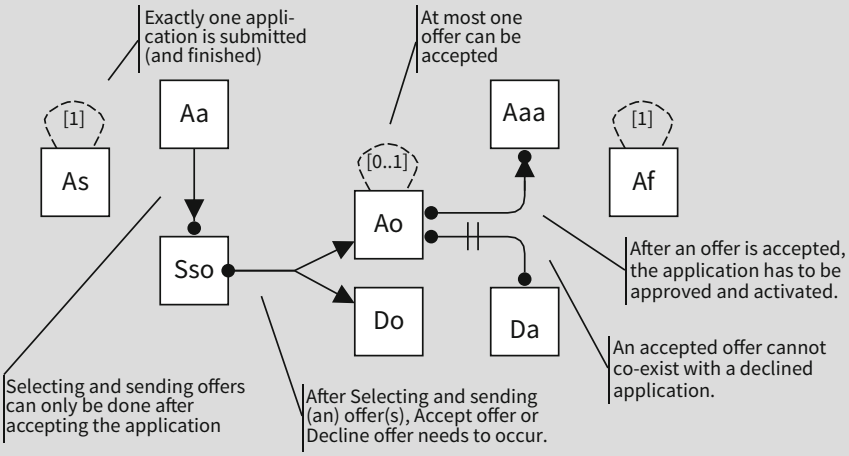


Figure 4.3 Adapted loan application process as a DECLARE model

Excursion 11

Computation of rules

Rules that constrain the execution of activities are derived from the execution dependencies defined between the respective tasks in a process model. A naive approach to compute these rules, therefore, is to explore all execution sequences of a process model and verify whether a particular rule holds true.

Taking up our example from Figure 4.2, for instance, to assess whether to use the rule that applications are declined (*Da*) at most once for conformance checking, all execution sequences of the process model are checked for more than one execution of the tasks that represent this activity. As discussed earlier, process models may have an infinite number of execution sequences, e.g., due to repetitive behaviour. However, even if so, they may only show a finite number of states that are reached by execution sequences. Then, conclusions on a particular rule can be drawn based on a representation of the states and state transitions of a process model, i.e., its state space. In fact, any execution sequence is a path in this state space. Checking whether an execution sequence can contain more than one task execution related to the activity of declining the application (*Da*), thus, becomes the check for the existence of a corresponding path in the state space. The latter is illustrated

(continued)

below for the process model of Figure 4.2. While there are two corresponding state transitions (referring to different tasks representing the activity), there is no path starting in the initial state that contains both transitions.

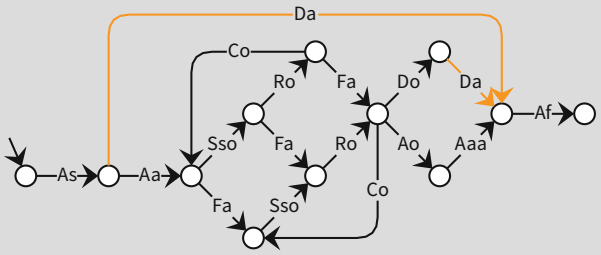


Figure 4.4 State space of the process model in Figure 4.2, with transitions that denote task executions related to activity *Da* being highlighted

Deriving rules from a state space of a process model, however, has to cope with the issue that the size of the state space may be exponential in the size of the process model—a phenomenon known as the state explosion problem. As visible already in Figure 4.4 for the part where an application is finalised (*Fa*) in parallel to selecting, sending, receiving, and potentially cancelling an order (*Sso*, *Ro*, *Co*), this blow-up of the number of states is caused by concurrent enabling of tasks.

Against this background, techniques have been developed to derive rules directly from the structure of a process model, without exploring its behaviour in terms of execution sequences (or states and state transitions). This is possible in particular for structured process models as discussed in Section 2.2. To give the intuition of such a derivation, consider the exclusiveness rule stating that an offer is either accepted (*Ao*) or declined (*Do*), but cannot be both accepted and declined. Exploring the model in Figure 4.2, we see that the two tasks capturing these activities are part of a model fragment representing an exclusive choice pattern, i.e., their nearest common predecessor in terms of the control flow arcs is an XOR split, whereas their nearest common successor is an XOR join (see Figure 2.5 for a visualization of this pattern). In such a situation, assuming that the exclusive choice pattern is not part of a loop pattern and that the XOR split can be enabled solely once, we can immediately conclude that the process model indeed induces an exclusiveness rule for the activities of accepting and declining an order. Such structural analysis is typically much more efficient than an exploration of all execution sequences of the process model.

4.1.2 Rule Checking by Example

Taking up the running example and some of the types of rules outlined above, rule-based conformance checking proceeds as follows. First, focusing on the number of possible activity executions, the cardinality rules derived for the example are given in Table 4.1. Here, we observe that the initial and final activities are required to be executed exactly once. The majority of activities, however, is not mandatory, but can optionally be executed in a case of the process. Except for three activities (*Sso*, *Ro*, *Co*) for which the corresponding tasks are part of a control flow loop in the process model, the activities of the process can be executed at most once, though.

The table further illustrates which of these rules are satisfied or violated for three exemplary traces of an event log. For trace $T_1 = \langle As, Aa, Sso, Ro, Ao, Aaa, Aaa \rangle$, two cardinality rules are violated: approval and activation of the application (*Aaa*) is done twice in the trace, whereas the model defines that this activity shall be executed at most once. Also, the last step defined by the process model indicating that the application has been finished (*Af*) is required to be recorded at least once in the trace, which does not hold true for trace T_1.

The remaining two traces, in turn, do not show any violation of a cardinality rule. In particular, the events signalling the repeated execution of the activity to receive an offer (*Ro*) in trace $T_2 = \langle As, Sso, Fa, Ro, Co, Ro, Aaa, Af \rangle$ are in line with the cardinality rules. It is interesting to note that, according to the process model in Figure 4.2, the number of executions of the activities to select and send an offer (*Sso*) and to receive an offer (*Ro*) shall be equivalent in any case of the process. Even though this does not hold true for trace T_2, this deviation is not detected as it is not reflected in the chosen formalism that captures cardinality rules solely in terms of a minimal and a maximal number of executions per activity.

Turning to the next type of rule, Table 4.2 illustrates the precedence rules derived for the running example and the result of verifying trace $T_2 = \langle As, Sso, Fa, Ro, Co, Ro, Aaa, Af \rangle$ against these rules. As defined by the model in Figure 4.2, for instance, there is a precedence rule that requires that, when an application is declined (*Da*), it must have been submitted (*As*) before. This rule is vacuously satisfied by trace T_2 as it does not contain any event related to declining

Table 4.1 Cardinality rules derived for the process model of the running example and their satisfaction (✓) and violation (✗) by exemplary log traces

Activity	***As***	***Da***	***Aa***	***Fa***	***Sso***	***Ro***	***Co***	***Ao***	***Aaa***	***Do***	***Af***
Cardinality:	**[1, 1]**	**[0, 1]**	**[0, 1]**	**[0, 1]**	**[0, *n*]**	**[0, *n*]**	**[0, *n*]**	**[0, 1]**	**[0, 1]**	**[0, 1]**	**[1, 1]**
$T_1 = \langle As,Aa,Sso,Ro,Ao,Aaa,Aaa \rangle$											
	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓	✗
$T_2 = \langle As,Sso,Fa,Ro,Co,Ro,Aaa,Af \rangle$											
	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
$T_3 = \langle As,Aa,Sso,Ro,Fa,Ao,Do,Da,Af \rangle$											
	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 4.2 Precedence rules derived for the process model of the running example and their satisfaction (✓) and violation (✗) by the exemplary log trace $T_2 = \langle As, Sso, Fa, Ro, Co, Ro, Aaa, Af \rangle$

	As	*Da*	*Aa*	*Fa*	*Sso*	*Ro*	*Co*	*Ao*	*Aaa*	*Do*	*Af*
As											
Da	✓										
Aa	✓										
Fa	✓		✗								
Sso	✓		✗								
Ro	✓		✗		✓						
Co	✓		✗		✓	✓					
Ao	✓		✓	✓	✓	✓					
Aaa	✓		✗	✓	✓	✓		✗			
Do	✓		✓	✓	✓	✓					
Af	✓										

an application (*Da*). The rule which enforces that approving and activating an application (*Aaa*) is preceded by the one to finalise an application (*Fa*) is an example of a non-vacuously satisfied rule. Yet, there are also several precedence rules that are violated. In particular, several rules require that the application was accepted (*Aa*) before the execution of another activity, such as receiving an offer (*Ro*) or cancelling an offer (*Co*). In addition, the lack of an acceptance of an offer (*Ao*) is highlighted by a violation of a precedence rule.

Even though it is not illustrated here, one can verify that traces T_1 and T_3 given in Table 4.1 satisfy all precedence rules derived for our running example.

In the same vein, response and ordering rules can be derived and verified for a given trace of a log. In the remainder, we limit the discussion to the exclusiveness rule as another example. The respective rules and their satisfaction or violation by trace $T_3 = \langle As, Aa, Sso, Ro, Fa, Ao, Do, Da, Af \rangle$ is illustrated in Table 4.3, which also highlights that the exclusiveness rules are symmetric. Note that most activities are exclusive with themselves, meaning that they cannot be executed more than once as part of a single case. An exception are the three activities that are part of a control flow loop (*Sso*, *Ro*, *Co*).

Most of the exclusiveness rules are satisfied by trace T_3. For instance, while the offer is declined (*Do*), the application is not approved and activated (*Aaa*). However, we observe also a few violations that stem from the acceptance of the offer (*Ao*), which is not in line with the exclusiveness rules related to the activities of declining the offer (*Do*) and the application (*Da*), for both of which the trace T_3 comprises corresponding events.

Finally, we note that traces T_1 and T_2 of Table 4.1 satisfy all exclusiveness rules derived for our running example.

Table 4.3 Exclusiveness rules derived for the process model of the running example and their satisfaction (✓) and violation (✗) by the exemplary log trace $T_3 = \langle As, Aa, Sso, Ro, Fa, Ao, Do, Da, Af \rangle$

	As	*Da*	*Aa*	*Fa*	*Sso*	*Ro*	*Co*	*Ao*	*Aaa*	*Do*	*Af*
As	✓										
Da		✓						✗	✓		
Aa			✓								
Fa				✓							
Sso											
Ro											
Co											
Ao		✗						✓		✗	
Aaa		✓							✓	✓	
Do								✗	✓	✓	
Af											✓

4.1.3 Conformance Feedback from Rule Checking

Rule checking focuses on the fitness dimension of conformance: By verifying whether the traces of a log satisfy the rules derived from a process model, it is assessed whether the model is well-suited to explain the recorded behaviour. Feedback on non-fitting behaviour then assumes the form of violations of rules by the traces.

First and foremost, the violation of rules can be interpreted locally, i.e., solely in the context of a particular trace. The set of violated rules identifies all activities, or pairs thereof for which the behaviour of the trace is not line with the process model. As such, rule checking provides fine-granular feedback that highlights which individual activities are involved in non-fitting behaviour. Referring to the three exemplary traces listed in Table 4.1 and the rule checking results in Tables 4.2 and 4.3, for instance, at most three of the 11 activities that are represented by tasks in the process model are identified as being involved in behavioural deviations for each of the three traces. For instance, for trace T_1, non-fitness is primarily related to the activity of approving and activating an application (*Aaa*) and the final step of having the application finished (*Af*), whereas for trace T_3, deviations involve the activities of accepting an offer (*Ao*), declining an application (*Da*), and declining an offer (*Do*).

Lifting the scope of conformance analysis from the level of individual traces to the event log as a whole, a global interpretation of the set of violated rules may provide further insights into ‘hot-spots’ of non-fitting behaviour. Activities that occur frequently in rules violated by a large number of different traces may point to systematic issues in how the process is conducted. For the running example, we note that the activity of accepting an offer (*Ao*) is identified as a source of non-fitness for both traces T_2 (violated precedence rule, see Table 4.2) and T_3 (violated

exclusiveness rule, see Table 4.3). As a consequence, one may wish to perform an audit and review the responsibility for this activity as well as the respective support by information systems.

Beyond feedback on individual activities that are involved in conformance violations, rule checking also enables a quantification of non-fitness for both aforementioned scopes: for an individual trace and a complete event log. Given a model M and an event log L, assume that R_M are the rules derived from M according to a formalism as detailed above. Then, for a single trace $\sigma \in L$ as well as the entire log L, a straightforward approach to quantify fitness is to compute the ratio of violated rules and all rules derived from the model:

$$\text{fitness}(\sigma, M) = \frac{|\{r \in R_M \mid r \text{ is satisfied by } \sigma\}|}{|R_M|} \tag{4.1}$$

$$\text{fitness}(L, M) = \frac{|\{r \in R_M \mid r \text{ is satisfied by all } \sigma \in L\}|}{|R_M|} \tag{4.2}$$

The above ratios seem to provide a reasonable means to quantify the extent to which a trace or an entire event log are explained by the model. The respective measures are normalised to the unit interval, while a fitness value of 1 means that all rules derived from the process model are satisfied. The opposite end of the respective fitness scale is more delicate, though. A fitness value of zero is obtained if all rules are violated. Depending on the formalism to specify rules and the specific set of rules instantiated for a model, this may be impossible. For instance, in our running example, any trace that violates the exclusiveness rule between the activities for accepting an offer (*Ao*) and declining an offer (*Do*) by their joint occurrence may implicitly satisfy rules on the cardinality on the minimal and maximal number of their execution (assuming the model and thus Table 4.1 would define a lower bound of zero and an upper bound of infinity for activities *Ao* and *Do*).

Reflecting on the above, it becomes apparent that the values obtained by the above measures depend on the chosen formalism to define rules. This is problematic as it assumes that the rules derived from a process model are complete and orthogonal. The former would ensure that any deviation between traces and a process model is taken into account. The latter would ensure that small behavioural deviations between traces and a process model affect a few rules, whereas large deviations impact a large number of rules. In the next section, we detail why the assumption of a formalism that is complete and comprises orthogonal rules is commonly violated.

4.1.4 Limitations of Rule Checking

Different types of rules detect different kinds of conformance issues or, more specifically, non-fitting behaviour. This was illustrated by means of examples in

Section 4.1.2. Taking the three traces $T_1 - T3$ given in Table 4.1, we observe that each of the three traces violates some rules of a particular type, while it satisfies all rules of other types.

The above observation hints at the major limitation of conformance checking based on rules: The analysis remains incomplete. Even if the full set of cardinality, precedence, response, ordering, and exclusiveness rules introduced above is exploited, there may be traces that satisfy *all* rules imposed by the process model, but are not fitting, i.e., they do not represent valid execution sequences of the model. An example for such a trace is $T_4 = \langle As, Aa, Sso, Ro, Ro, Co, Fa, Ao, Aaa, Af \rangle$. It is not valid according to the model in Figure 4.2, since an offer is received (*Ro*) two times in a row, whereas an offer is selected and sent (*Sso*) only once and not after an offer has been cancelled (*Co*). Yet, all of the rules introduced above are satisfied, so that the deviation from the model cannot be detected.

Clearly, the issue of incompleteness observed in the verification of the particular trace T_4 can be fixed by defining additional types of rules to be checked. However, in general, an exponential number of types of rules, in the number of activities, is needed to fully characterize the behaviour of process models captured in common modelling languages.

Extending the number of considered rule types further hinders reasonable quantification of non-fitness. By adding more expressive rules, redundancies are introduced by means of implication dependencies between rules. Already in the above formalism, rules are not fully orthogonal. For instance, defining a rule that requires the execution of two activities at least once implies that an exclusiveness rule cannot be defined for the activities at the same time. As a consequence, deviations in a trace may sometimes be reflected in a few, and sometimes in a large number of violated rules. Trying to overcome the issue of incompleteness by adding more expressive rules, therefore, introduces a bias into any assessment of the extent of non-conformance.

To conclude, conformance analysis based on rule checking may quickly become inefficient—an exponential number of rules need to be checked—and ineffective—the result of conformance checking is overwhelming to users due to its exponential size, whereas any quantification is biased by the chosen formalisms to specify rules.

4.2 A Gentle Introduction to Token Replay

Unlike rule checking that is grounded in information derived from the process model, token replay takes the event log as the starting point for conformance analysis. As indicated already by its name, this technique *replays* each trace of the event log in the process model by executing tasks according to the order of the respective events. By observing the states of the process model during the replay, it can be determined whether, and to what extent, the trace indeed corresponds to a valid execution sequence of the model.

4.2.1 Foundations of Token Replay

A process model defines states and state transitions, which encode the dependencies for the execution of tasks, such as a sequential ordering or repetitive behaviour. As sketched in Section 2.2.3, state transitions correspond to executions of tasks, so that, given an initial state of a process model, a representation of the set of all valid execution sequences can be constructed.

In essence, token replay postulates that each trace in the event log corresponds to a valid execution sequence of the process model. Whether, and to what extent, this is indeed true is verified by step-wise executing tasks of the process model according to the order of the respective events in the trace. During this replay, we may observe two cases that hint at non-conformance:

(i) the execution of a task requires the consumption of a token on the incoming arc, but the arc is not assigned any token in the current state, i.e., a token is *missing* during replay;
(ii) the execution of a task produces a token at an outgoing arc, but this token is not consumed eventually, i.e., a token is *remaining* after replay.

By exploring whether the replay of a trace yields missing or remaining tokens, replay-based conformance checking mainly focuses on the fitness dimension. That is, the ability of the model to explain the recorded behaviour is the primary concern. Traces are fitting if their replay does not yield any missing or remaining tokens, and non-fitting otherwise. In contrast to rule checking, precision can be estimated using token replay [72], but unfortunately, the corresponding technique strongly relies on the assumption that traces are fitting; if they are not, then the estimation of precision through token replay can be significantly degraded. Therefore, we refrain from introducing techniques for precision based on token replay due to the aforementioned weakness.

In order to check conformance of a single trace, it is replayed event by event in the process model. During replay, the state of the process model is adapted according to the execution semantics of the model. This idea of token replay is described in more detail as follows:

(1) Initialise the replay procedure by setting the *first event* of the trace as the *current event* of the replay.
(2) Locate the task of the activity for which execution is signalled by the *current event* of the trace in the process model.
(3) Replay the *current event* of the trace by executing the identified task in the *current state* of the process model by:
    - Consuming a token on the incoming arc of the respective task, even if there is none. If the arc was not assigned a token in the *current state*, record that a token was missing on the respective arc.
    - Producing a token on the outgoing arc of the respective task (if there is any).
(4) Set the state reached as the *current state* of the process model.

(5) If the *current event* is not the last in the trace, consider the next event in the trace as the *current event* and continue the replay from step 2 onwards.

Despite its simplicity, the above replay procedure is based on a subtle assumption. That is, in step 2, it is assumed that a task can unambiguously be located in the process model for the activity for which the execution is signalled by the current event of the trace. Intuitively, this means that there is no uncertainty on how to replay a single event of the trace in the process model. Towards the end of this section, we discuss the fact that this assumption is problematic in practice, since an event of the log may relate to multiple tasks in the model, or none at all. As a consequence, locating the task in the model for an event of the trace may be ambiguous or even impossible.

4.2.2 Token Replay by Example

We illustrate token replay with our running example and the trace $T_1 = \langle As, Aa, Sso, Ro, Ao, Aaa, Aaa \rangle$, used in Table 4.1 already to exemplify rule-based conformance checking. For this trace, the first event denotes that the application has been submitted (*As*). The respective task can indeed be executed in the initial state of the process model, as it is represented by the start event without incoming control flow arcs. The state reached afterwards assigns a token to the arc between the start event and the XOR-gateway; see Figure 4.5.

The next event to replay is the acceptance of the application (*Aa*). We note that in the BPMN model, the XOR-gateway representing the choice of declining or accepting an application may consume the token on its incoming arc and produce a token on one of its outgoing arcs. This transition between states of the process model is assumed to be implicit as the act of taking the decision is not reflected in the event log. Replay, therefore, has to take into account all possible implicit transitions from a particular state of the process model. As we discuss later, this requirement constitutes a major limitation of replay-based conformance checking.

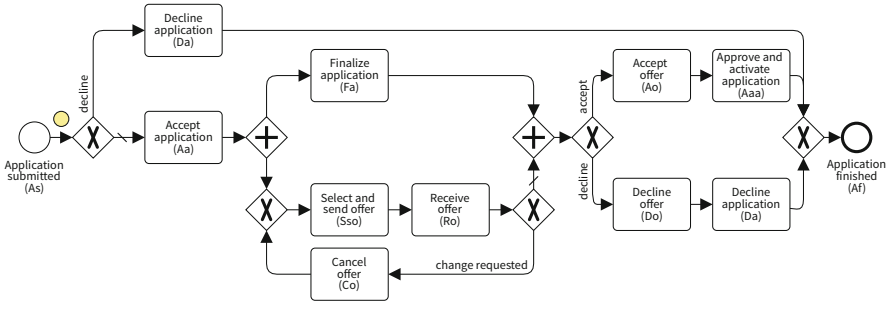


Figure 4.5 State reached in the loan application process model after replaying the first event of the trace

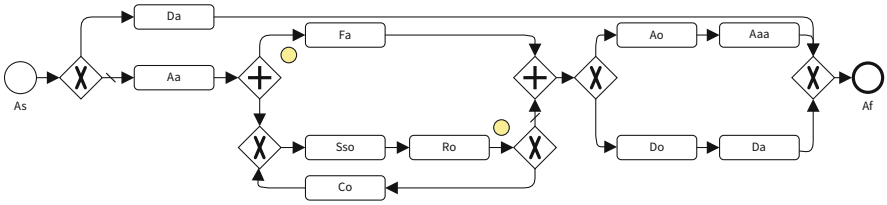


Figure 4.6 State reached after replaying the trace prefix $\langle As, Aa, Sso, Ro \rangle$

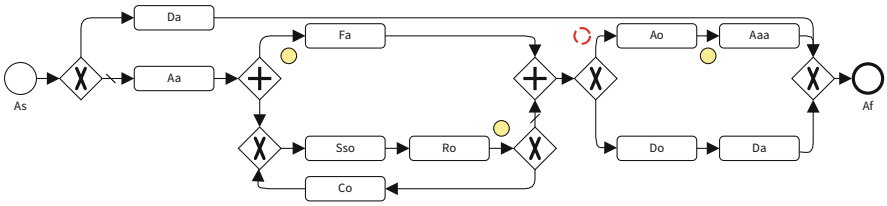


Figure 4.7 State reached after replaying the trace prefix $\langle As, Aa, Sso, Ro, Ao \rangle$

Assuming that the implicit transitions are taken as required to replay the trace, the event of accepting the application (*Aa*) can be replayed without any issue. The same holds true for the events of selecting and sending an offer (*Sso*) and receiving an offer (*Ro*). The state reached after replaying the prefix $\langle As, Aa, Sso, Ro \rangle$ of the considered trace is illustrated in Figure 4.6.

In the trace $T_1 = \langle As, Aa, Sso, Ro, Ao, Aaa, Aaa \rangle$, the next event to replay is the one of accepting the offer (*Ao*). However, in the state visualized in Figure 4.6, the incoming arc of the respective task is not assigned a token. Note that this is not a matter of implicit transitions. While the token on the outgoing arc of task *Ro* may pass the XOR-gateway (cross sign), it cannot pass the AND-gateway (plus sign), which requires both of its incoming arcs to carry a token.

Following the outlined replay procedure, however, the task modelling acceptance of an offer (*Ao*) is executed, which produces a token on its outgoing arc. Since the incoming arc is not assigned a token, a missing token is recorded, as illustrated in Figure 4.7.

When executing the next event to replay, i.e., approving and activating the application (*Aaa*), no issue is identified. As illustrated in Figure 4.7, the incoming arc of the corresponding task is assigned a token. Consuming this token and producing a token on the outgoing arc leads to the state visualized in Figure 4.8.

Finally, the last event of trace $T_1 = \langle As, Aa, Sso, Ro, Ao, Aaa, Aaa \rangle$ needs to be replayed, which, again, is the approval and activation of the application (*Aaa*). In the current state of replay, the incoming arc of the task is no longer assigned a token, so that another missing token is recorded. In addition, a token is produced, so that the outgoing arc is now assigned two tokens. With that, the replay of trace T_1 completes, yielding the state that is illustrated in Figure 4.9.

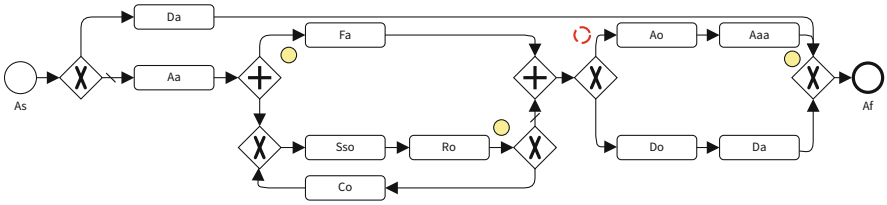


Figure 4.8 State reached after replaying the trace prefix $\langle As, Aa, Sso, Ro, Ao, Aaa \rangle$

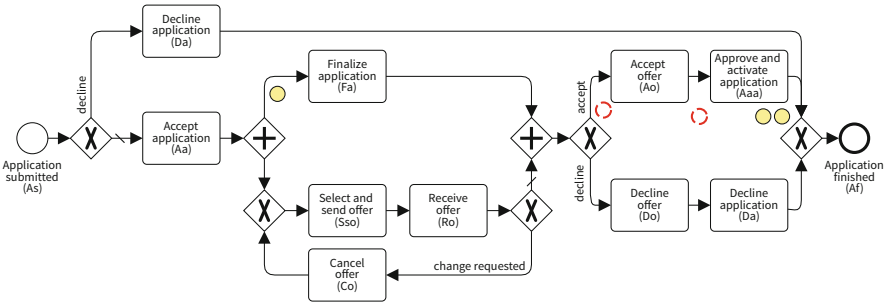


Figure 4.9 State reached after replaying the full trace $T_1 = \langle As, Aa, Sso, Ro, Ao, Aaa, Aaa \rangle$

From Figure 4.9, we conclude that the trace T_1 does not conform to the given process model. A trace that conforms to the process model should yield a state that does not assign any tokens to arcs. In the loan application process, the last activity should have been the one to finish the application (*Af*). In the process model, execution of the corresponding task (or rather BPMN end event) would solely consume, but not produce any tokens, thereby indicating that the final state of processing is reached. Yet, the replay of T_1 yields several remaining and missing tokens, which provide further hints on the type of conformance issues. The remaining token on the incoming arc of the task to finalise the application (*Fa*), for instance, hints at the lack of a necessary execution of this activity. Missing tokens, in turn, hint at executions of activities that were unexpected in the current state of the process, such as activities *Ao* and *Aaa* in the example.

We note that missing and remaining tokens often occur together. A non-conforming execution of an activity, such as the second execution of *Aaa*, may be unexpected in the current state of replay (missing token) and also unnecessary with respect to the completion of the process instance (remaining token).

4.2.3 Conformance Feedback from Token Replay

Token replay checks the fitness of an event log with respect to a given model. If a trace can be replayed without any missing or remaining tokens, it fits, i.e., the trace

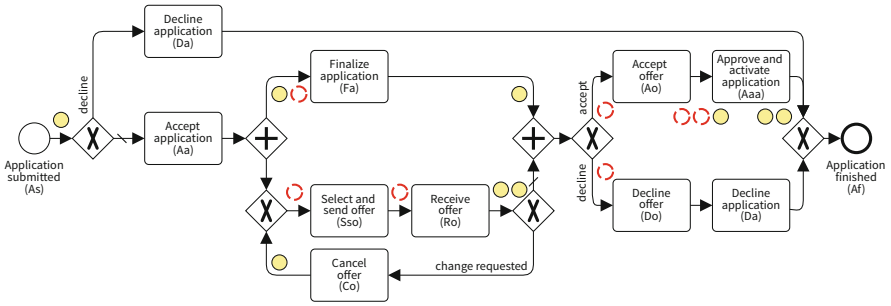


Figure 4.10 Missing and remaining tokens after replaying traces T_1 to T_3 of Table 4.1

corresponds to a valid execution sequence of the model. If all traces of an event log fit, the model is well-suited to explain the observed behaviour.

Feedback on non-conformance derived by token replay can be given on the level of an individual trace. Such local feedback comes in the form of the missing and remaining tokens after replaying the trace in the model, as illustrated for the example trace $T_1 = \langle As, Aa, Sso, Ro, Ao, Aaa, Aaa \rangle$ in Figure 4.9. Here, the tasks representing activities to finalise an application (*Fa*), to accept the offer (*Aa*), and to approve and activate the application (*Aaa*) are highlighted as the root cause of non-fitness.

Similarly to the approach illustrated for rule checking in Section 4.1.3, feedback on non-conformance can also be derived on a global level, taking into account all traces of an event log. For traces T_1 to T_3 given in Table 4.1, the aggregated result in terms of missing and remaining token is shown in Figure 4.10. Here, some activities appear to be more frequently involved than others in conformance violations. For example, the approval and activation of the application (*Aaa*) is clearly a 'hot-spot' of non-fitting behaviour in the event log. This insight helps to interpret the conformance checking results and may also trigger respective actions to avoid non-conformance in the future.

Feedback on conformance issues related to particular activities can be complemented by a quantification of non-fitness, on the level of an individual trace or the event log as a whole. However, simply using the counts of missing and remaining tokens recorded during replay may be misleading as the resulting measure would not be normalised and independent of the length of the replayed trace. Therefore, a common approach to quantify fitness based on token replay is to set the number of missing and remaining tokens into relation to the overall number of consumed and produced tokens during the replay. This yields two ratios that denote the relative share of non-fitness in the replay of a trace or an event log, respectively.

Specifically, we assume that a model is given as M, while L is an event log. Replaying a trace $\sigma \in L$ in M yields four token counts: $consumed(\sigma, M)$, $produced(\sigma, M)$, $missing(\sigma, M)$, $remaining(\sigma, M)$, denoting the overall numbers of tokens consumed and produced during the replay, and the numbers of missing and remaining tokens, respectively. Based thereon, fitness based on token replay is

quantified for a trace or an event log:

$$\text{fitness}(\sigma, M) = \frac{1}{2}\left(1 - \frac{missing(\sigma, M)}{consumed(\sigma, M)}\right) + \frac{1}{2}\left(1 - \frac{remaining(\sigma, M)}{produced(\sigma, M)}\right) \tag{4.3}$$

$$\text{fitness}(L, M) = \frac{1}{2}\left(1 - \frac{\sum_{\sigma \in L} missing(\sigma, M)}{\sum_{\sigma \in L} consumed(\sigma, M)}\right) + \frac{1}{2}\left(1 - \frac{\sum_{\sigma \in L} remaining(\sigma, M)}{\sum_{\sigma \in L} produced(\sigma, M)}\right) \tag{4.4}$$

The above measures yield a fitness value of 1, if a trace (or all traces of an event log) can be replayed without missing or remaining tokens, i.e., if they correspond to valid execution sequences of the process model. A fitness value of zero, in turn, means that all tokens consumed and produced during the replay have also been missing and remaining, respectively. If that is possible at all depends on the assumptions imposed on the structure of the process model and the traces in the event log.

In any case, we note that feedback on non-fitness is provided by token replay solely for events of a trace that relate to activities that are represented by tasks in the process model. As detailed in Section 4.2.1, when introducing a general procedure for token replay, there is the fundamental assumption that tasks for the activities of events to replay can be located unambiguously in the process model. As a consequence, events of a trace that represent activities that are not captured in a process model are not considered in the above fitness measures. Below, we discuss that the assumed ability to, given an event, locate a corresponding task in the process model constitutes the major limitation of conformance checking based on token replay.

4.2.4 Limitations of Token Replay

Token replay is based on an assessment of the states of the process model when replaying the events of a trace. This involves locating the tasks in the process model that correspond to the activities for which the events of a trace signal an execution; see Section 4.2.1. As such, events related to activities that are not captured in a process model cannot be considered during replay. Technically, such an event may simply be skipped, so that replay continues with the next event of the trace. Practically, however, this means that the process model does not explain the behaviour recorded in the event log—the log does not fit the model.

Beyond this obvious issue, locating a task in the process model for an activity imposes also more subtle challenges that limit the utility of replay-based conformance checking. In particular, replay may become non-deterministic for two reasons: (1) it may be impossible to unambiguously locate the task to execute when replaying an event; and (2) it may be impossible to unambiguously characterize the state reached after replaying an event.

The first situation may occur when a process model defines several tasks that represent the same activity. Considering the process model of our running example, e.g., in Figure 4.5, two tasks refer to the application being declined (*Da*), either immediately after the application has been submitted (*As*) or once an offer has been processed, but eventually declined (*Do*). Replaying a trace that contains an event representing that the application has been declined, it is unclear which of the tasks in the process model shall be executed—one of them may be chosen in a non-deterministic manner. As a consequence, the result of conformance checking becomes non-deterministic, leading to different numbers and locations of missing and remaining tokens.

Any heuristic to pick among multiple tasks representing the same activities based on the current state of the process model does not solve this problem. Consider the trace $T_5 = \langle As, Da, Fa, Sso, Ro, Do, Af \rangle$ that shall be replayed in the model of our running example (e.g., Figure 4.5). In this trace, after the application has been submitted (*As*), the application is declined (*Da*). Choosing the upper one of the two respective tasks in the process model may seem preferred for the replay, as it does not immediately yield any missing token. In that case, however, replaying the next event in the trace by executing the task to accept the application (*Aa*), would yield a missing token. For this particular trace, choosing either of the two tasks in the process model during replay yields the same total number of missing tokens. In general, this is not true, though, and a heuristic choice among multiple options to replay an event does not necessarily lead to the smallest number of missing and remaining tokens for the complete trace.

Another issue is related to the encoding of the control flow in a process model. Replay may require several state transitions in the model that are not visible in a trace of an event log, because they only relate to control flow dependencies. Taking BPMN as an example, the state of a process model is an assignment of tokens to arcs and gateways represent state transitions that do not involve the execution of any activity. This is problematic for token replay. After an event has been replayed, there may actually be multiple states that qualify as the current state of replay, each potentially leading to different conformance results in terms of missing and remaining tokens.

Consider the state, visualized in Figure 4.5, reached in the loan application process model after replaying the event that the application has been submitted (*As*). In this state, a token is assigned to the arc between activity *As* and the XOR-gateway. However, the current state of replay could also be one of the states assumed after the first decision on the application, as represented by the XOR-gateway, has been taken. That is, the two states that assign a token to either of the outgoing arcs of the XOR-gateway would also be viable states, as the act of taking this decision is not expected to be visible in the recorded trace.

Again, one may decide to chose non-deterministically between the possible states. Yet, this decision has implications for the replay of a trace. Starting in the state that assigns a token to the arc between the XOR-gateway and task *Da* would mean that subsequent replay of an event related to task *Aa* yields a missing token (and potentially a remaining token, if there is no event related to task *Da* in the

trace). A non-deterministic choice between the different possible states, therefore introduces a strong bias in conformance checking and precludes any reasonable interpretation of the obtained results.

Both issues, selecting among tasks representing the same activity during replay and considering different possible states in the model, give rise to a search problem. That is, one could explore all possible options, striving for the *best* way to replay a trace, e.g., in terms of the smallest sum of missing and remaining tokens. As we outline in the remainder, generalizing this idea leads to the concept of alignments, which provide a comprehensive basis for conformance checking.

4.3 A Gentle Introduction to Alignments

As a third approach to conformance checking, this section gives an overview of alignments. Unlike the aforementioned approaches, alignments take a symmetric view on the relation between modelled and recorded behaviour. Specifically, they can be seen as an evolution of token replay. Instead of establishing a link between a trace and sequences of task executions in the model through replay, alignments directly connect a trace with a valid execution sequence. Alignments are the core of many further techniques for conformance checking, which is why the overview of alignments in this section is complemented by an in-depth discussion and formalisation of alignments in the second part of this book.

4.3.1 Foundations of Alignments

An alignment connects a trace of the event log with an execution sequence of the model. It is represented by a two-row matrix, where the first row consists of activities as their execution is signalled by the events of the trace and a special symbol $\gg$ (jointly denoted by e_i below). The second row consists of the activities that are captured by task executions of an execution sequence of the model and a special symbol $\gg$ (jointly denoted by a_i):

log trace	e_1	e_2	$\ldots$	e_n
execution sequence	a_1	a_2	$\ldots$	a_n

Each column in this matrix, a pair (e_i, a_i), is a *move* of the alignment, meaning that an alignment can also be understood as a sequence of moves. There are different types of such moves, each encoding a different situation that can be encountered when comparing modelled and recorded behaviour. We consider three types of moves, as follows:

- *Synchronous move*: A step in which the event of the trace and the task in the execution sequence correspond to each other, i.e., both the event and the executed

task refer to the same activity. Synchronous moves denote the expected situation that the recorded events in the trace are in line with the tasks of an execution sequence of the process model. In the above model, a synchronous move means that it holds $e_i = a_i$ and $e_i \neq \gg$ (and thus $a_i \neq \gg$).

- *Model move*: When a task and thus an activity should have been executed according to the model, but there is no related event in the trace, we refer to this situation as a model move. As such, the move represents a deviation between the trace and the execution sequence of the model in the sense that the execution of an activity has been skipped. In the above model, a model move is denoted by a pair (e_i, a_i) with $e_i = \gg$ and $a_i \neq \gg$.
- *Log move*: When an event in the trace indicates that an activity has been executed, even though it should not have been executed according to the model, the alignment contains a log move. Being the counterpart of a model move, a log move also represents a deviation in the sense of a superfluous execution of an activity. A log move is denoted by a pair (e_i, a_i) with $e_i \neq \gg$ and $a_i = \gg$.

The above model allows for the definition of further types of moves, i.e., those pairs (e_i, a_i) with either $e_i = a_i = \gg$ or $e_i \neq \gg$, $a_i \neq \gg$, and $e_i \neq a_i$. The former, however, is not meaningful from a conformance checking point of view, as neither the trace nor the execution sequence advances. The latter, in turn, could represent a situation where the event indicates the execution of an activity, whereas a task capturing another, similar activity was executed in the process model. Striving for an intuitive overview of alignments in this section, however, we exclude such considerations here and refer to the second part of the book for further details.

For illustration, we take up our running example (see Figure 4.2) and the traces used in the discussion of rule checking and token replay. Consider trace $T_1 = \langle As, Aa, Sso, Ro, Ao, Aaa, Aaa \rangle$ and the execution sequence $E_1 = \langle As, Aa, Sso, Ro, Fa, Ao, Aaa, Af \rangle$. Then, a possible alignment is given as follows:

log trace T_1	As	Aa	Sso	Ro	$\gg$	Ao	Aaa	Aaa	$\gg$
execution sequence E_1	As	Aa	Sso	Ro	Fa	Ao	Aaa	$\gg$	Af

This alignment comprises six synchronous moves, one log move, $(Aaa, \gg)$, and two model moves, $(\gg, Fa)$ and $(\gg, Af)$. The log move $(Aaa, \gg)$ indicates that the application had been approved and activated, even though this was not expected in the current state of processing (as this had just been done). The model move $(\gg, Fa)$ is the situation of the process model requiring that the application shall be finalised, which has not been done according to the trace. Furthermore, one can easily extract the original trace by projecting away the special symbol for skipping from the top row. Applying the projection to the bottom row yields the execution sequence of the model.

The above alignment is only *one* possible alignment, though. Even for the same trace and execution sequence, there are many more possible alignments, e.g., the following one:

log trace T_1	As	$\gg$	Aa	Sso	Ro	$\gg$	Ao	Aaa	Aaa	$\gg$
execution sequence E_1	As	Aa	$\gg$	Sso	Ro	Fa	Ao	Aaa	$\gg$	Af

This alignment contains five synchronous moves, two log moves, and three model moves. While it is a viable alignment for trace T_1 and execution sequence E_1, the log move $(Aa, \gg)$ and the model move $(\gg, Aa)$ are unnecessary and can be avoided. Therefore, conformance checking is based on so-called *optimal* alignments, i.e., alignments that link a trace with an execution sequence with as few deviations as possible.

Optimality of an alignment may simply be defined based on the number of model moves and log moves. However, in many applications, deviations are not equally severe among the activities of a process. In our example scenario, declining an application twice is not as severe as erroneously accepting an application. Therefore, in alignments, model moves and log moves are given a certain penalty through a cost function, assigning a cost to each move. Then, an optimal alignment minimizes the total cost of moves. In the remainder of this book, we commonly use solely the alignment, implicitly assuming that the alignment is optimal.

Regardless of how costs are assigned to model moves and log moves, there can be multiple alignments that are optimal, i.e., have the same minimal cost. For instance, in the first alignment of T_1 and E_1 shown above, there is a synchronous move (Aaa, Aaa) that is followed by a log move $(Aaa, \gg)$. However, one can also construct an alignment in which this log move is followed by the synchronous move. This alignment would have a different sequence of moves compared to the first one, yet the assigned cost would be the same. We reflect on the consequences of this phenomenon when discussing feedback based on alignments and, in more detail, in the second part of this book.

We conclude that alignments provide a fine-granular approach to conformance checking—detecting deviations on the level of individual events and task executions. In contrast to the aforementioned approaches based on rules and replay, they also avoid the respective completeness issues. If a trace in the event log perfectly fits the model, any (optimal) alignment does not show any deviations, i.e., contains only synchronous moves. This provides guarantees to the user that if no deviation is detected, the trace indeed corresponds to a valid execution sequence of the process model. If so, a process analyst can focus their attention on the actual deviations detected.

4.3.2 Alignment Computation by Example

Having introduced the notion of alignments in the previous section, we now turn to the computation of an (optimal) alignment for a given trace of an event log. This is far from trivial given that a trace can be aligned with any execution sequence of the process model, while for each of them, there is a large number of possible

moves that can be considered. In this section, we provide an intuitive idea behind the computation of alignments, based on a special type of model called a *synchronous product*.

A synchronous product model is a combination of the original process model and the trace. To this end, the trace is expressed as a sequential process model, such that each event is translated into an appropriate task. We refer to this representation of the trace as the *trace model*. In addition to the original process model and the trace model, the synchronous product further contains synchronous tasks. For each pair of tasks in the process model and trace model that refer to the same activity (i.e., that are equally labelled), a synchronous task is created.

Intuitively, the execution of a synchronous task represents the situation that the event in the trace indeed relates to the task execution in the execution sequence of the model. In other words, the execution of a synchronous task corresponds to a synchronous move in the construction of an alignment. Executing a task that originates from the original process model, in turn, represents a model move. As the mirrored situation, the execution of a task from the trace model corresponds to a log move.

For our running example and trace $T_1 = \langle As, Aa, Sso, Ro, Ao, Aaa, Aaa \rangle$, the synchronous product is shown in Figure 4.11. On the top, the original process model is depicted, here representing the possibility of model moves. The bottom is the sequential trace model that encodes possible log moves. The synchronous tasks are visualized in the middle and highlighted in grey colour. Their execution corresponds to simultaneous progress in the top and the bottom model.

Execution semantics of the synchronous product follow directly from the semantics of the original process model and the trace model. That is, tasks can be executed if their incoming control flow arc is assigned at least one token in the

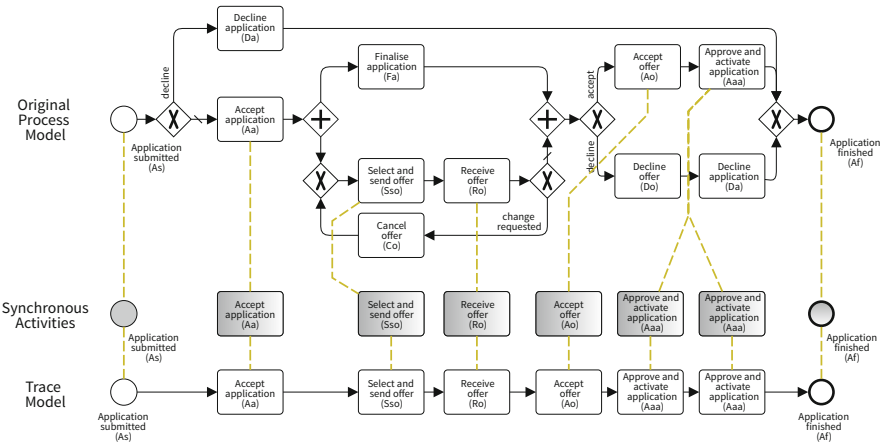


Figure 4.11 The synchronous product model for the running example and trace $T_1 = \langle As, Aa, Sso, Ro, Ao, Aaa, Aaa \rangle$

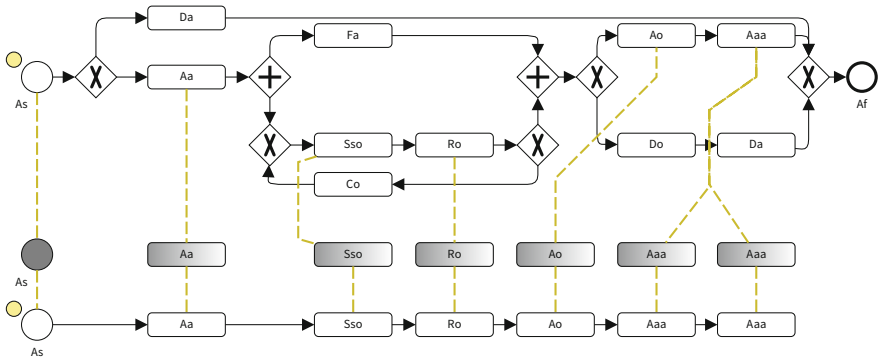


Figure 4.12 Synchronous product in its initial state

current state, while, upon execution, the state is changed by adding a token on the outgoing arc. Synchronous tasks represent the joint occurrence of a pair of tasks related to the same activity, one task originating from the original process model and one being part of the trace model. Following this line, a synchronous task can be executed if *both* these tasks have their incoming arc assigned at least one token, while its execution adds a token to outgoing arcs of *both* tasks.

For our example, the initial state of the synchronous product is shown in Figure 4.12. Here, the two tasks indicating that the application has been submitted (*As*) in the original process model as well as the trace model may be executed, separately of each other or jointly through the synchronous task.

The above definition of semantics induces the sets of all states and state transitions of the synchronous product, i.e., its state space. Here, the initial state is given by the union of the initial states of the original process model and the trace model. Similarly, the final state of the synchronous product is derived from the final states of the original process model and the trace model.

An optimal alignment is then constructed by traversing the state space of the synchronous product, until a shortest path is found from the initial state to the final state. Since each task in this model corresponds to a particular move, the aforementioned cost function for moves is used to define the distance between states in this space. Based on this distance, a shortest path is computed using well-established algorithms. When this path is found, it is translated into a alignment by simply considering each execution of a task as a model move, log move, or synchronous move, respectively.

Consider again the running example. From the initial state shown in Figure 4.12, we can reach the state visualized in Figure 4.13 by executing the synchronous tasks $\langle As, Aa, Sso, Ro \rangle$, highlighted in green colour in the figure. As discussed for BPMN process models in general, reaching that state involves several implicit state transitions due to the gateways stemming from the original process model.

In the state reached, however, we cannot execute a synchronous task. While the task representing the activity to accept an offer (*Ao*) could be executed in the

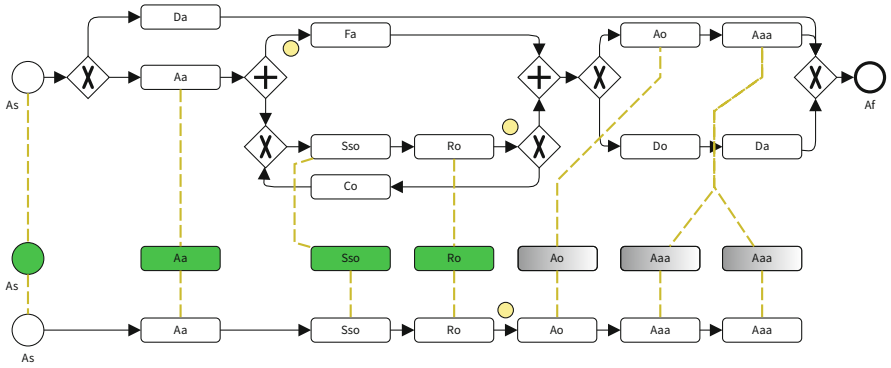


Figure 4.13 State of the synchronous product reached after executing the synchronous activities $\langle As, Aa, Sso, Ro \rangle$

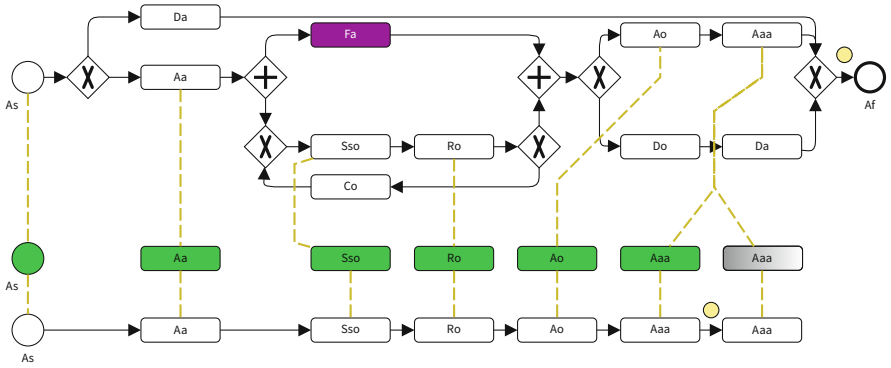


Figure 4.14 State of the synchronous product reached after executing the synchronous activities $\langle As, Aa, Sso, Ro \rangle$, followed by a model move related to activity *Fa* and the execution of two more synchronous activities $\langle Ao, Aaa \rangle$

trace model, it cannot be executed in the part representing the original process model. Rather, the task of finalising the application (*Fa*) can be executed in the original process model. This execution corresponds to a model move, a conformance deviation as the execution of the respective activity has not been recorded in the trace. Subsequently, two more synchronous tasks can be executed, *Ao* and *Aaa*, to reach the state shown in Figure 4.14.

In this state, again, there is no synchronous task that can be executed, while there are two options to proceed. Either the task (or rather BPMN end event) indicating that the application has been finished (*Af*) is executed in the part of the original process model, which would correspond to a model move; or the task to approve and activate the application is executed a second time, only in the trace model, thereby corresponding to a log move. Executing both, Figure 4.15 illustrates the final state that is reached. The execution sequence involving the coloured activities

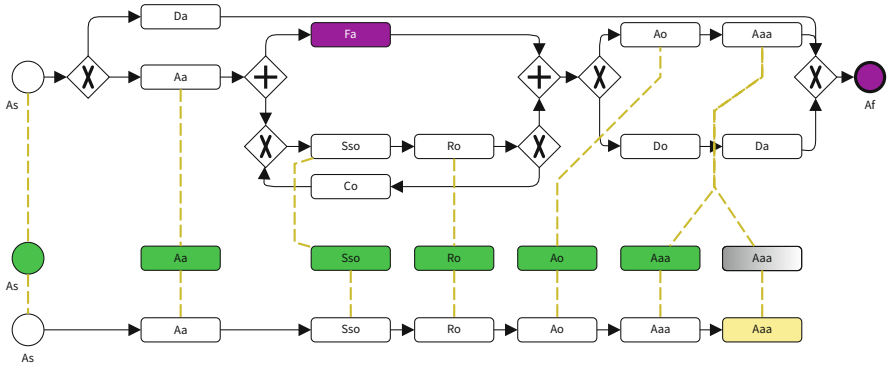


Figure 4.15 An optimal alignment, represented as an execution sequence of the synchronous product

in the synchronous product model corresponds to the alignment introduced earlier for the exemplary trace. In fact, it represents two alignments with equal cost, due to the different possible ordering of the last two task executions.

Evaluating all possible paths in the state space of the synchronous product model reveals that these alignments are indeed optimal. In addition, there are more optimal alignments of the same cost. As mentioned earlier the order of the synchronous move (Aaa, Aaa) and the log move $(Aaa, \gg)$ can also be swapped without changing the cost. Likewise, the model move $(\gg, Fa)$ can be moved forward in the alignment.

The method described here for finding an optimal alignment relies on the computation of the complete state space of the synchronous product. In practice, however, this state space may be huge or even infinite. In the second part of the book, we therefore revisit the computation of alignments. Based on a formalisation of the problem, we discuss methods for efficient alignment computation.

4.3.3 Conformance Feedback from Alignments

Alignments link a trace of an event log with an execution sequence of the process model, so that the accumulated cost of log moves and model moves is minimal. If the trace fits the model, it represents an execution sequence and the alignment contains solely synchronous moves. If all traces of the log can be aligned by synchronous moves only, the log as a whole fits the model, i.e. the model explains the recorded behaviour well.

If a trace is not fitting, the deviations are given by the model moves and log moves of the alignment. For illustration, we consider the traces T_1 to T_3 of our running example; see Table 4.1. While we discussed feedback on non-fitness already for trace T_1, also for traces T_2 and T_3 the optimal alignments have a cost larger than zero, i.e., they contain log moves or model moves:

log trace T_2	As	$\gg$	Sso	Fa	Ro	Co	$\gg$	Ro	$\gg$	Aaa	Af
execution sequence E_2	As	Aa	Sso	Fa	Ro	Co	Sso	Ro	Ao	Aaa	Af

The above alignment for trace T_2 highlights non-fitness that is related to the absence of recordings of expected executions of activities, e.g., to accept the application (Aa), select and send an offer (Sso), and accept an offer (Ao). However, as explained above, there may be a large number of optimal alignments for a given trace and process model. The presence of model moves and absence of log moves in the above alignment, therefore, needs to be interpreted with caution. Another optimal alignment for trace T_2 (assuming equal costs of all log moves and model moves) is the following, which also comprises a log move:

log trace T_2	As	$\gg$	Sso	Fa	Ro	Co	Ro	$\gg$	Aaa	Af
execution sequence E_4	As	Aa	Sso	Fa	Ro	$\gg$	Ro	Ao	Aaa	Af

A comprehensive way to give feedback on potential root causes of non-conformance, therefore, could consider *all* optimal alignments and assess which activities are often part of log moves and model moves. Intuitively, in the above example, the alignments differ with respect to the number of iterations of handling an offer (Sso, Ro, and Co). Yet, the alignments agree on the model moves related to activities Aa and Ao, providing evidence that the execution of these activities is indeed a major issue for this trace.

For trace T_3, however, such considerations are not needed, as there is a unique optimal alignment (assuming equal costs for all log moves and model moves):

log trace T_3	As	Aa	Sso	Ro	Fa	Ao	Do	Da	Af
execution sequence E_3	As	Aa	Sso	Ro	Fa	$\gg$	Do	Da	Af

As discussed for conformance checking based on rules and token replay, feedback on non-fitness can also be aggregated over all traces. That is, exploring all optimal alignments of all traces of a log reveals which activities are frequently part of log moves and model moves, so that actions to follow up on these deviations can be guided. Considering the three alignments illustrated above, for instance, the acceptance of an offer (Ao) appears to be major source of non-conformance.

Furthermore, alignments provide a straightforward means to quantify the observed deviation. Again, this may be done based on the level of an individual trace or the event log as a whole. In line with the fitness discussed above based on token replay, the aggregated cost of log moves and model moves may be a misleading measure, though, as it is not normalised. A common approach, therefore, is to normalise this cost by dividing it by the worst-case cost of a aligning the trace with the given model. Under a uniform assignment of costs to log and model moves, such a worst-case cost originates from an alignment in which each event of the trace relates to a log move, whereas all task executions of an execution sequence of the model relate to a model move and the execution sequence is as short as possible.

Put differently, the alignment does not contain any synchronous move:

log trace	e_1	e_2	...	e_n	$\gg$	$\gg$	...	$\gg$
execution sequence	$\gg$	$\gg$	...	$\gg$	a_1	a_2	...	a_m

Since the cost induced by the model moves of an execution sequence depends on its length, the shortest (and thus cheapest) possible execution sequence leading from the initial state to a final state in the model is considered for this purpose.

Realizing the above idea, we obtain two ratios that denote the relative share of non-fitness in the alignments of a trace or an event log, respectively. Let M be a model and L an event log. Then, we denote by $cost(\sigma, M)$ the cost of an optimal alignment of a trace $\sigma \in L$ with respect to the model. Furthermore, let $cost(\sigma, \langle\rangle)$ and $cost(\langle\rangle, x)$ be the costs of aligning a trace σ with an empty execution sequence, or some execution sequence $x \in M$ of the model with an empty trace, respectively. Then, fitness based on alignments is quantified for a trace or an event log:

$$\text{fitness}(\sigma, M) = 1 - \left(\frac{cost(\sigma, M)}{cost(\sigma, \langle\rangle) + \min_{x \in M} cost(\langle\rangle, x)} \right) \tag{4.5}$$

$$\text{fitness}(L, M) = 1 - \left(\frac{\sum_{\sigma \in L} cost(\sigma, M)}{\sum_{\sigma \in L} (cost(\sigma, \langle\rangle)) + |L| \times \min_{x \in M} cost(\langle\rangle, x)} \right) \tag{4.6}$$

Interpretation of these measures is similar to those introduced for the other approaches to conformance checking: A fitness value of 1 is obtained if a trace (or all traces of an event log) can be aligned by synchronous moves only, i.e., if they correspond to execution sequences of the process model. A fitness value of zero, in turn, means that any event of the trace is aligned by a log move, whereas any task execution of the shortest execution sequence of the model is aligned by a model move. This is only a theoretical edge case, though. For any trace that contains an event related to a task that can be executed in some execution sequence of the process model, an optimal alignment would contain at least one synchronous move and have a fitness score larger than zero.

Due to their symmetric view on traces and execution sequences, alignments enable an assessment not only of fitness, but also of the precision of a process model with respect to an event log. Execution sequences in the model that are never part of an alignment represent behaviour that goes beyond what has been recorded in the event log. Below, we summarize the main idea behind a simple operationalisation of precision measures based on alignments.

Precision measurement based on alignments is grounded in the general idea of escaping edges as outlined already in Section 3.5. To give the intuition of the operationalisation of this approach based on alignments, we assume that (1) the event log fits the model; and (2) that the model is deterministic. The former means that we simply exclude non-fitting traces, for which the optimal alignment contains log moves or model moves, from the assessment of the precision of the model. The

latter refers to a process model not being able to reach a state, in which two tasks that capture the same activity of the process are enabled. The model of our running example (see Figure 4.2) is deterministic.

For the activity of each event of a trace of the event log, we can determine a state of the process model right before the respective task would be executed. Under the above assumptions, this state is uniquely characterized. What is relevant when assessing precision, is the number of tasks enabled in this state of the process model. Let M be a process model and L an event log, with $\sigma \in L$ as a trace and, overloading notation, $e \in \sigma$ as one of the events of the trace. Then, by $enabled_M(e)$, we denote the number of tasks and, due to determinism of the model also the number of activities, that can be executed in the state right before executing the task corresponding to e.

Similarly, we consider all traces of the log that also contain events related to the activity of event e, say a, and have the same prefix, i.e., events that indicate that the same sequence of activities has been executed before an event signalling the execution of activity a. Then, we determine the number of activities for which events signal the execution directly after this prefix, i.e., the set of activities that have been executed in the same context as the activity a as indicated by event e. Let this number of activities be denoted by $enabled_L(e)$, which, under the above assumptions, is necessarily less than or equal to $enabled_M(e)$. Then, the ratio of both numbers captures the amount of escaping edges that represent modelled behaviour that has not been recorded. As such, precision of log L and model M is quantified as follows:

$$\text{precision}(L, M) = \frac{\sum_{\sigma \in L, e \in \sigma} enabled_L(e)}{\sum_{\sigma \in L, e \in \sigma} enabled_M(e)} \tag{4.7}$$

The above notions provide only the intuition of how to conduct conformance checking based on alignments. However, such alignments are fundamental to many more aspects of conformance checking, which are reviewed in the second part of the book.

Bibliographic Notes

Rule-based conformance checking, exploiting constraints as formalised by binary rules for the order of potential occurrences of tasks in process models, has been introduced in [132]. It also includes the definition of different versions of fitness measures, essentially incorporating various strategies to normalise the measures. The general approach, however, may also be instantiated with different sets of rules, such as transition adjacency relations [137], causal behavioural profiles [133], or the relations of the 4C spectrum [82].

Techniques for token replay were first introduced in [93]. Alternative techniques were presented in [126], and later adapted to an online scenario in [125].

Alignments and cost-based fitness checking were introduced in [1] and corresponding publications [5, 115]. Precision checking was initially proposed in [72], and later adapted to be computed on top of alignments in [3, 4].

4.4 Exercises

4.A) Derivation of rules

Consider the model of a bug fix process in Figure 4.16 (partially known already from Chapter 2). Derive the cardinality rules, precedence rules, and exclusiveness rules for this model.

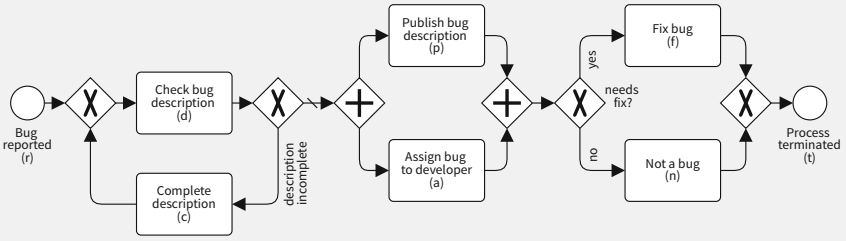


Figure 4.16 A simple bug fix process

4.B) Rule checking

Given the rules derived in Exercise 4.A, assess the conformance of the following traces based on rule checking.

- $\langle r, d, c, p, a, t \rangle$
- $\langle r, d, c, d, a, p, f, t \rangle$
- $\langle r, d, d, p, f, n \rangle$

Which rules are violated, which of them are satisfied?

4.C) Fitness and feedback based on rule checking

Using the results of rule checking as done in Exercise 4.B, compute the fitness values of the three respective traces. Also, do the traces correspond to an execution sequence of the model in Figure 4.16? If not, what is the feedback given on non-conformance?

4.D) Token replay

Consider again the process model of a bug fix process in Figure 4.16. Replay the following trace: $\langle r, d, c, p, a, t \rangle$. Why is this replay challenging, e.g., once the prefix $\langle r, d \rangle$ has been replayed? Also, how many missing and remaining tokens are recorded during replay of this trace, and where ?

4.E) Fitness and feedback based on token replay

Using the process model of Figure 4.16, replay the following traces (known from Exercise 4.B):

- $\langle r, d, c, p, a, t \rangle$
- $\langle r, d, c, d, a, p, f, t \rangle$
- $\langle r, d, d, p, f, n \rangle$

Compute the fitness values of the three traces. What is the feedback given on non-conformance, if a trace does not fit the model?

4.F) Alignment computation

Consider again the process model in Figure 4.16 and the following three alignments for the trace $\langle r, d, c, p, a, t \rangle$:

Alignment 1:

log trace	r	d	c	p	a	$\gg$	t
execution sequence	r	d	$\gg$	p	a	n	t

Alignment 2:

log trace	r	d	c	$\gg$	p	a	$\gg$	t
execution sequence	r	d	$\gg$	a	p	$\gg$	f	t

Alignment 3:

log trace	r	d	c	$\gg$	p	a	$\gg$	t
execution sequence	r	d	c	d	p	a	f	t

Assuming a unit cost of one for log moves and model moves: Which of the alignments is optimal? Are there further optimal alignments for the given model and traces?

4.G) Fitness and feedback based on alignments

Consider again the process model in Figure 4.16 and the three traces from Exercise 4.B. Compute the fitness of each trace using alignments. Also, what is the feedback given on non-conformance, if a trace does not fit the model?

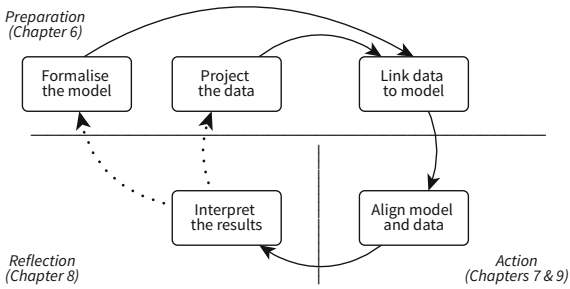
4) In the lab: tool support for fitness analysis

Check out the lab session to practise with tools that allow analysing fitness of a process model with respect to an event log.

http://www.conformancechecking.com/CC_book_Chapter_4

Part II
A Deep Dive into Conformance Checking

Preface to the Second Part



Lifecycle of conformance checking projects

This second part of the book provides a detailed picture of all the necessary elements for applying conformance checking in real-life projects for which Chapter 5 provides the mathematical foundation. The figure above shows the necessary steps in a conformance checking project. First, the *preparation* phase is started, which we cover in Chapter 6. For conformance checking, a formal representation of the model is required, so we often need to translate a process model that might only exist in an informal diagram or flow chart, to a model formalism with clear execution semantics (e.g., Petri nets). Simultaneously, we need to project the data that we are given to the notion of an event log, which records distinguishable traces of unique cases. Events from different granularities need to be collected and extracted from potentially multiple sources. Once we have formalised the model and projected the data, we need to link these two artefacts. The link is necessary to relate the two representations of activity executions to each other: It establishes which events in the data record which executions of tasks of the model. This linking can be trivial,

if the model is already on the same level of granularity as the data, and even the labels match (i.e., there is a 1:1 relation of events and task executions). However, it can also be difficult, when the relation between events and task executions is 1:n, or even n:m.

The *action* phase captures the algorithmic details of getting an optimal alignment between traces of an event log and a process model. We cover a basic cost-based approach to derive an alignment in Chapter 7, while more advanced techniques are covered in Chapter 9.

The *reflection* phase targets interpretation of alignment results. Chapter 8 covers this aspect and exemplifies the kind of insights that we can extract from alignments. As the flow indicates, there is a link back to the start of the conformance checking cycle. The results of one iteration might be not be in line with the expectations, so that, in the reflection phase, potential errors in the projection, formalisation or linking of model and data can be detected.

Chapter 5
Preliminaries to Conformance Checking

Next, we formally describe notations that help us to unambiguously define the concepts and methods of conformance checking. We start with simple mathematical notions and visit relevant graph-theoretic concepts. At the end of this chapter, we describe linear programming problems.

Sets, Multisets and Operations A *set* X is a collection of distinct objects, called *elements* or *members*. We use $x \in X$ to express that x is a member of the set X. Two ways of describing a set X are introduced:

- Exhaustive list of its elements: $X = \{a, b, c\}$, $X = \{x_1, \ldots, x_n\}$.
- Members of the set must satisfy some condition. The format is $X = \{x \mid x$ satisfies condition $\Pi\}$ where the condition can be expressed either in natural language or as some predicate in first-order logic.

Moreover, some symbols are used to denote universal sets:

- the set of real numbers, $\mathbb{R}$
- the set of integers, $\mathbb{Z}$
- the set of non-negative integers, $\mathbb{N}$
- the set of binary numbers, $\mathbb{B} = \{0, 1\}$

Given two sets X and Y, $X = Y$ and $X \neq Y$ denote equality and inequality of the sets X and Y, respectively. We write $X \subseteq Y$ to denote that X is a subset of Y, i.e.,

$$X \subseteq Y \iff \forall x \in X : x \in Y$$

and $X \subset Y$ to denote that X is a proper subset of Y, i.e., $X \subseteq Y$ and $X \neq Y$. $X \setminus Y$ denotes the set of members of X that are not members of Y.

The *powerset* of a set X is the set of all its subsets, i.e., $\wp(X) = \{Y \mid Y \subseteq X\}$. The *Cartesian product* of X and Y, denoted by $X \times Y$, is defined by

$$X \times Y = \{(x, y) \mid x \in X \wedge y \in Y\}$$

J. Carmona et al., *Conformance Checking*,
https://doi.org/10.1007/978-3-319-99414-7_5

and the operator can be defined for any given (finite) arity. The Cartesian product of $X_1, \ldots, X_n$ is defined by

$$X_1 \times \ldots \times X_n = \{(x_1, \ldots, x_n) \mid x_1 \in X_1 \wedge \ldots \wedge x_n \in X_n\}.$$

When all the sets are the same, the Cartesian product can be abbreviated as X^n. For instance, $X \times X \times X$ is X^3. Moreover, in this situation, more than one dimension can be defined in the Cartesian product:

$$X^{n \times m} = \{((x_{1,1}, \ldots, x_{1,m}), \ldots, (x_{n,1}, \ldots, x_{n,m})) \mid x_{i,j} \in X\}.$$

An element $(x_1, \ldots, x_n) \in X_1 \times \ldots \times X_n$ of an n-ary Cartesian product is a tuple of n elements of the respective sets. We write $\pi_i((x_1, \ldots, x_n)) = x_i$ for $1 \leq i \leq n$ to denote the projection on the i-th component of such a tuple. For instance for $(x_1, x_2, x_3) \in X_1 \times X_2 \times X_3$, $\pi_1((x_1, x_2, x_3)) = x_1$, and $\pi_3((x_1, x_2, x_3)) = x_3$.

Given a set X, a multiset M of X is a mapping $M : X \to \mathbb{N}$. $\mathcal{B}(X)$ denotes the set of all multisets over X. Multisets can be represented in vector format, i.e., $[x^3, y, z^2]$ is the multiset that has three occurrences of x, one of y and two occurrences of z. We write $M_1 \leq M_2$, if $M_1(x) \leq M_2(x)$ for all $x \in X$. For example, $[y, z] \leq [x^3, y, z^2]$ and $[y^2] \not\leq [x^3, y, z^2]$. The difference $(M_1 - M_2)$ and union $(M_1 + M_2)$ are defined as usual. For example, $[x^3, y, z^2] - [y, z] = [x^3, z]$.

In the remainder, with a slight abuse of notation, we adopt some concepts defined only for sets and interpret them in the context of multisets. This includes, in particular, containment and summation: Given a multiset M of X, we write $x \in M$, if $M(x) > 0$. In addition, given some function $f : X \to \mathbb{R}$, we write $\sum_{x \in M} f(x)$ as a short-hand for $\sum_{x \in \{y \in X \mid M(x) > 0\}} M(x) f(x)$.

In a similar manner, we also adopt the operators, $\leq$, $+$, and $-$ defined for multisets and use them for pairs of sets and multisets: Given a multiset M of X and a set $Y = \{y_1, \ldots, y_n\}$, the operations $M \leq Y$, $M + Y$, and $M - Y$ are interpreted as $M \leq M_Y$, $M + M_Y$, and $M - M_Y$ with M_Y being a multiset of Y defined as $M_Y = [y_1, \ldots, y_n]$.

Vectors and Matrices A column vector $\vec{v}$ of dimension n over one of the universal sets $X \in \{\mathbb{R}, \mathbb{Z}, \mathbb{N}, \mathbb{B}\}$ is an element from X^n, i.e., $\vec{v} \in X^n$. It is denoted by $\begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix}$.

For instance, $\vec{w} = \begin{bmatrix} 3 \\ 0.5 \\ 5.2 \end{bmatrix}$ is a column vector of dimension three over $\mathbb{R}$, i.e. $\vec{w} \in \mathbb{R}^3$.

With $\vec{v}^{\,\intercal}$ we denote the transpose of a vector, i.e. $\vec{w}^{\,\intercal} = [3, 0.5, 5.2]$ is the transpose of $\vec{w}$. Given two vectors $\vec{x}$ and $\vec{y}$ of dimension n, $\vec{x}^{\,\intercal} \cdot \vec{y}$ denotes the product of the two vectors, defined by

$$\vec{x}^{\,\intercal} \cdot \vec{y} = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n.$$

A matrix $\mathbf{C}$ of dimension $n \times m$ over a universal set X is an element from $X^{n \times m}$. The operations $\vec{w} \cdot \mathbf{C}$ and $\mathbf{C} \cdot \vec{w}$ denote the left and right products of matrix $\mathbf{C}$ and the vectors $\vec{w} \in X^n$ and $\vec{w} \in X^m$, only defined if the dimensions agree. The symbol $\vec{0}$ denotes a vector such that every component is 0. When a vector is defined over a set X, the set X is implicitly assumed to be totally ordered and for all $x \in X$, $\vec{1}_x$ denotes the vector with all elements 0, except the element at the index corresponding to x, which equals 1. If a function $f : X \to Y$ is defined on X, then f can be applied to a vector or matrix over X by applying it to all elements of X individually.

Sequences and Their Parikh Representation Let Σ be a set, called *alphabet*. Σ^* is the set of all sequences over Σ. A finite sequence (of length n) over Σ is a mapping $\{1, \ldots, n\} \to \Sigma$. We represent a finite sequence $\sigma : \{1, \ldots, n\} \to \Sigma$ as $\langle x_1, x_2, \ldots, x_n \rangle$, where $x_i = \sigma(i)$, for $1 \leq i \leq n$. The length of a sequence is denoted by $|\sigma|$ and we write $\sigma \in \Sigma^*$. The empty sequence with length 0 is defined as $\langle\rangle$. For instance, if $\Sigma = \{a, b, c, d\}$, a possible sequence of length 5 over Σ is $\sigma = \langle a, a, a, c, d \rangle$.

If $\sigma = \langle x_1, x_2, \ldots, x_n \rangle$ and $\gamma = \langle y_1, y_2, \ldots, y_m \rangle$ are finite sequences then the concatenation of σ and γ, denoted by $\sigma \cdot \gamma$, is the sequence $\langle x_1, x_2, \ldots, x_n, y_1, y_2, \ldots, y_m \rangle$ of length $n + m$. For any sequence σ, the concatenation with the empty sequence $\langle\rangle$ does not change it, that is, $\sigma \cdot \langle\rangle = \sigma$.

The operator $|\sigma|_x$ denotes the number of occurrences of x in σ. For instance, if $\sigma = \langle a, d, a, c, a \rangle$, $|\sigma|_a = 3$, $|\sigma|_c = 1$, and $|\sigma|_b = 0$.

The projection of sequence σ into set X, denoted by $\sigma|_X$, is a new sequence obtained by removing from σ those elements not belonging to X. For instance, for $\sigma = \langle a, d, c, a, c \rangle$, we have $\sigma|_{\{d,c\}} = \langle d, c, c \rangle$.

A sequence σ is said to be a (finite) prefix of a sequence γ if there exists a sequence θ such that $\gamma = \sigma \cdot \theta$.

Given an alphabet $\Sigma = \{a_1, \ldots, a_n\}$, the Parikh vector of a sequence over this alphabet is a function $\vec{\cdot} : \Sigma^* \to \mathbb{N}^{|\Sigma|}$ resulting in a column vector, such that $\vec{\sigma}^\intercal = [|\sigma|_{a_1}, \ldots, |\sigma|_{a_n}]$.[1]

If a function $f : \Sigma \to Y$ is defined over Σ, then f can be applied to a sequence over Σ by applying it to all elements of the sequence individually.

Graphs and Their Representations A *graph* is a pair (V, E) where V is the set of nodes or vertices, and E is a set of edges between nodes. If the edges are undirected, then the graph itself is called undirected. Otherwise, the graph is directed. In this book we focus on directed graphs that have labels on the edges. A *labelled directed graph* is a tuple $DG = (V, E, L)$ where V is the set of nodes, L is the set of labels, and $E \subseteq V \times L \times V$ is the set of labelled edges.

Given a labelled directed graph $DG = (V, E, L)$ and two nodes $v, v' \in V$, a path from v to v' is a sequence of edges $\sigma = \langle e_1 \ldots e_{|\sigma|} \rangle \in E^*$, where $\sigma = \langle\rangle$ if

[1] Please note that, when using Parikh vectors, we assume the alphabet to be totally ordered and the vector to respect that ordering.

$v = v'$, and $\sigma \neq \langle\rangle \implies \pi_1(\sigma(1)) = v, \pi_3(\sigma(|\sigma|)) = v'$, and for all $1 \leq i < |\sigma| : \pi_3(\sigma(i)) = \pi_1(\sigma(i+1))$.

Linear Programming A *linear inequality* or *constraint* is given by a vector $\vec{a} \in \mathbb{R}^n$, a vector of variables $\vec{a}$, and a real value b. It is represented by

$$\vec{a}^{\intercal} \cdot \vec{x} \leq b$$

and it is *feasible* if there exists some assignment $\vec{k} \in \mathbb{R}^n$ to $\vec{x}$ satisfying $\vec{a}^{\intercal} \cdot \vec{k} \leq b$.

A *system of linear inequalities* is a set of linear inequalities. It is *feasible* if there exists a vector that satisfies all inequalities of the set. If it is finite then it has a matrix-based representation

$$\mathbf{A} \cdot \vec{x} \leq \vec{b}$$

where the vectors $\vec{a}$ of the linear inequalities are the rows of the matrix $\mathbf{A}$ and the numbers b are the components of the vector $\vec{b}$.

A *linear programming problem* (LP) is a system $\mathbf{A} \cdot \vec{x} \leq \vec{b}$ of linear inequalities, and optionally a linear function $\vec{c}^{\intercal} \cdot \vec{x}$ called the *objective function*. A solution of the problem is a vector of rational numbers that satisfies the constraints. A solution is *optimal* if it minimizes the value of the objective function (over the set of all solutions). An LP is *feasible* if it has a solution.

An *integer linear programming problem* (ILP) is an LP where the set of solutions is given by vectors of integers only, i.e. it is *feasible* only if there exists some assignment $\vec{k} \in \mathbb{Z}^n$ to $\vec{x}$ satisfying $\mathbf{A} \cdot \vec{k} \leq \vec{b}$.

The complexity of solving a linear problem depends on the domain under consideration. Specifically, it is known [98] that:

(1) Each linear programming problem (LP) over $\mathbb{R}$ can be solved in polynomial time.
(2) The solubility of integer linear programming problems (ILP) is NP-complete.

Chapter 6
Preparation

To prepare the artefacts needed by conformance checking algorithms, we first are required to clarify the different representations of a process: its recorded behaviour in terms of an event log and its modelled behaviour as captured by a process model. Figure 6.1 revisits the terminology of conformance checking and also points to the subsections, in which we discuss the respective concepts. In Section 6.1, we consider the notions of events, traces, and event logs. In Section 6.2, in turn, we formally describe models that are used in conformance checking. Finally, in Section 6.3, we discuss that recorded behaviour in terms of events may not directly be of the form required for conformance checking. Rather, events need to be related to traces (and thus cases and execution sequences) and activity executions (and thus task executions) explicitly.

6.1 Processes in Action: Event Data

Processes of an organization are often complex by nature. Their complexity is assessed along various dimensions: The processes can comprise a large number of activities. They can have many decision points. There can be many participants working collaboratively on a case. Also, there can be many work streams running in parallel, possibly with the support of multiple information systems and also external suppliers. But regardless of the particular complexities of a process, many processes have one thing in common: They generate data. We call the data generated by a process *event data*.

Collecting the data that is generated by a process opens various opportunities for improving the process. It enables conclusions on whether we do the right things, whether we do them in the correct order, and also whether we do them timely. Yet, to quantify process improvements, we need to be able to measure how the recorded behaviour of a process relates to its modelled representation. To this end, event data

J. Carmona et al., *Conformance Checking*,
https://doi.org/10.1007/978-3-319-99414-7_6

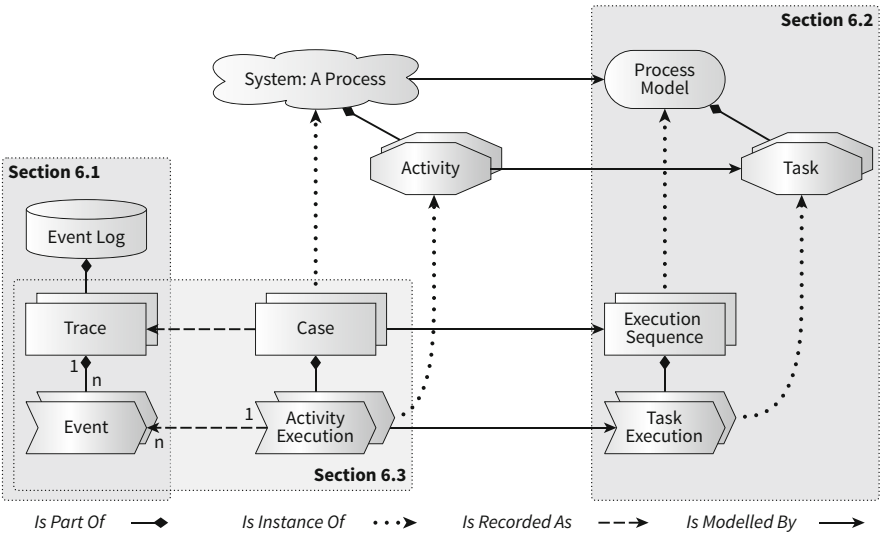


Figure 6.1 Preparation for conformance checking relates to different representations of a process, event logs and process models, and how they are linked

and a process model need to be linked. Before turning to this relation, however, we need a clear definition of what we understand as event data. Below, we first provide a basic model of events, traces, and event logs. We then turn to organizational and technical aspects of obtaining event logs in practice.

6.1.1 Essential Notions of Event Data

Definition 1 (Events and Attributes) Let $\mathcal{E}$ be the universe of events. An event $e \in \mathcal{E}$ denotes that something has happened. Events are annotated with attributes from some domain $\mathcal{N}$. That is, for all events $e \in \mathcal{E}$ and attribute $n \in \mathcal{N}$, $\#_n(e)$ is the value of attribute n for event e. Each event has at least two attributes, *ID* and *Time*, which capture its identity and its time of occurrence, respectively.

While attributes may relate to various aspects, a few attributes are commonly observed in practice in addition to the event's identity and time of occurrence. For instance, events are often annotated with a type that indicates what has happened. In the example of the loan application process (see Table 2.2) the event e_{13} has an identity denoted by the subscript, $\#_{ID}(e_{13}) = 13$, and a time of occurrence, $\#_{Time}(e_{13})$ = January 1, 12:31. In addition, it is assigned the value 'A5634' for the attribute 'Application', i.e., $\#_{Application}(e_{13})$ = 'A5634'.

The main task of conformance checking is to see whether the recorded event data matches the behaviour described by a process model. The link between the two is

established by the activities and the cases of the process that is modelled or recorded, respectively. We capture these concepts as follows:

Definition 2 (Activities and Cases) Let $\mathcal{A}$ be the universe of activities and C be the universe of cases. Then, an event $e \in \mathcal{E}$ is assumed to have attributes *Activity*, *Case* $\in \mathcal{N}$, such that $\#_{Activity}(e) \in \mathcal{A}$ is the activity for which the execution is signalled by the event as part of the case $\#_{Case}(e) \in C$.

For the aforementioned event e_{13} in Table 2.2, for instance, it holds that $\#_{Activity}(e_{13}) =$ 'Application submitted'. That is, the event denotes that the execution of the activity of submitting the application has been recorded, for one specific case. The latter may be characterized in this example by the identifier of the loan application, i.e., $\#_{Case}(e_{13}) =$ 'A5634'.

Excursion 12

What is our case?

Let us come back to the question of what constitutes a case. Raw event data can be viewed from different angles, depending on the perspective assumed by some analytics question. For example, there is an inherent duality between resources that execute the process (e.g., people, machines, software services), and the elements that would constitute a process instance (e.g., customer order, patient in a hospital, complaint). Typically, when looking at process models, we are interested in the latter. We can, however, also ask the question of how the resources move through the process from activity to activity.

The notion of a case determines how we package the events and it impacts the analysis results that we obtain in a conformance checking project. Note that as long as we keep the information about the resources in the events as attributes, we can later regroup the events according to the resources, if we are interested in that perspective. Thus, good practice is to start with the process perspective that matches the models at hand.

Events that denote activity executions as part of the same case are grouped together to form a trace. A trace thereby represents the behaviour recorded for one specific case of the process. When referring to event data in general, a trace may simply be considered as a set of events, each having the same value for the *Case* attribute. Yet, adopting a language-based view for conformance checking (see Section 2.4) it is sometimes convenient to think of a trace simply as a sequence of activities. Given a set of events with equal values for the *Case* attribute, this sequence is obtained by ordering the events according to their timestamps (assuming that this order is total) and then interpreting each event solely by means of its *Activity* attribute. We capture either view on traces as follows:

Definition 3 (Trace) A set of events $E \subseteq \mathcal{E}$ is a trace if $\#_{Case}(e) = \#_{Case}(e')$ for all $e, e' \in E$. A trace E is also represented as a sequence of activities $\sigma =$

$\langle a_1, a_2, \ldots, a_{|E|} \rangle \in \mathcal{A}^*$, such that $\langle e_1, e_2, \ldots, e_{|E|} \rangle \in E^*$ is the sequence obtained when ordering all events by their timestamps, $\#_{Time}(e_i) < \#_{Time}(e_j)$ for $1 \leq i < j \leq |E|$, and it holds that $\#_{Activity}(e_i) = a_i$ for $1 \leq i \leq |E|$.

Referring to the example in Tables 2.1 and 2.2, we see that the events $e_{13}, e_{14}, e_{22}, e_{37}, e_{42}, e_{54}, e_{64}, e_{65}$ form a trace when cases are defined by the identifier of the loan application. As discussed earlier, this trace may be represented as a sequence of activities, i.e., $\langle As, Aa, Fa, Sso, Ro, Do, Da, Af \rangle$.

An event log captures the information of several traces, thereby representing the behaviour recorded for the process as a whole. However, the two aforementioned views on how to capture a trace, i.e., a set of events or a sequence of activities, also yield two different views on an event log. It may be considered as a set of distinct traces, or a multiset of sequences of activities.

Definition 4 (Event Log) A set of traces $L \subseteq \wp(\mathcal{E})$ is an event log, if for all distinct $E, E' \in L$ it holds that $E \cap E' = \emptyset$, and $e \in E$ and $e' \in E'$ implies that $\#_{Case}(e) \neq \#_{Case}(e')$. An event log L is also represented as a multiset of sequences of activities, i.e., the event log is a multiset of $\mathcal{A}^*$.

Note that, when representing an event log as a multiset of $\mathcal{A}^*$, we adopt set notation when talking about containment and summation. As detailed in Chapter 5, by $\sigma \in L$, we refer to the fact that the activity sequence σ representing a trace is part of the event log. Also, when summing up the value of a function $f : \mathcal{A}^* \to \mathbb{R}$, we write $\sum_{\sigma \in L} f(\sigma)$ to denote the sum of applying the function to each *occurrence* of the trace in the event log.

6.1.2 Sources of Event Data

Many different sources of event data exist. Potentially any information system that supports the conduct of a process can be a source of event data. In the best case, we have specific process-aware information systems (PAIS) [38] that log event data during the execution of activities as part of a case, along the causal and temporal dependencies defined by a process model. At the other end of the spectrum, processes are documented or tracked by sensors, or in the worst case, only manual documentation in unstructured text exists. All these scenarios require different and tailored handling to extract the information into the format we require for conformance checking: event logs.

Preparing event data for conformance checking can pose a significant challenge. This becomes apparent when reflecting on the assumptions on event data as imposed by the model introduced above:

- The scope of the process, its start and its end, must be well-defined.
- Information on the time (or at least the order) of occurrence needs to be available, so that events can be interpreted in terms of temporal and causal relations.

- The notion of a case needs to be well-defined and reflected in the data assigned to each event.
- Events shall assume an appropriate level of abstraction, i.e., they shall indicate a single execution of one particular activity of the process.

In order to satisfy these assumptions and obtain an event log as put forward in Definition 4, several organizational techniques and hurdles have to be overcome in common conformance checking projects.

Organizational Aspects to Getting Event Logs Let us consider the following situation. As a process analyst (or a team of analysts), we need to check conformance of a particular process in an organization. First, the stakeholders in the conformance checking project need to agree on the process to be analysed, and the goal of the project should be clearly communicated. A standard information system can easily contain more than 10,000 relational tables storing an organization's data, meta data, and internal system data [114]. Simply starting to dig into these amounts of data, any analyst would risk getting lost. Therefore, we need to make sure that the right people are involved with the project. Ideally, organizations have *process owners* that have a good overview over the end-to-end process and can refer the process analyst to technical developers for detailed questions about the stored data.

The primary goal of the project could be that the organization wants to achieve a degree of conformance to a given process model. To be able to achieve this goal, the boundaries and milestones in the process need to be known. Typical questions include: What are the entry points into the process? Is there more than one end state? Where are the decision points? What are the activities performed in the process? Who is performing those activities?

When the process is sufficiently understood, the analyst needs to identify the objects that carry the data handled during process execution. In case of our running example of the loan application process, such objects are the applications, the offers, and the applicants. Typically, an object has at least one table in a relational data schema, and these tables are connected with foreign key relations. Foreign keys are links that potentially tie one instance of an object to another object instance. For example, it may link an offer to the respective loan application.

Relevant objects can best be identified by asking the people involved. In such a meeting, the goal should be to acquire a picture of the relational data schema, as well as the most relevant attributes of the important objects. For a loan application, for example, we would be looking for attributes like the amount, interest rate, applicant status, timestamp, etc. However, it is not always clear how these values are represented in the relational data schema.

Also, legal and privacy aspects of the analysis project must be documented and be conducted in line with company ethics and the legislation. For instance, the European Union's General Data Protection Regulation (GDPR) states that organizations failing to comply to the regulation can face fines of up to €20,000,000 or four percent of their global annual gross turnover (whichever is higher) [78]. Therefore, event data gathered must not only be secured and protected against attacks, but additional rules on personal data of people involved in the process

need to be considered. As such, in many conformance checking projects, legal departments of an organization must be involved in order to ensure that the project is conducted in line with the existing regulations.

Technical Challenges of Getting Event Logs Collecting data from different information systems requires a basic understanding of the process and the way the data is stored in the systems. As relevant event data can be scattered in information systems in different forms, there is a need for data normalisation. For instance, the order of events from various systems is important for conformance checking. Hence, we need to be able to compare timestamps across all event sources. As a normalisation step, it is good practice to convert all timestamps to one common format and incorporate normalisation factors, if needed. In general, such normalisation is facilitated by a standardized target format to hold the event data for conformance checking, which also serves as a unified interface to conformance checking tools.

The IEEE Standard for eXtensible Event Stream (XES) [136] was developed as such a format for event data. It helps process mining and conformance checking researchers and practitioners to exchange event logs and to use different algorithms on the same event data. The schema of the XES format is shown in Figure 6.2.

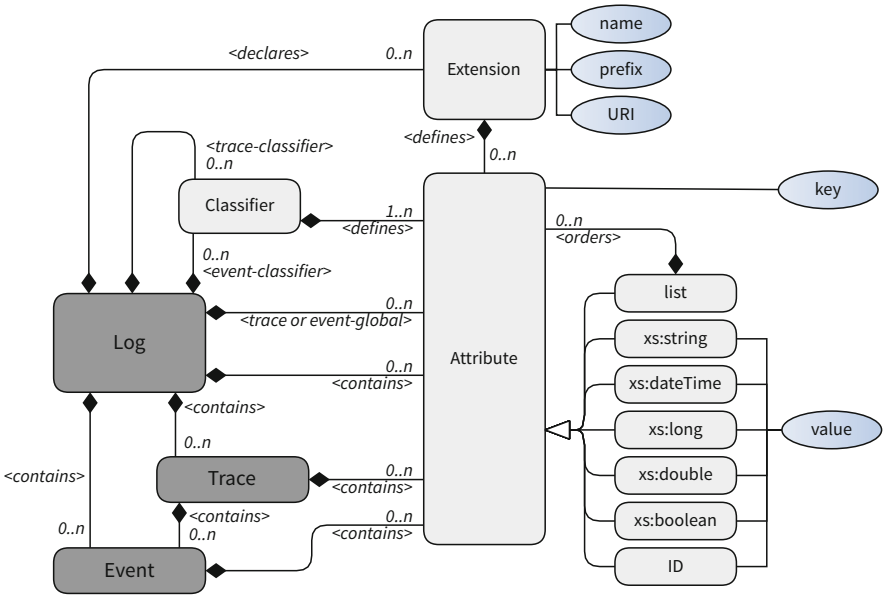


Figure 6.2 Schema of event data according to the XES-specification; cf. [136]. Every element can be extended to cater for various domain-specific needs. On the left, there is the hierarchy of logs, traces, and events. They are in a containment relation. They all can have attributes, which can be of different data types (e.g., numeric, date, textual) or even be composed (self-contains). Events and Traces can be classified with a classifier

We note that the XES model resembles the formal model introduced above: A log contains traces, which contain events. All these objects have attributes, which can be of different types. An event log also specifies classifiers that classify events or traces. This allows us to flexibly control the mechanism to group events and also to specify the amount of information encoded in a class. More concretely, a coarse-grained classifier could only consider the attribute 'name' of events to determine the event class. A more fine-grained classifier could also consider lifecycle transitions, as detailed below, and thus distinguish between start and end events of the same 'name'. This flexibility helps to tune the classification of events to the required abstraction level. Finally, the XES standard explicitly declares extensions that allow organisations to capture additional data in the XES format.

Next, we revisit our event log in Table 6.1 and translate it into the XES format which results in the representation shown in Table 6.2. After the Extensible Markup Language (XML) declaration at the top, the main entity is the log itself. It is the root of the event log that groups the traces. Therefore, before we can produce an XES document, we need to collect all events belonging to the traces. Depending on the data sources, this can pose a significant challenge.

We see immediately that the XES version is more verbose than the table in Table 6.1, as all event attributes are stored as elements instead of columns. The `<log>` element is the outermost entity and contains all traces as children. Within one trace, some trace attributes can be given. Subsequently, the events of the trace are ordered by their timestamp.

Sometimes, logging happens in a fine-grained manner, such that there is not just one event per activity execution, but several. If so, an event log contains records about the different states of activity execution, or more precisely, for the state transitions passed during the execution of an activity. Such states and

Table 6.1 Example of log of the loan application process, from the perspective of the offers

Event	Application	Offer	Activity	Amount	Signed	Timestamp
...	...	...	...	...	...	...
e_{30}	$O3521$	$A5636$	Select and send offer	€500		Jan 04, 16:32
...	...	...	...	...	...	...
e_{37}	$O3541$	$A5634$	Select and send offer	€1500		Jan 05, 12:32
e_{38}	$O3521$	$A5636$	Receive offer		NO	Jan 05, 12:33
e_{38}	$O3521$	$A5636$	Cancel offer			Jan 05, 12:34
e_{39}	$O3542$	$A5636$	Select and send offer	€500		Jan 05, 13:29
e_{40}	$O3542$	$A5636$	Receive offer		YES	Jan 08, 08:33
e_{41}	$O3542$	$A5636$	Accept offer			Jan 08, 16:34
e_{42}	$O3541$	$A5634$	Receive offer		NO	Jan 10, 10:00
...	...	...	...	...	...	...
e_{54}	$O3541$	$A5634$	Decline offer			Jan 10, 10:04
...	...	...	...	...	...	...

Repeated from Table 2.3

Table 6.2 XES representation of the event log in Table 6.1

```
<?xml version="1.0" encoding="utf−8"?>
<log xes.version="1.0">
 ...
 <trace> <!−− Trace of application A5634 −−>
  ...
  <string key="concept:name" value="A5634"/>
  <event>
   <string key="concept:name" value="Select and send offer"/>
   <string key="offer" value="O3541"/>
   <float key="amount" value="1500.00"/>
   <date key="time:timestamp" value="2018−01−05T12:32:04.000+02:00"/>
  </event>
  <event>
   <string key="concept:name" value="Receive offer"/>
   <string key="offer" value="O3541"/>
   <boolean key="signed" value="False"/>
   <date key="time:timestamp" value="2018−01−10T10:00:00.000+02:00"/>
  </event>
  <event>
   <string key="concept:name" value="Decline offer"/>
   <string key="offer" value="O3541"/>
   <date key="time:timestamp" value="2018−01−10T10:04:00.000+02:00"/>
  </event>
  ...
 </trace>
 <trace> <!−− Trace of application A5636 −−>
  <string key="concept:name" value="A5636"/>
  <event>
   <string key="concept:name" value="Select and send offer"/>
   <string key="offer" value="O3521"/>
   <float key="amount" value="500.00"/>
   <date key="time:timestamp" value="2018−01−04T16:32:00.000+02:00"/>
  </event>
  ...
 </trace>
</log>
```

state transitions are known as the activity lifecycle [135]. Examples of common lifecycles, with increasing complexity, are illustrated in Figure 6.3. A simple lifecycle may comprise only a single transition, i.e., an event corresponds to an atomic signal that a particular activity has been executed. A slightly more complex lifecycle contains two transitions, so that the respective events signal the start and the end of activity execution. However, lifecycles may even be more complex and contain transitions that denote, for instance, that activity execution has been paused and later resumed.

In general, activity lifecycles can be arbitrarily complex, notably, when the states in the lifecycle also distinguish resource allocations and hand-overs within one activity execution. In principle, conformance checking could also be done on this refined level: Each transition in the lifecycle would be considered as an individual execution of a more fine-granular activity. Assuming that the tasks in a process model also reflect these fine-grained activities, the techniques discussed in the remainder of this book can be applied in a straightforward manner. Whether we shall consider this level of granularity depends on the scope of the conformance

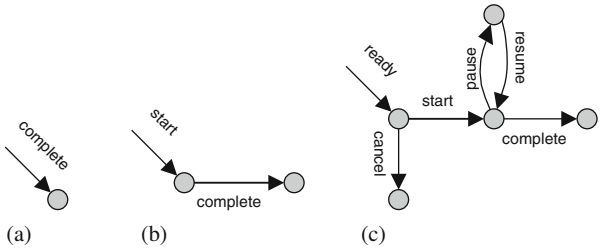


Figure 6.3 Activity lifecycles from simple, (**a**) and (**b**), to more complex, (**c**). Edges mark transitions, and nodes represent states. (**a**) Single event. (**b**) Start and complete. (**c**) More complex lifecycle

checking project and the questions that are asked. If the question, whether a process' activities are executed according to the given activity lifecycles, is not prioritized, the overhead of conducting conformance checking on such a fine-grained level could outweigh the gained knowledge.

6.2 Formalising Process Models as Petri Nets

In Part I of this book, we used a simplified version of BPMN to model processes. While BPMN provides a standardized language, widely adopted in industry, for this purpose, it lacks a mathematical foundation of its execution semantics. While the BPMN specification contains a description of these semantics based on states, it covers only a subset of the language. Yet, this description is inspired by Petri nets, a formalism that does not only provide a comprehensive mathematical foundation, but also comes with plethora of formal results and analysis techniques. Therefore, in the remainder of this book, we use Petri nets as a formal foundation for process models.

6.2.1 Essential Notions of Petri Nets

Petri nets are particular types of graphs that consist of places, transitions, and arcs between them. A major advantage of the Petri net formalism is that its mathematical definition is equivalent to its visual representation. That is, the places of a Petri net are depicted as circles and transitions as squares, which are connected by directed arcs. An example Petri net is shown in Figure 6.4. This Petri net contains 12 places, and 13 transitions. Each transition captures a task of the process model, for which the label indicates the respective activity. In the example, there is one transition, i.e., *As*, which represents the activity 'Application submitted'. The activity 'Decline application', in turn, is captured with two tasks, i.e., there are two transitions, identified as Da_1 and Da_2, that model the activity.

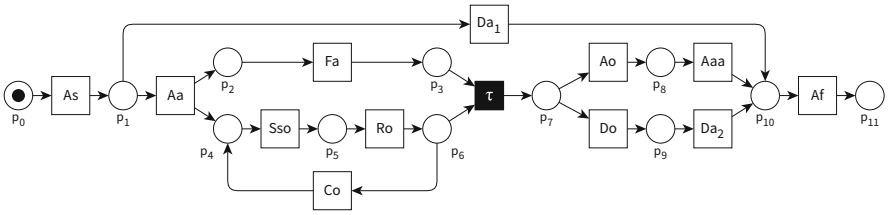


Figure 6.4 Petri net for the running example

The model also shows one τ-labelled transition, typically depicted by a solid, black square. Such a transition does not represent any activity of the process. To capture such transitions, given a set of activities $A \subseteq \mathcal{A}$, we define $A^\tau = A \cup \{\tau\}$ with $\tau \notin \mathcal{A}$ as the set of labels that may be used in a Petri net that models a process with activities A.

Definition 5 (Labelled Petri Net) Let $A \subseteq \mathcal{A}$ be a set of activities. A labelled Petri net is a tuple $N = (P, T, F, \lambda)$, with P as the finite set of places, T as the finite set of transitions, where $P \cap T = \emptyset$, $F \subseteq (P \times T) \cup (T \times P)$ as the flow relation, and $\lambda : T \to A^\tau$ as the labelling function assigning to each transition an activity or τ, indicating that it is not associated with an activity.

In this book, we always assume labelled Petri nets when referring to Petri nets. Moreover, for a node $n \in P \cup T$, a place or a transition of a Petri net, $\bullet n$ ($n\bullet$) is the predecessor (successor) set of n according to F. The predecessors (successors) of a transition are also referred to as input (output) places, and vice versa, the terms input (output) transitions are used for the predecessors (successors) of a place.

The structure of a Petri net in terms of the flow relation can be captured by its *incidence matrix*. It is defined as a matrix of $|P|$ rows and $|T|$ columns, in which values of -1 and $+1$ indicate the presence of a flow arc from a place to a transition, or from a transition to a place, respectively, whereas a value of 0 denotes that a place and a transition are not connected. The incidence matrix of our example Petri net is given in Figure 6.5.

When using Petri nets to model a process, it is often convenient to assume some notion of well-structuredness. This is captured by the concept of a workflow net.

Definition 6 (Workflow Net) A workflow net is a Petri net that has one distinguished source place $start \in P$, with $\bullet start = \emptyset$, and one distinguished sink place $end \in P$, with $end\bullet = \emptyset$, and all other nodes $n \in (P \cup T) \setminus \{start, end\}$ are on a path from place $start$ to place end.

We note that the Petri net in Figure 6.4 is a workflow net. There is a distinguished source place, p_0, a distinguished sink place, p_{11}, whereas all other nodes are on a path between them.

Execution semantics of a Petri net are defined as a token flow game. That is, the state of a Petri net is defined as a distribution of tokens over places, which is called a *marking*. As such, a state, or marking, corresponds to a multiset of places, in which

	As	*Aa*	Da_1	*Fa*	*Sso*	*Ro*	*Co*	τ	*Ao*	*Do*	*Aaa*	Da_2	*Af*
p_0	−1	0	0	0	0	0	0	0	0	0	0	0	0
p_1	1	−1	−1	0	0	0	0	0	0	0	0	0	0
p_2	0	1	0	−1	0	0	0	0	0	0	0	0	0
p_3	0	0	0	1	0	0	0	−1	0	0	0	0	0
p_4	0	1	0	0	−1	0	1	0	0	0	0	0	0
p_5	0	0	0	0	1	−1	0	0	0	0	0	0	0
p_6	0	0	0	0	0	1	−1	−1	0	0	0	0	0
p_7	0	0	0	0	0	0	0	1	−1	−1	0	0	0
p_8	0	0	0	0	0	0	0	0	1	0	−1	0	0
p_9	0	0	0	0	0	0	0	0	0	1	0	−1	0
p_{10}	0	0	1	0	0	0	0	0	0	0	1	1	−1
p_{11}	0	0	0	0	0	0	0	0	0	0	0	0	1

Figure 6.5 Incidence matrix of the example Petri net of Figure 6.4

the number of occurrences of a place denote the number of tokens assigned to it. Again, there is a standard visualization of markings that is equivalent to their formal definition. In our example, the visualized marking defines that a single token (small solid black circle) is assigned to place p_0. This corresponds to the multiset of places given as $[p_0]$.

The execution of an activity is represented by the execution of a task, which is called firing of a transition in Petri net terminology. A transition can fire in a marking, if all its input places are assigned a token in this marking. Firing a transition then yields a new marking that is obtained by consuming one token from all input places of the respective transition, and producing one token on all of its output places. Using this simple firing rule, the entire set of possible markings of the Petri net, called its state space, is defined. Next, we define these concepts formally (note the remarks in Chapter 5 on applying operators for multisets also in the context of sets).

Definition 7 (Marking, Firability, Reachability, Boundedness) Let $N = (P, T, F, \lambda)$ be a Petri net. A marking m is a multiset of places, i.e., $m \in \mathcal{B}(P)$. A transition $t \in T$ is *enabled* in a marking m iff $\bullet t \leq m$. *Firing* transition t in m results in a new marking $m' = m - \bullet t + t\bullet$, i.e., tokens are removed from $\bullet t$ and added to $t\bullet$. This firing is denoted by $m[t\rangle m'$. A marking m' is *reachable* from m if there is a sequence of firings $\sigma = \langle t_1, t_2, \ldots, t_n \rangle$ that transforms m into m', denoted by $m[\sigma\rangle m'$.

Given an initial marking m_i of Petri net N, the set of reachable markings of N is denoted by $\mathsf{RS}(N)$. The *Reachability Graph* of N, denoted by $\mathsf{RG}(N)$, is a graph in which the set of nodes is the set of markings $\mathsf{RS}(N)$ and the edges correspond to firing transitions, i.e. the edge $(m_1, t, m_2) \in \mathsf{RS}(N) \times T \times \mathsf{RS}(N)$ exists, if and only if $m_1[t\rangle m_2$.

A Petri net is said to be *k-bounded* or simply *bounded*, given an initial marking, if all reachable markings are bounded by k, i.e., no place in a marking contains more than k tokens. When k is 1 the Petri net is called *safe*.

From the above, it becomes apparent that the behaviour described by a Petri net is fully determined by that net, an initial marking, and all markings reachable from the initial marking. Figure 6.6 shows the reachability graph, or state space, of the example Petri net from Figure 6.4. It clearly shows the initial marking $[p_0]$, which corresponds to the state in which a process starts. Also, we note that there is a distinguished marking $[p_{11}]$ that marks only the sink place, thereby indicating that the process has finished execution. In this book, we assume that such a final marking is given, which leads to the notion of a system net.

Definition 8 (System Net) A system net is a tuple $SN = (N, m_i, m_f)$, where N is a Petri net and m_i, m_f define the initial and final marking of the net, respectively.

Apart from the state space, a system net also defines a set of firing sequences, sequences of transitions that can be fired one after the other, starting in the initial marking, and ending in the final marking. The set of these sequences is the language

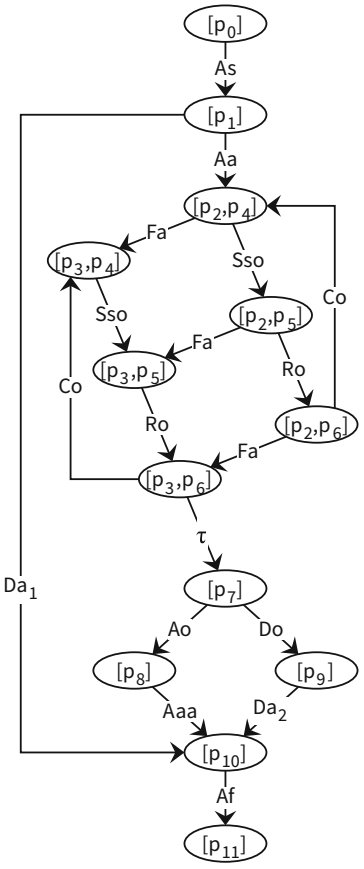


Figure 6.6 Reachability graph, or state space, of the Petri net for the running example

of the system net. Using $[p_0]$ and $[p_{11}]$ as initial and final markings, respectively, we see that, for our example, the language is infinite. The state space in Figure 6.6 contains two loops, i.e., the language can indefinitely often contain the transitions *Sso*, *Ro*, and *Co*. Yet, all finite sequences eventually reaching marking $[p_{11}]$ are part of the language of the system net.

Definition 9 (System Net Language) Let $SN = (N, m_i, m_f)$ be a system net. The set $\mathcal{L}_{SN} = \{\sigma \mid (N, m_i)[\sigma\rangle(N, m_f)\}$ denotes the language of SN.

In the analysis of Petri nets, the *Marking Equation* [75] is an important tool. It establishes a link between the structure of a Petri net and the reachability of markings in this net. Specifically, the Marking Equation provides a necessary, yet not sufficient condition for reachability of markings.

For illustration, let p be a place of a Petri net with $\bullet p = \{x_1, \ldots, x_n\}$ and $p\bullet = \{y_1, \ldots, y_l\}$. Then, given an initial marking m_i for this Petri net, the following equality holds for any firing sequence σ that starts in m_i:

$$m(p) = m_i(p) + \vec{\sigma}(x_1) + \cdots + \vec{\sigma}(x_k) - \vec{\sigma}(y_1) - \cdots - \vec{\sigma}(y_l).$$

The previous equation can be generalized:

$$m(p) = m_i(p) + \sum_{x_i \in \bullet p} \vec{\sigma}(x_i) - \sum_{y_i \in p\bullet} \vec{\sigma}(y_i).$$

If we formulate the previous equation for all places in a Petri net, we can compress it using a matrix notation, which yields the Marking Equation:

$$\vec{m} = \vec{m_i} + \mathbf{C} \cdot \vec{\sigma}$$

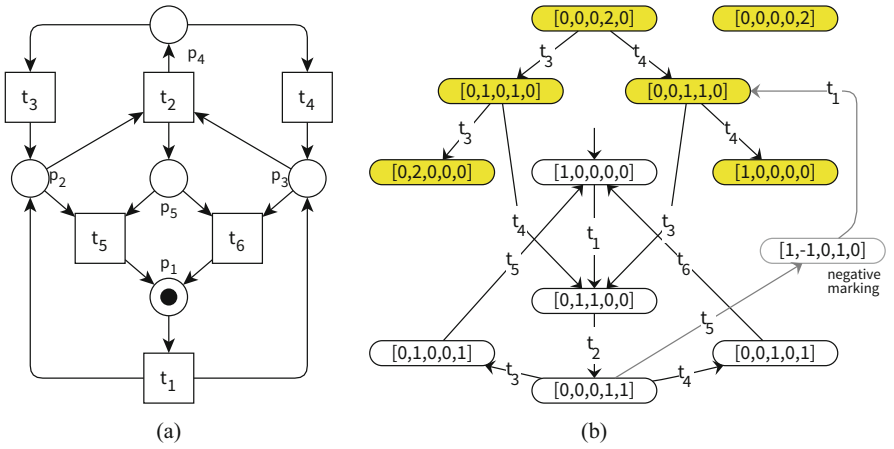
where $\vec{m}$ and $\vec{m_i}$ are place vectors and $\mathbf{C}$ is the incidence matrix of the Petri net.

This equation enables conclusion on the reachability of markings, which, in general, have to be non-negative. The set of markings m for which the following inequality holds

$$\vec{m} = \vec{m_i} + \mathbf{C} \cdot \vec{\sigma} \geq 0 \tag{6.1}$$

is called the *Potentially Reachable Set*, denoted by $\mathsf{PRS}(N)$, from marking m_i. All reachable markings of a Petri net fulfil Equation 6.1. However, the opposite is not always true. In general, there can be unreachable markings for which Equation 6.1 also holds, i.e., $\mathsf{RS}(N) \subseteq \mathsf{PRS}(N)$. A detailed discussion on the relation between $\mathsf{RS}(N)$ and $\mathsf{PRS}(N)$ can be found in [102].

Figure 6.7 illustrates the concept of the *Potentially Reachable Set*. Figure 6.7b depicts the set of (potential) markings reachable from the Petri net shown in Figure 6.7a. Looking at the figure, one can see that the sequence $T_1 = \langle t_1, t_2, t_5, t_1, t_3\rangle$ leads to a non-negative marking. However, looking at the set of reachable states, one can see that T_1 is not firable, since the non-negativity



$$\vec{m} \quad \vec{m_i} \qquad\qquad \mathbf{C} \qquad\qquad \vec{\sigma}$$

$$\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -1 & 0 & 0 & 0 & 1 & 1 \\ 1 & -1 & 1 & 0 & -1 & 0 \\ 1 & -1 & 0 & 1 & 0 & -1 \\ 0 & 1 & -1 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 & -1 & -1 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

(c)

Figure 6.7 Potentially reachable set: states in $\mathsf{PRS}(N) \setminus \mathsf{RS}(N)$ are highlighted

requirement is violated for one of the intermediate markings. However, this sequence reaches the same marking as the firing sequence $T_1 = \langle t_1 \rangle$.

6.2.2 Process Modelling Beyond Basic Petri Nets

While Petri nets form a mathematical basis for modelling processes, not all concepts of processes found in practice can be expressed using the Petri nets described above. In real-life process modelling, further concepts such as cancellation and multiple-instances are required.

Cancellation refers to the idea that some activities, when executed, cancel entire parts of a process. For example, cancelling a loan application before it has been activated cancels activities related to processing of the corresponding offer, regardless of which state that part of the process is in.

Another example is the modelling of multiple instances. Consider, for example, the process of taking witness reports of a traffic accident by an insurance company.

Beforehand, it is not known how many witness reports are going to be taken and once the first witness report comes in, decisions may be made to ask for further reports. These reports are generally not requested sequentially, one after the other, but multiple reports are requested in parallel.

Behavioural concepts such as the above ones have been captured as so-called workflow patterns. Today, there exists a large collection of such patterns along with assessments about the extent to which they are supported by process modelling tools. Translating all these patterns to basic Petri nets as introduced in this chapter is not possible. However, extensions of the Petri net formalism may increase its expressiveness, thereby making it possible to capture more advanced behavioural patterns. Here, we reflect on two such extensions.

The first extension adds so-called 'inhibitor arcs' to Petri nets. These arcs can be defined between a place and a transition to indicate that a transition can only fire if the respective place is not marked, i.e., it is not assigned a token in the current marking. Put differently, any token in the place inhibits the firing of the connected transition.

The second extension considers the definition of so-called 'reset arcs'. These arcs are defined between a place and a transition to indicate that after the firing of a transition, the respective place is no longer marked. That is, upon firing the transition, all tokens are removed from the place, so that the place is not marked in the reached state. As such, firing of the transition resets the respective place.

With these extensions, behavioural patterns such as cancellation and modelling of multiple instances can be captured in a straightforward way. We illustrate this with the example scenario of cancelling a loan application. Figure 6.8 shows a fragment of our running example, which also includes an additional transition *Ca* that represents an activity 'Cancel application'. It enables the customer to cancel the application, after the process continued from the offer selection part, but before the application is activated. The leftmost τ transition enables the cancellation. *Ca* can only fire if there is no token in the place after *Aaa* and *Da*, indicated by the inhibitor arc with a circle. When fired, the transition *Ca* removes tokens from three places, if there are any, as indicated by the reset arcs with double-headed arrows.

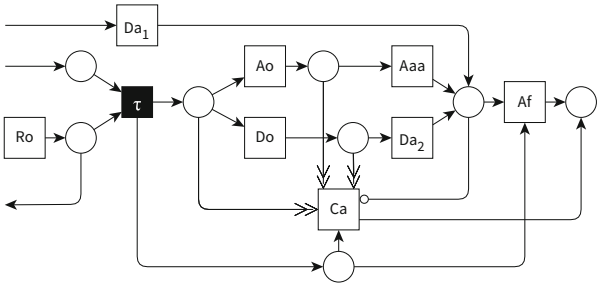


Figure 6.8 Reset-inhibitor example that allows for cancelling the application after an offer has been sent back by the customer

The transition *Af* to the right removes a token from the input place of transition *Ca*, thereby cleaning up the state if cancellation did not occur.

In the remainder, we focus on basic Petri nets as defined formally in this chapter. However, much of the presented work extends to more expressive Petri nets, such as those with reset and/or inhibitor arcs.

6.3 Relating Event Logs and Process Models

Conformance checking requires that a link between the modelled and recorded behaviour of a process is established. Revisiting the overview of the concepts involved in conformance checking (see Figures 6.1 and 6.9) this link is established by means of the activities of a process, as well as their executions as part of specific cases. On the one hand, a task in a process model represents an activity, so that a task execution indicates an activity execution. An execution sequence comprising task executions thereby represents a case of the process. On the other hand, activity executions are recorded as events and all events of a trace jointly presented the information recorded about a specific case.

From the above, it becomes clear that relating event logs and process models actually requires both, relating events and activity executions as well as relating tasks and activities (and, thus, their executions). However, in this book, we focus on the former relation and assume that the matching between activities and tasks is given, or can trivially be established. Put differently, we assume that the models

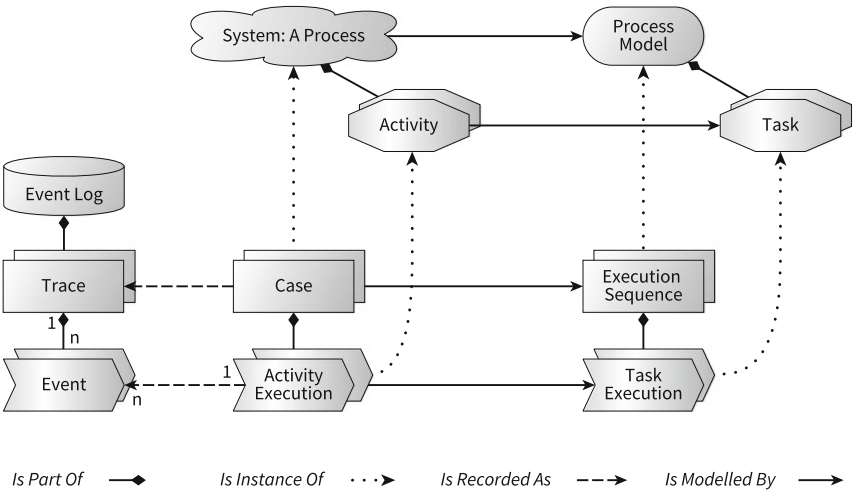


Figure 6.9 Different representations of a process, event logs and process models, and how they are linked, repeated from Figure 6.1

are captured at the right abstraction level, so that each task represents a single activity. Note, however, that there may be multiple tasks that model an activity, which is useful to capture that an activity may be executed in different contexts in terms of process progress. In this book, the relation between tasks and activities is always encoded directly by the task labels, which denote activities, with numerical subscripts distinguishing different tasks that refer to the same activity.

The relation between events and activity executions is typically more problematic in practice. This difference is apparent when reflecting on how the different representations of a process, an event log and a process model, are commonly derived. When capturing a process with a model, the definition of the tasks representing activities is part of the construction of the model. Being done by expert users in a careful manner, this definition of tasks can be expected to be precise, unambiguous, and at the right level of abstraction. This is not true for event data, though. Events used as the basis for conformance checking stem from diverse sources, at predefined granularity, with little control over data quality aspects. As a consequence, establishing a link between events and activity executions faces various problems.

The fact that conformance checking has to rely on event data as it is recorded by information systems, which not necessarily assumes the form required for analysis, means that also the relation between events and traces may need to be handled explicitly. That is, even if recorded events can be linked to activity executions, their grouping into traces that represent individual cases may be far from trivial.

6.3.1 Relating Events and Activity Executions

Below, we provide an overview of common problems faced when trying to link events and activity executions. While we point to solution strategies, an in-depth description of the respective methods is beyond the scope of this book.

Different Sources Processes are often supported by a multitude of information systems. The number and heterogeneity of sources from which data is integrated in the event log must be taken into account when relating events and traces. Analysts should be aware that the timestamps can be affected due to asynchronous clocks in the respective systems. In addition, event sources may also differ in their adopted temporal scale when recording data, e.g., timestamps only indicate the day of activity executions, but do not order the executions throughout a single day.

Approaches Timestamp anomalies in event data can be approached with techniques for outlier detection or based on time boundary constraints. As a starting point for such techniques, we refer the reader to [89] and [104], respectively. Issues that stem from the temporal scale employed by different event sources can be addressed within one trace by exploiting information about related traces [60].

Noisy Data Generally, before checking conformance, a data cleaning step can be necessary. The process of recording events in processes can itself be subject to errors. Erroneous event data potentially stems from manual recording of events or noisy sensors. Stripping noisy and irrelevant event data is one of the most important problems to be addressed in order to conduct meaningful analysis.

Approaches Filtering infrequent events with the assumption that infrequent events are noise is one direction to approach this problem. Again, methods for outlier detection provide an angle to identify noise, assuming that it is indeed infrequent. An overview of noise patterns and strategies to identify them has been presented in [105], while filtering strategies for event data have been proposed in [25, 128].

Process Change In many domains, processes are not static, but subject to continuous change. The evolution of the context in which a process is executed, e.g., the legal and regulatory context or the supporting technical infrastructures, lead to frequent adaptations of a process. Yet, event data at hand does not always explicitly state this context. As such, the event data may have recorded different variants of a process, i.e., it may include so-called *concept drift*. Ignoring this drift would mean that analysis is based on unified representation of multiple variants of a process, thereby introducing a severe bias.

Approaches A first set of techniques relates to the identification of drift points in the data. In the simplest case, a change happens suddenly, meaning that there is a specific date at which the process has been adapted. Then, statistical methods comparing two adjacent time windows before and after the change, can detect the respective behavioural difference. Based thereon, the event data is partitioned accordingly and the data before and after the change is analysed separately. Following this general idea, various specific techniques for drift detection have been presented in the literature [16, 22, 62]. Also, online learning techniques provide an angle to achieve robustness of analysis models against concept drift [63].

Granularity The assumption of a 1:1 relation between events and activity executions does not always hold true. Events can be recorded in a higher frequency than required, meaning that, for instance, the joint occurrence of a set of events indicates that an activity has been executed. Yet, the opposite may also be observed. For example, an event may indicate that a milestone has been reached in a process, thereby signalling that a set of activities has been executed successfully.

Approaches Grouping low-level events to higher-level events, thereby projecting the data to a higher level of abstraction is one way to cope with such issues. Specifically, this may be done with rule-based approaches based on boundary conditions [11]. Assuming that low-level events follow certain patterns, which are known, the technique in [66] enables the abstraction of event data. If event data stems from location sensors, a relation to activity executions may be established by exploiting background information on the process at hand [99].

Ambiguity Sometimes, recorded event data is not precise, but ambiguous. For example, two events may be of the same type, but relate to two different activities of the process. In the worst case, there is solely a single type of event (e.g., events from a real-time locating system all include a timestamp and spatial coordinates of tracked objects), and each event could theoretically be linked to an execution of any activity.

Approaches Semantic ambiguities are one of the most difficult problems in relating events and activity executions. Only specialised solutions exist, as the problem tends to be domain-specific. One direction to address this problem, if only few events are ambiguous, is to exploit the context in which they have been observed in order to identify an activity execution; see for example [35, 103].

6.3.2 Relating Events and Traces

In many application scenarios, events carry information on a case of the process. For instance, in our running example of a loan application process, the identifier of the application distinguishes between different cases (note that other notions of a case may still be employed; see also Excursion 12). Based on such a key, also known as case identifier, the events that relate to activity executions within the same case can be grouped, which yields the traces of an event log.

However, if we are unable to identify, or simply lack a case identifier in the event data, we face the problem of relating events and traces. This is often observed if event data from different sources needs to be integrated. In such a context, we consider two scenarios: (1) If it is possible to correlate the events within one system by a case identifier, the problem is reduced to matching groups of events to one another, which can reduce the original combinatorial problem by orders of magnitudes. Auxiliary attributes that exist in several systems may be exploited to correlate events, using techniques such as those presented in [76]. (2) If the correlation of events even within one system is unclear, the problem becomes more difficult, due to the number of possible groupings being exponential in the number of events. Again, certain auxiliary attributes may serve as correlation anchors [76]. However, in general, complex optimization problems need to be solved in order to find a partitioning of events into traces [84].

Bibliographic Notes

Getting event data from information systems is a common problem for which many solutions have been proposed. Getting event data out of relational databases is discussed in [114, Chapter 4] and in [33, 61]. A procedure to get event data is also summarized in the report [55].

Petri nets are a well-known formalism for describing systems [75, 86]. Their formal grounding and their ability to represent common control flow structures

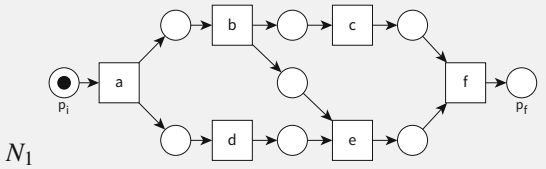
makes them particularly well-suited for process modelling. Moreover, Petri net-based formalisations of high-level languages for process modelling have been presented in the literature [36, 59]. Since Carl Adam Petri developed their foundations in terms of a generalization of automata theory [80], there has been considerable research in the field of Petri net theory. The class of Petri nets widely used in this book, workflow nets, was first introduced in [111]. Structural techniques for Petri net analysis play an important role in parts of this book. The reader can find a comprehensive summary in [102].

Furthermore, we argued that conformance checking requires that event logs and process models be related to each other. While this relation is established through the notions of activities and cases, establishing the respective links is often non-trivial. In Section 6.3, we discussed especially the problems faced when linking events and activity executions, as well as events and traces, and gave pointers to related work for each of the encountered problems.

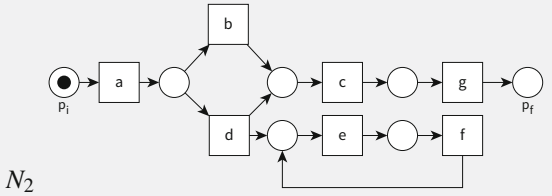
6.4 Exercises

6.A) Petri net analysis

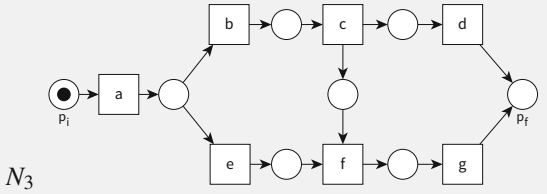
Given are the following system nets (N_1, N_2, N_3, N_4). By default, the initial marking m_i is defined as $[p_i]$, and the final marking m_f is $[p_f]$.



N_1



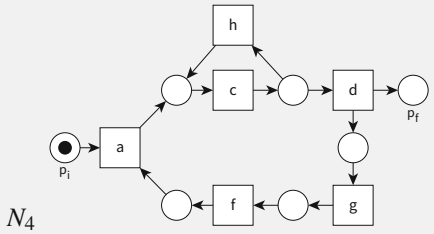
N_2



N_3

(continued)

6.A) (continued)



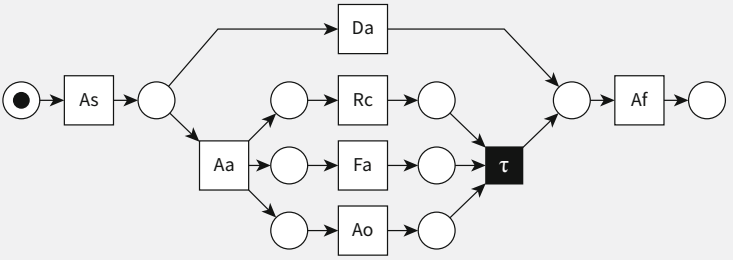
N_4

Analyse the systems and decide for each, whether:

a) it is bounded, and if so, whether it is also safe.
b) the final marking m_f can be reached from all reachable markings.
c) eventually a marking is reached that does not enable any transition.
e) for any transition, there exists a reachable marking that enables it.

6.B) Play out

Given is the following system net, with the initial marking m_i being visualized, while the final marking m_f being defined by a single token assigned to the rightmost place.



Consider the language of the above system net. How many sequences does it contain? Write down three of these sequences.

6.C) Inspecting an event log

Given the following event log.

Event	CaseID	Label	Lifecycle	Resource	Timestamp
e_1	*c01*	inspect	BEGIN	Ed Eagle	Apr 10, 15:11
e_2	*c02*	accept	BEGIN	Lu Light	Apr 10, 15:27
e_3	*c01*	inspect	COMPLETE	Ed Eagle	Apr 10, 15:41
e_4	*c02*	accept	COMPLETE	Lu Light	Apr 10, 15:27
e_5	*c02*	inspect	BEGIN	Ed Eagle	Apr 10, 15:30
e_6	*c01*	repair	BEGIN	Lu Light	Apr 10, 15:31
e_7	*c02*	inspect	COMPLETE	Ed Eagle	Apr 10, 15:40
e_8	*c01*	repair	COMPLETE	Lu Light	Apr 10, 16:19
e_9	*c02*	repair	BEGIN	Lu Light	Apr 10, 16:25
e_{10}	*c02*	repair	COMPLETE	Lu Light	Apr 10, 17:03

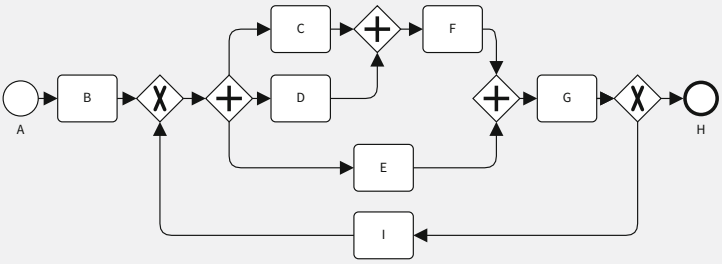
Extract the traces of this event log according to the following criteria:

1. A trace is determined by the 'CaseID' attribute. An event is specified by the combination of 'Label' and 'Lifecycle'.
2. A trace is determined by the 'CaseID' attribute. An event is specified by the label only. Successive BEGIN and COMPLETE events are merged.
3. A trace is determined by the 'Resource' attribute. An event is specified by the combination of 'Label' and 'Lifecycle'.

6.D) Formalising BPMN with Petri nets

Consider the following process models specified in BPMN. While many BPMN constructs can be translated directly in Petri nets, some also impose challenges in terms of semantics that cannot be captured with basic Petri nets. For each of the following BPMN models, come up with a Petri net that has the same language or explain why this cannot be achieved.

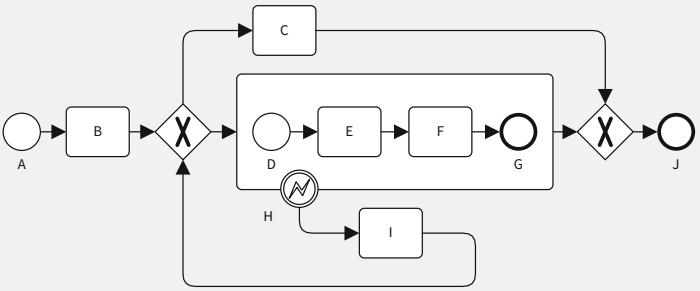
M_1 The following BPMN model contains solely basic constructs.



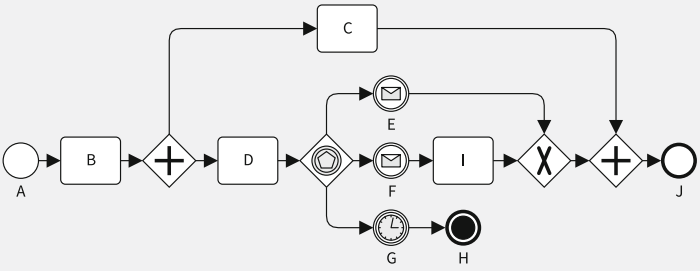
(continued)

6.D) (continued)

M_2 The following BPMN model contains a subprocess that contains the BPMN start event D, tasks E and F, and the BPMN end event G. When this subprocess is executed, an error H may occur at any time. Upon occurrence of this error, the subprocess is terminated immediately and the execution continues with task I.



M_3 The following BPMN model contains an event-based gateway (diamond shape containing a pentagon) that is followed by two BPMN message events E and F, and a BPMN timer event G. The semantics is that once the control flow reaches the event-based gateway, process execution continues with the BPMN event that is triggered first, i.e., with the first message (E or F) that arrives, or with the time-out G if no message is received within a particular time frame. If execution continues with the time-out G, the next element H is a BPMN terminate end event. Its semantics is that the overall process execution terminates immediately, meaning that also all concurrently executed tasks are stopped right away.



6) In the lab: tool support for converting raw data and for Petri net analysis

Check out the lab session to practise with tools that allow projecting raw data into event logs. Also, tutorials on Petri net analysis can be followed.

http://www.conformancechecking.com/CC_book_Chapter_6

Chapter 7
Aligning Event Data and Process Models

In Part I of this book, we reflected on the relation between event data and process models, and we presented several ways to exploit this relation. We explained that, to compute fitness, we need insights into the part of the event data that cannot be explained by the model. We also discussed how, for computing precision, we need to relate behaviour of the model that goes beyond what was recorded in the event data. In Chapter 4 we informally described several ways to relate event data and process models. We discussed rule checking, token flow replay and alignments on an intuitive level.

This chapter focuses on how to compute optimal alignments. Since alignments represent the most informative conformance artefact, their computation is a challenge. Still, the techniques described in this section, which are grounded in casting the alignment computation as a form of reachability with costs using search algorithms, can be often applied in real-life conformance projects.

Recall that alignments can be represented by combination of the trace on the one hand and an execution sequence of the model on the other hand, as shown below.

log trace	As	Aa	Sso	Ro	$\gg$	$\gg$	Ao	Aaa	Aaa	$\gg$
execution sequence	As	Aa	Sso	Ro	Fa	τ	Ao	Aaa	$\gg$	Af

The alignment above shows how a trace in the event log is aligned with an execution sequence in the process model. In this case, the trace does not show the activity "Finalise application" (Fa), nor the activity "Application finished" (Af), and the execution sequence cannot contain the transition "Accept and activate application" (Aaa) twice. Hence, these are deviations.

However, the alignment also shows something we did not discuss before, namely: the τ transition that fired in the model. To focus on process models with clear execution semantics, we consider Petri nets as a formal language for describing processes. In particular, we consider Petri nets in which transitions represent tasks, i.e., they are labelled by the activities of the process under consideration that are also

J. Carmona et al., *Conformance Checking*,
https://doi.org/10.1007/978-3-319-99414-7_7

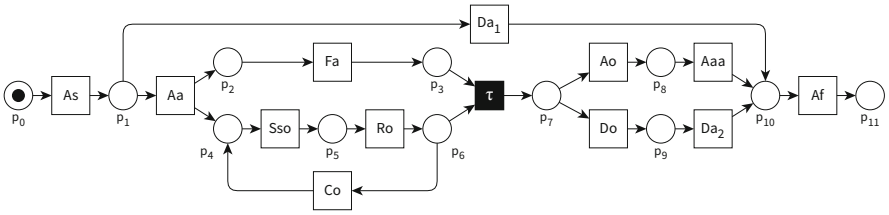


Figure 7.1 Petri net for the running example

referenced by the events in an event log. Yet, some transitions may not be labelled with an activity, but with a special label τ (so-called τ-labelled transitions). Firing such τ-labelled transitions in a Petri net might be required for correct execution, but are not considered a deviation. They are also often referred to as *routing* transitions or *invisible transitions*.

Figure 7.1 shows the Petri net translation of our running example again. Note that the label "Application submitted" (*As*) is translated into a transition in the beginning of the model, as it happens for the label "Application finished" (*Af*) at the end. Furthermore, we see that in the middle a transition labelled with τ is added for routing purposes and there are two transitions Da_1 and Da_2 labelled with the activity "Decline application". The alignment above corresponds to this Petri net.

This chapter is organized as follows. First, we unambiguously define alignments using a concept called a synchronous product (Section 7.1). Then, in Section 7.2 we describe a method to compute alignments based on the classical shortest paths Dijkstra algorithm, which is then improved in Section 7.3 by turning it into an $A^\star$ search problem. Then, in Section 7.4 several heuristics and optimizations to the base technique are reported, so that further improvements on the exploration can be attained. Finally, we show another example of the $A^\star$ evolution on a trace with a swapped activity in Section 7.5 before reflecting on the complexity of computing alignments in relation with similar problems in Section 7.6.

7.1 Alignments as Traces of the Synchronous Product

As discussed in Section 4.3 the basis for aligning a trace and a model is the notion of synchronous product. In Figure 4.3 the synchronous product was informally presented as a combination of a process model, a trace model and synchronised activities in the middle. In this section, we formalise the notions of a process model, a trace model and the synchronised product. Then, we show how an alignment corresponds to an execution sequence of the synchronous product from its initial marking to its final marking.[1]

[1] In this chapter we use the terms marking and state interchangeably.

At the foundation of any process lie activities. These are the building blocks of processes and represent atomic units of work that need to be performed by people or other actors. Recall that we assume there is a known universe of all possible activities denoted by $\mathcal{A}$ (see Section 6.1).

The basis for alignments are process models. We consider labelled Petri nets as the formal representation of a process. Recall from Definition 5 that a labelled Petri net is such that each transition is labelled with an activity or with τ indicating that the transition has no corresponding activity, i.e. formally, process models are system nets in which the designated final state is reachable from the initial state (see Section 6.2).

Definition 10 (Process Model) Let $A \subseteq \mathcal{A}$ be a set of activities, $N = (P, T, F, \lambda)$ a Petri net and $SN = (N, m_i, m_f)$ a system net. We call SN a process model if and only if $\mathcal{L}_{SN} \neq \emptyset$.

In contrast to general Petri nets, process models have a distinguished final marking which can be reached from the initial marking, i.e. there is a distinguished state in the model that, once reached, indicates that execution of a process instance has finished. Nearly all other process modelling languages have an explicit notion of such a final state and we exploit this later for efficient computations of alignments.

It is important to realize that we assume the final marking to be reachable from the initial marking. This assumption is very relaxed and in Petri net theory is often referred to as easy soundness. In this book, we assume all models to be easy sound, but in practice much stronger soundness notions hold for all models presented in this book.

Definition 11 (Easy Soundness) Let $A \subseteq \mathcal{A}$ be a set of activities, $N = (P, T, F, \lambda)$ a Petri net and $SN = (N, m_i, m_f)$ a system net. SN is called *easy sound* if and only if there exists a $\sigma \in \mathcal{L}_{SN}$, i.e. if there is at least one execution sequence from the initial to the final marking.

Note that, in this book, all process models are easy sound according to Definition 10.

When aligning a trace to a model, the trace is translated into a so-called *trace model*. This again is a labelled Petri net with an initial and a final marking, but without any choices as the model represents a past execution in which every decision was made.

Definition 12 (Trace Model) Let $A \subseteq \mathcal{A}$ be a set of activities, and $\sigma \in A^*$ a sequence over these activities. A trace model $TN = ((P, T, F, \lambda), m_i, m_f)$ is a system net, such that

- $P = \{p_0, \ldots, p_{|\sigma|}\}$,
- $T = \{t_1, \ldots, t_{|\sigma|}\}$,
- $F = \{(p_i, t_{i+1}) \mid 0 \leq i < |\sigma|\} \cup \{(t_i, p_i) \mid 1 \leq i \leq |\sigma|\}$,
- $m_i = [p_0]$,
- $m_f = [p_{|\sigma|}]$, and
- for all $1 \leq i \leq |\sigma|$ it holds $\lambda(t_i) = \sigma(i)$.

Figure 7.2 Trace model for the running example

The trace model defined above is most commonly a simple sequence of the transitions labelled with activities as they appeared in the trace (more precisely, as they are assigned to the events of a trace). However, the theory presented in this chapter does not rely on this fact. Instead, the requirement is that the trace model is a Petri net without choices, or more specifically, that each place has at most one incoming and at most one outgoing arc.

Figure 7.2 shows the trace model for our example trace $\langle As, Aa, Sso, Ro, Ao, Aaa, Aaa\rangle$, where we chose new identifiers for the places, different from the places in Figure 7.1. Note the two distinct transitions Aaa_1 and Aaa_2 corresponding to the two events that indicate an execution of activity *Aaa* in the trace. Formally, both these transitions are assigned the label *Aaa* denoting the respective activity. However, we use the subscript to highlight that there are two distinguished transitions that carry the same label.

Using the process and the trace model, we define the synchronous product model as the combination of the two with an additional set of synchronous transitions based on identical labels.

Definition 13 (Synchronous Product) Let $A \subseteq \mathcal{A}$ be a set of activities, $SN = ((P^{SN}, T^{SN}, F^{SN}, \lambda^{SN}), m_i^{SN}, m_f^{SN})$ a process model and $\sigma \in A^*$ a trace with its corresponding trace model $TN = ((P^{TN}, T^{TN}, F^{TN}, \lambda^{TN}), m_i^{TN}, m_f^{TN})$.

The synchronous product $SP = ((P, T, F, \lambda), m_i, m_f)$ is again a system net, such that:

- $P = P^{SN} \cup P^{TN}$ is the set of places,
- $T = (T^{MM} \cup T^{LM} \cup T^{SM}) \subseteq (T^{SN} \cup \{\gg\}) \times (T^{TN} \cup \{\gg\})$ is the set of transitions where $\gg$ denotes a new element, i.e. $\gg \notin T^{SN} \cup T^{TN}$, with
  - $T^{MM} = T^{SN} \times \{\gg\}$ (model moves),
  - $T^{LM} = \{\gg\} \times T^{TN}$ (log moves) and
  - $T^{SM} = \{(t_1, t_2) \in T^{SN} \times T^{TN} \mid \lambda^{SN}(t_1) = \lambda^{TN}(t_2)\}$ (synchronous moves),
- $F = \{(p, (t_1, t_2)) \in P \times T \mid (p, t_1) \in F^{SN} \vee (p, t_2) \in F^{TN}\} \cup \{((t_1, t_2), p) \in T \times P \mid (t_1, p) \in F^{SN} \vee (t_2, p) \in F^{TN}\}$,
- $m_i = m_i^{SN} + m_i^{TN}$,
- $m_f = m_f^{SN} + m_f^{TN}$, and
- for all $(t_1, t_2) \in T$ holds that $\lambda((t_1, t_2)) = (l_1, l_2)$, where $l_1 = \lambda^{SN}, if t_1 \in T^{SN}$, and $l_1 = \gg$ otherwise; and $l_2 = \lambda^{TN}, if t_2 \in T^{TN}$, and $l_2 = \gg$ otherwise.

The synchronous product is essentially a combination of the original process model with the trace model, in such a way that each pair of transitions that are labelled with the same activity are also represented by a special synchronous

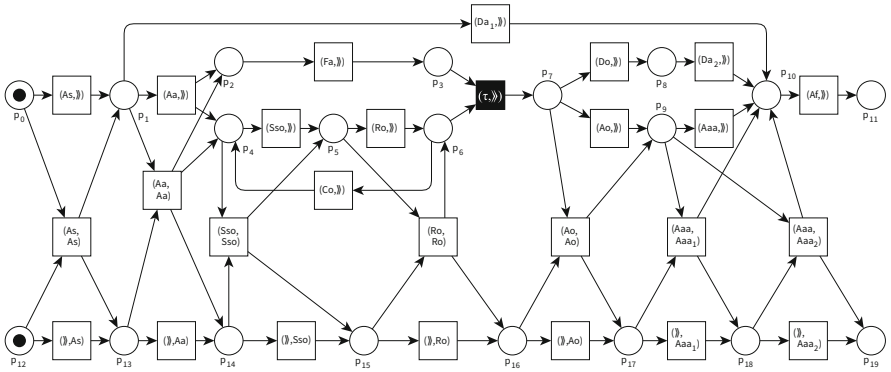


Figure 7.3 Synchronous product for the running example

transition. The transitions of the original process and trace model are represented by pairing them with the new symbol $\gg$. Figure 7.3 shows the synchronous product for our example process model and trace.

An alignment, as discussed earlier in Section 4.3, is an execution sequence of the synchronous product starting in the initial marking and ending in the final marking.

Definition 14 (Alignments) Let $A \subseteq \mathcal{A}$ be a set of activities, $\sigma \in A^*$ a trace with *TN* its corresponding trace net, *SN* a process model and $SP = ((P, T, F, \lambda), m_i, m_f)$ the synchronous product of *SN* and *TN*.

Let $\gamma \in \mathcal{L}_{SP}$ be a full execution sequence of the synchronous product. γ is called an alignment between *SN* and σ.

An alignment is a sequence of transition firings in the synchronous product. Firing a transition $(t, \gg) \in T$ corresponds to a model move. Firing a transition $(\gg, t) \in T$ corresponds to a log move and firing a transition $(t_1, t_2) \in T$ with $t_1 \neq \gg$ and $t_2 \neq \gg$ corresponds to a synchronous move. For model moves, we typically distinguish between model moves where the corresponding transition is labelled with an activity and those where the transition is labelled with τ, as the former indicates a deviation, whereas and the latter does not.

It is easy to see that there is always an alignment for any given trace and model, since there is an execution sequence in the model that does not touch any tokens in the trace model (this execution sequence exists because the model is easy sound) and the trace model can be executed by definition. Therefore, the alignment consisting of only model moves, followed by only log moves is a correct execution sequence of the synchronous product.

Proposition 1 *Let $A \subseteq \mathcal{A}$ be a set of activities, $\sigma \in A^*$ a trace with TN its corresponding trace net, SN a process model and $SP = ((P, T, F, \lambda), m_i, m_f)$ the synchronous product of SN and TN. It holds that $\mathcal{L}_{SP} \neq \emptyset$, i.e. an alignment between SN and σ exists.*

Proof Since SN is a process model, we know from Definition 10 that there exists a $\sigma^{SN} \in \mathcal{L}_{SN}$. Furthermore, we know that $\sigma^{TN} = \langle t_1, \ldots, t_{|\sigma|} \rangle \in \mathcal{L}_{TN}$. Let $\gamma = \langle (\sigma_1^{SN}, \gg), \ldots, (\sigma_{|\sigma^{SN}|}^{SN}, \gg), (\gg, \sigma_1^{TN}), \ldots, (\gg, \sigma_{|\sigma^{TN}|}^{TN}) \rangle$. It's trivial to see that $\gamma \in \mathcal{L}_{SP}$.

□

The problem of finding an alignment for a given trace and model can therefore be reformulated as finding an execution sequence from the initial to the final marking in the synchronous product (reachability). This execution sequence should be minimal with respect to a cost function c that penalises for deviations, i.e. for transitions corresponding to model and log moves.

Definition 15 (Cost Function, Default Cost Function) Let $SP = ((P, T, F, \lambda), m_i, m_f)$ be a synchronous product. A cost function $c : T \to \mathbb{R}^+ \cup \{0\}$ is a function associating a non-negative cost to each transition incurred when firing that transition.

The cost function assigning cost 0 to synchronous moves and τ-labelled transitions and cost 1 to other model moves and log moves is called the *default cost function*.

The cost function associates a non-negative cost to each transition of the synchronous product to indicate the severity of that transition firing. Recall that these transitions are partitioned into model moves, log moves and synchronous moves. Typically, synchronous moves and τ-labelled model moves are assigned cost 0. Using the cost function we define optimal alignments.

Definition 16 (Optimal Alignment) Let $A \subseteq \mathcal{A}$ be a set of activities, $\sigma \in A^*$ a trace with TN its corresponding trace net, SN a process model and SP the synchronous product of SN and TN. Furthermore, let $c : T \to \mathbb{R}^+ \cup \{0\}$ be a cost function. An optimal alignment $\gamma^{opt} \in \mathcal{L}_{SP}$ is a full execution sequence of the synchronous product, such that for all $\gamma \in \mathcal{L}_{SP}$ it holds that $c(\gamma) \geq c(\gamma^{opt})$, where $c(\gamma) = \sum_{1 \leq i \leq |\gamma|} c(\gamma(i))$.

Figure 7.4 illustrates an optimal alignment for our example, depicted on top of the synchronous product shown earlier. It shows that there are three deviations between the trace and the model, namely a model move on transition *Fa*, a log move on (here, the first) event labelled *Aaa* and a model move on the transition *Af*.

Excursion 13

Notes on the cost function

When defining optimal alignments, the cost function plays an important role. An alignment is defined as optimal with respect to a cost function $c : T \to \mathbb{R}^+ \cup \{0\}$, which associates costs to each transition in the synchronous product. By assigning non-zero costs to transitions representing log moves

(continued)

and model moves labelled with activities, and zero costs to synchronous moves and τ-labelled transitions, we guarantee that an optimal alignment favours synchronous moves over others.

However, formally speaking, associating a cost of zero to τ-labelled transitions can cause problems for the algorithms presented in this chapter, since these algorithms are only guaranteed to terminate as long as there is no marking m such that there is an infinite set of markings m' reachable from m with cost 0.

The class of models for which a cost of 0 could cause the algorithm to run indefinitely is very unlikely to appear in real life. So far, we have seen such models only as translations of so-called C-nets, the representation used in [114].

However, to be on the safe side in any generic implementation, it is wise to ensure that the cost function returns some negligible ϵ cost for τ-labelled transitions.

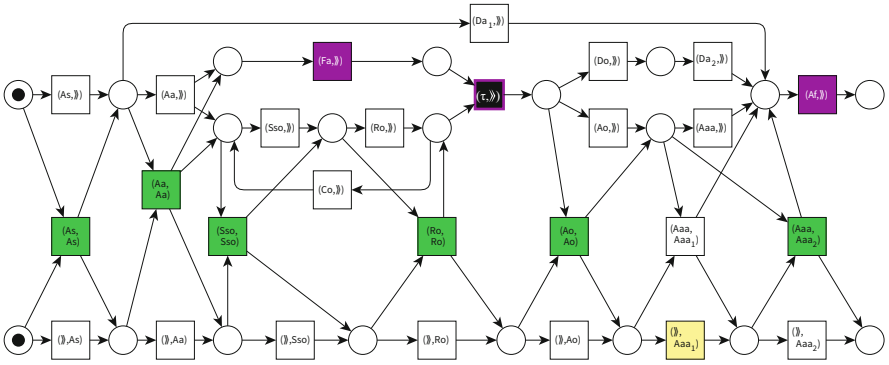


Figure 7.4 Full run of the synchronous product corresponding to an optimal alignment, assuming a default cost function

7.2 Computing Optimal Alignments

After we have formalised the problem of finding optimal alignments as identifying the cheapest (i.e., cost minimal) execution sequence of a synchronous product model according to some cost function, we present an algorithm for constructively computing optimal alignments.

The easiest solution is to build the reachability graph of the synchronous product model (see Definition 7), label each arc in that graph with the costs associated to the transition it represents, and then find the shortest path from the initial to the

final marking. However, this solution has an important drawback, namely that the reachability graph of a Petri net is not necessarily finite, hence it cannot be computed explicitly in general. Furthermore, even if it is finite, it is typically very large due to the explicit representation of all possible interleavings of parallel task executions a problem well-known as the *state space explosion problem*.

Figure 7.5 shows the full reachability graph of the synchronous product in Figure 7.3. The structure of this graph is typical for the alignment problem. There are several copies (laid out top-to-bottom, depicted in purple on the left and black elsewhere) of the reachability graph of the original process model, which correspond to model moves in an alignment. From left to right are the log moves, depicted in yellow on the top row and grey elsewhere. They do not change the state of the process model, but change the marking of the trace model and hence the state of the synchronous product. The synchronous moves (depicted in green) change both the state of the process model and the state of the trace model. The graph contains 104 states and 244 edges.

Rather than explicitly building the full reachability graph in memory, we use an incremental shortest path algorithm to compute an optimal alignment. In essence, we only build the relevant part of the reachability graph in memory. In this section, we present several variations of this algorithm each having their own pros and cons in terms of time and memory complexity.

One of the most fundamental algorithms for computing the shortest path in a graph is Dijkstra's shortest path algorithm [37]. Essentially, it considers two sets of nodes, namely the closed set A and the open set X. Furthermore, it keeps a function $d : X \to \mathbb{R}^+ \cup \{0\}$ to keep track of the shortest distance known for each node and a function $p : X \to (T \cup \{\tau\} \times X \cup \{\tau\})$ to keep track of the predecessors in the shortest path. The closed set is the set of nodes for which the shortest path from the source node is known and the open set is the set of nodes for which a path may be known, but for which it is unknown whether this is the shortest path.

A specific version of the algorithm for computing alignments is detailed in Algorithm 1. The main difference with the regular Dijkstra algorithm is the fact that new markings are generated on the fly and that the predecessor function keeps track of the transitions fired, not only the preceding marking.

To get a feeling for the workings of Dijkstra's algorithm, consider Figure 7.6. This is part of the reachability graph depicted in Figure 7.5 searched by Dijkstra's algorithm as soon as all states with distance $d(m) = 0$ have been expanded. So far, 17 markings were found and 16 edges were traversed. The set X now contains all states with distance 1 (the distance of each state $d(m)$ is written in the state). All states with distance 0 are now in the closed set A.

For our running example, Dijkstra's algorithm visits between 71 and 95 states and traverses between 128 and 180 edges. The main reason for these differences is the arbitrary way a marking is selected from the open set in line 8 of Algorithm 1. In the best case, depicted in Figure 7.7, the final marking is selected as soon as it is in the open set and it is one of the markings minimizing $d(m)$. In the worst case, depicted in Figure 7.8, it is the selected last. Still, this search space is considerably smaller than the full reachability graph of the synchronous product.

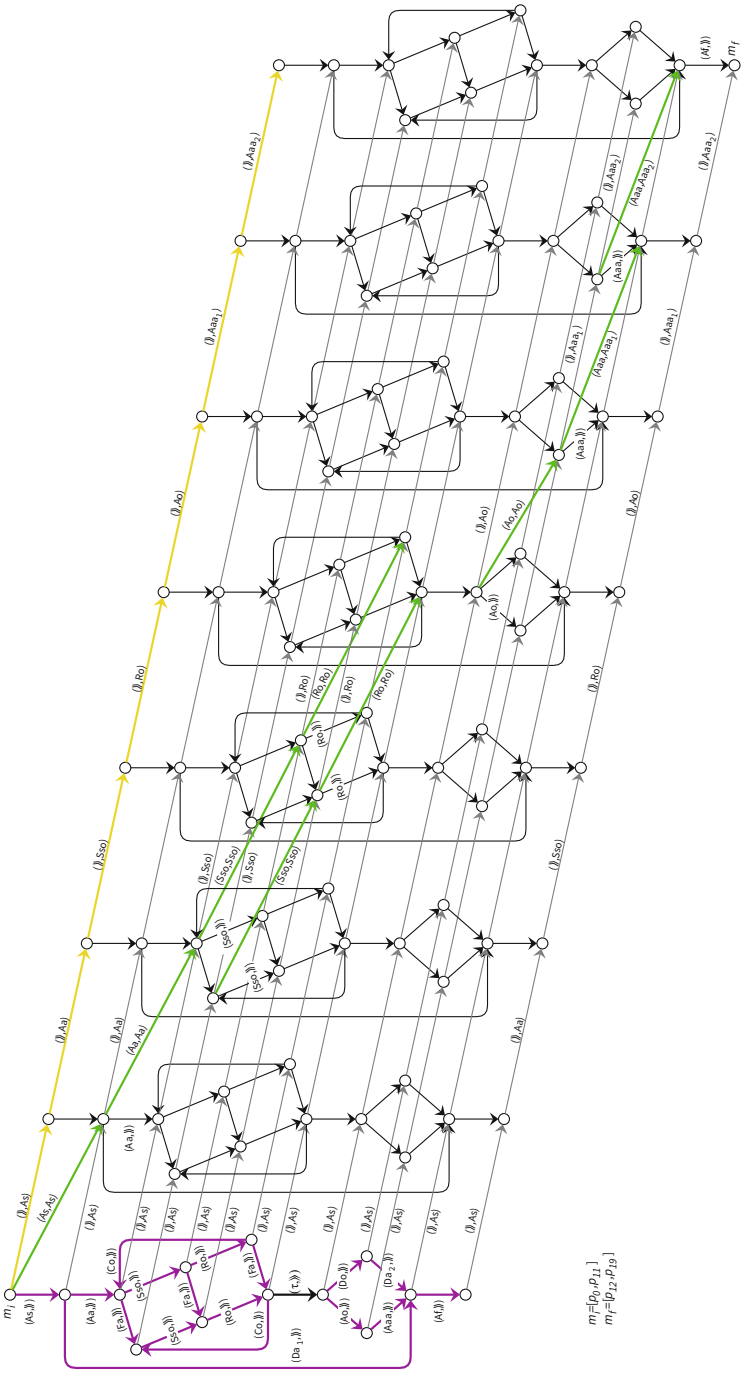


Figure 7.5 Full state space of the synchronous product for the running example

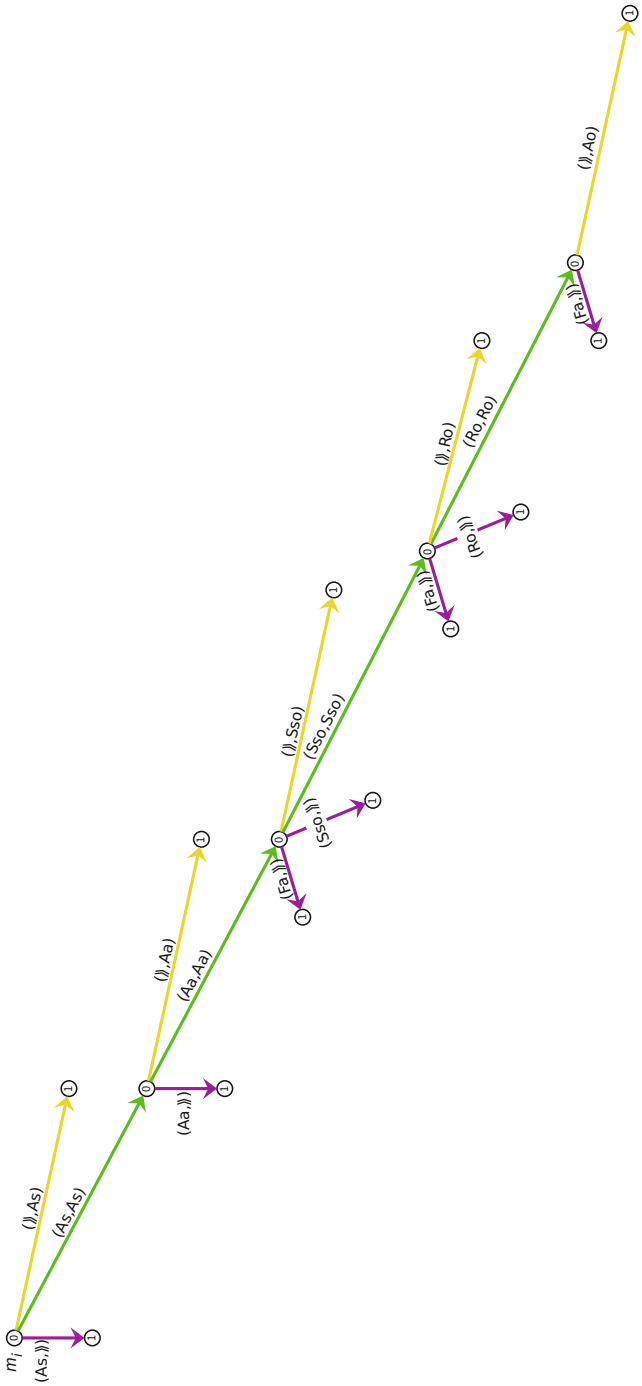


Figure 7.6 Part of the graph generated in Dijkstra's algorithm once all markings reachable with $d(m) = 0$ have been expanded

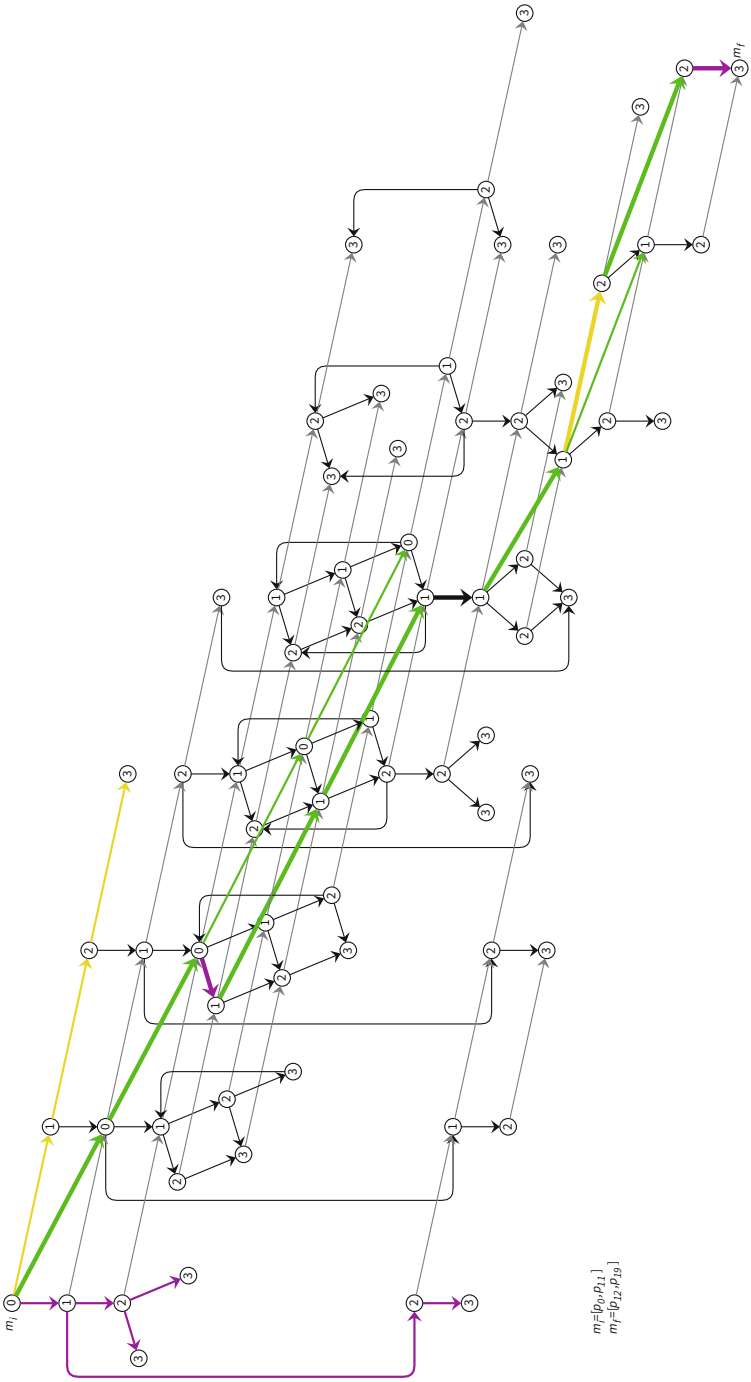


Figure 7.7 Full search space of Dijkstra's algorithm in the best case. All nodes up to distance 3 have been reached, and m_f is the first marking at distance 3 investigated

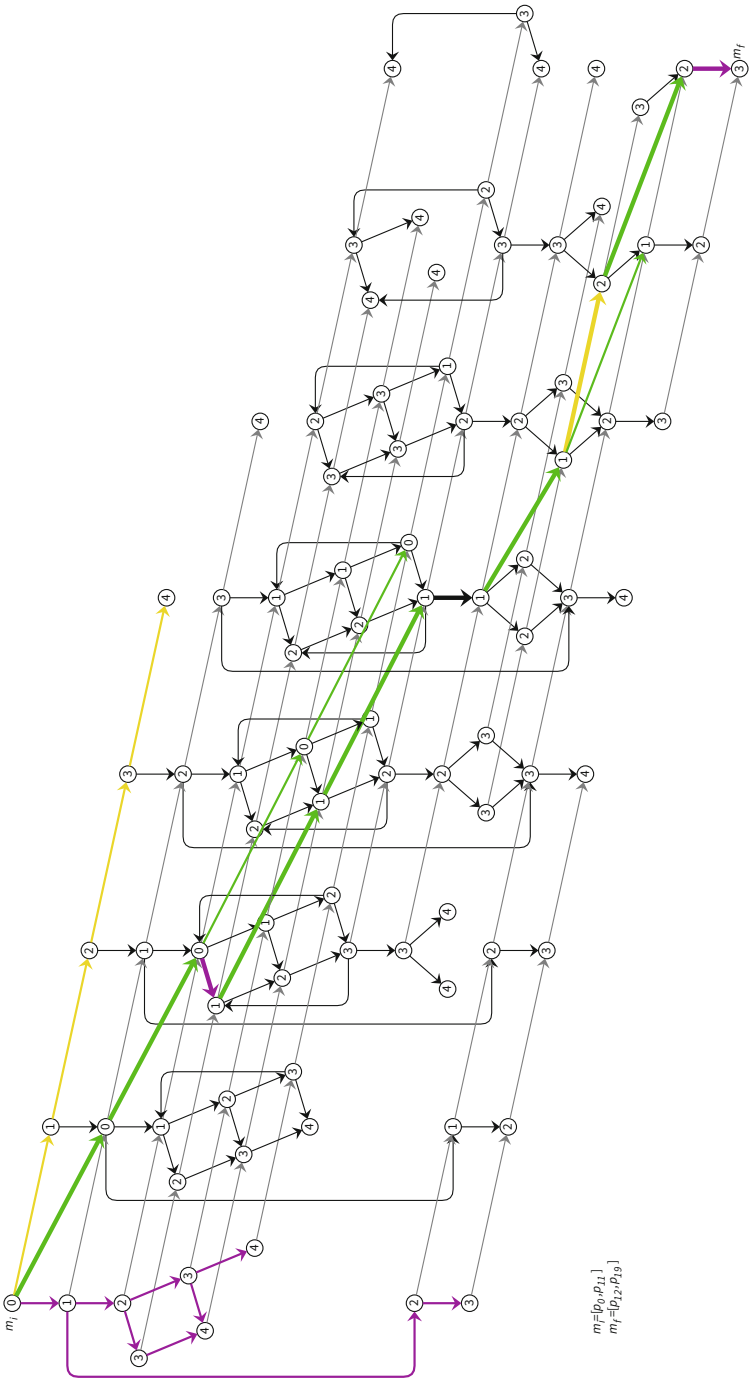


Figure 7.8 Full search space of Dijkstra's algorithm in the worst case. All nodes up to distance 4 have been reached, since m_f was the last marking at distance 3 investigated

Procedure 1 Dijkstra's algorithm for alignments

Let $SP = ((P, T, F, \lambda), m_i, m_f)$ be a synchronous product and let $c : T \to \mathbb{R}^+ \cup \{0\}$ be a cost function.

function DIJKSTRA(SP, c)
  $A \leftarrow \emptyset$ ▷ Initialise closed set
  $X \leftarrow \{m_i\}$ ▷ Initialise open set
  $p(m_i) = (\tau, \tau)$ ▷ Initialise predecessor function
  $\forall_{m \in \mathsf{RS}(SN)}\ d(m) = \infty$ ▷ Initialise function d
  $d(m_i) = 0$
  while $X \neq \emptyset$ **do** ▷ While not all states visited
    $m \leftarrow m \in X$ minimizing $d(m)$ ▷ Get the marking m closest to m_i
    if $m = m_f$ **then** ▷ final marking reached
      break while
    end if
    $A \leftarrow A \cup \{m\}$ ▷ Add m to the closed set
    $X \leftarrow X \setminus \{m\}$ ▷ Remove m from the open set
    for all $t \in T$ with $m[t\rangle m'$ **do** ▷ For each transition enabled in m
      if $m' \notin A$ **then** ▷ Reaching a marking not yet visited
        $X \leftarrow X \cup \{m'\}$ ▷ Add m' to the open set
        $a \leftarrow d(m) + c(t)$ ▷ Compute the current cost of reaching m'
        if $a < d(m')$ **then** ▷ If this current cost is better than known cost
          $d(m') \leftarrow a$ ▷ Update distance function
          $p(m') \leftarrow (t, m)$ ▷ Update predecessor function
        end if
      end if
    end for
  end while
  $\gamma \leftarrow \langle t_0, \ldots, t_n \rangle$ such that $t_n = \#_1(p(m_f))$, $t_{n-1} = \#_1(p(\#_2(p(m_f))))$ etc. until the initial marking is reached recursively.
  return $d(m_f), \gamma$ ▷ Return distance and alignment
end function

The alignment found is highlighted in both Figures 7.7 and 7.8 using bold lines and this corresponds to:

log trace	As	Aa	$\gg$	Sso	Ro	$\gg$	Ao	Aaa	Aaa	$\gg$
execution sequence	As	Aa	Fa	Sso	Ro	τ	Ao	$\gg$	Aaa	Af

Dijkstra's shortest path algorithm will terminate eventually if one of two conditions is met, namely: (1) the number of reachable markings is finite, or (2) the number of markings reachable from any marking with cost 0 is finite. The proof for this claim is straightforward as we already have a known path to the target state (the witness alignment of Proposition 1). Under the assumptions above, any (infinite) path investigated will eventually become more costly than this witness alignment.

More advanced path-finding algorithms exist which can be used to find shortest paths. In this book, we limit ourselves to $A^\star$ which has become the de-facto standard for computing alignments. Mainly because, as in the case of Dijkstra's algorithm, it does not require the graph to be built explicitly in memory.

7.3 Efficiently Computing Optimal Alignments

The $A^\star$ shortest path algorithm [53] is a generalization of Dijkstra's algorithm. In each iteration, the selection of the element from the set of open nodes is not based on the distance travelled so far, but based on a combination of the distance travelled so far and an *underestimate* of the remaining distance. If the remaining distance to travel can be underestimated with some accuracy, the $A^\star$ algorithm visits considerably fewer vertices in the graph to find a shortest path as it will not explore parts of the graph that inevitably lead to longer paths.

In short, the $A^\star$ algorithm is derived from Dijkstra's algorithm. However, rather than selecting a marking m that minimizes $d(m)$ in line 8 of Algorithm 1, we select a marking m that minimizes $f(m)$ in line 10 of Algorithm 2. The function $f(m) = g(m) + h(m)$ takes into consideration the *estimated* remaining cost of the alignment $h(m)$ on top of the known distance $g(m)$.

The $A^\star$ algorithm depends heavily on the function h. For $A^\star$ to return an optimal result, the function h needs to be *admissible*, i.e. for any reachable state m it holds that $h(m) \leq h^*(m)$, where $h^*(m)$ is the shortest distance from m to the final marking m_f. In other words, h *underestimates* the remaining costs to reach the target marking. Furthermore, the better the quality of the underestimation, the fewer nodes are expanded.

To get a feeling for the $A^\star$ algorithm, consider Figure 7.9 which, like before in Figure 7.6, shows the search space of the algorithm after all nodes at distance 0 have been expanded from the initial state of the running example. In contrast to Dijkstra however, now all nodes also have an associated estimated remaining distance, indicated by the h function. Since the nodes in the open set are prioritized based on the sum of the distance so far and the estimated remaining distance, one of the three highlighted nodes will be investigated next, while in Dijkstra's case all nodes at distance 1 would be equally likely to be investigated.

Algorithm 2 assumes that the underestimation function h is *monotonic*, i.e. for any two markings m, m' and transition t with $m[t\rangle m'$, it holds that $h(m) \leq c(t) + h(m')$. It is not too hard to see that in Figure 7.9 this is indeed the case. The trivial function $h_d(x) = 0$ satisfies this requirement, in which case $A^\star$ essentially becomes equal to Dijkstra's algorithm.

In the remainder of this section, we discuss the use of the Petri net marking equation as a heuristic functions as well as general techniques to reduce the search space.

One of the most important features of the $A^\star$ algorithm is that it uses an underestimation function h to guide the search. This underestimation function should return a cost "as close as possible" to the actual remaining costs. As our input models are Petri nets, we can make use of the marking equation. As introduced in Section 6.2.1, the later is a system of linear equations, such that each variable represents the number of firings of a specific transition. Furthermore, given a sequence of transition firings from marking m to marking m', the Parikh vector representation of this sequence is a solution to the marking equation. In other words, any sequence of transition firings in the synchronous product yields an integer

Procedure 2 $A^\star$ algorithm for alignments

Let $SP = ((P, T, F, \lambda), m_i, m_f)$ be a synchronous product and let $c : T \to \mathbb{R}^+$ be a cost function and $h : \mathsf{RS} \to \mathbb{R}^+$ be an heuristic underestimating the cost of getting from any marking to the final marking.

function ASTAR(SP, O, c)
  $A \leftarrow \emptyset$ ▷ Initialise closed set
  $X \leftarrow \{m_i\}$ ▷ Initialise open set
  $p(m_i) = (\tau, \tau)$ ▷ Initialise predecessor function
  $\forall_{m \in \mathsf{RS}(SN)}\ g(m) = \infty$ ▷ Initialise cost so far function g
  $g(m_i) = 0$
  $\forall_{m \in \mathsf{RS}(SN)}\ f(m) = \infty$ ▷ Initialise estimated total cost function f
  $f(m_i) = h(m_i)$ ▷ Compute estimate for initial marking
  while $X \neq \emptyset$ **do** ▷ While not all states visited
    $m \leftarrow m \in X$ minimizing $f(m)$ ▷ Get the most promising marking m
    if $m = m_f$ **then** ▷ final marking reached
      break while
    end if
    $A \leftarrow A \cup \{m\}$ ▷ Add m to the closed set
    $X \leftarrow X \setminus \{m\}$ ▷ Remove m from the open set
    for all $t \in T$ with $m[t\rangle m'$ **do** ▷ For each transition enabled in m
      if $m' \notin A$ **then** ▷ Reaching a marking not yet visited
        $X \leftarrow X \cup \{m'\}$ ▷ Add m' to the open set
        $a \leftarrow g(m) + c(t)$ ▷ Compute the cost so far of reaching m' via m
        if $a < g(m')$ **then** ▷ If this current cost is better than known cost so far
          $g(m') \leftarrow a$ ▷ Update cost so far function
          $f(m') \leftarrow g(m') + h(m')$ ▷ Update estimated total cost function
          $p(m') \leftarrow (t, m)$ ▷ Update predecessor function
        end if
      end if
    end for
  end while
  $\gamma \leftarrow \langle t_0, \ldots, t_n \rangle$ such that $t_n = \#_1(p(m_f))$, $t_{n-1} = \#_1(p(\#_2(p(m_f))))$ etc. until the initial marking is reached recursively.
  return $f(m_f), \gamma$ ▷ Return distance and alignment
end function

solution to the marking equation and the cost function over this sequence is the cost of the alignment. Now, if we simply solve the linear equation system while minimizing our cost function as a target function, we also get a solution which cannot have higher costs than the actual path. Hence, the marking equation provides an underestimate.

Definition 17 (Underestimation Function h) Let $SP = ((P, T, F, \lambda), m_i, m_f)$ be a system net with incidence matrix $\mathbf{C}$ and let $c : T \to \mathbb{R}^+ \cup \{0\}$ be a cost function. We define the underestimation function $h : \mathsf{RS}(SP) \to \mathbb{R}^+$ underestimating the

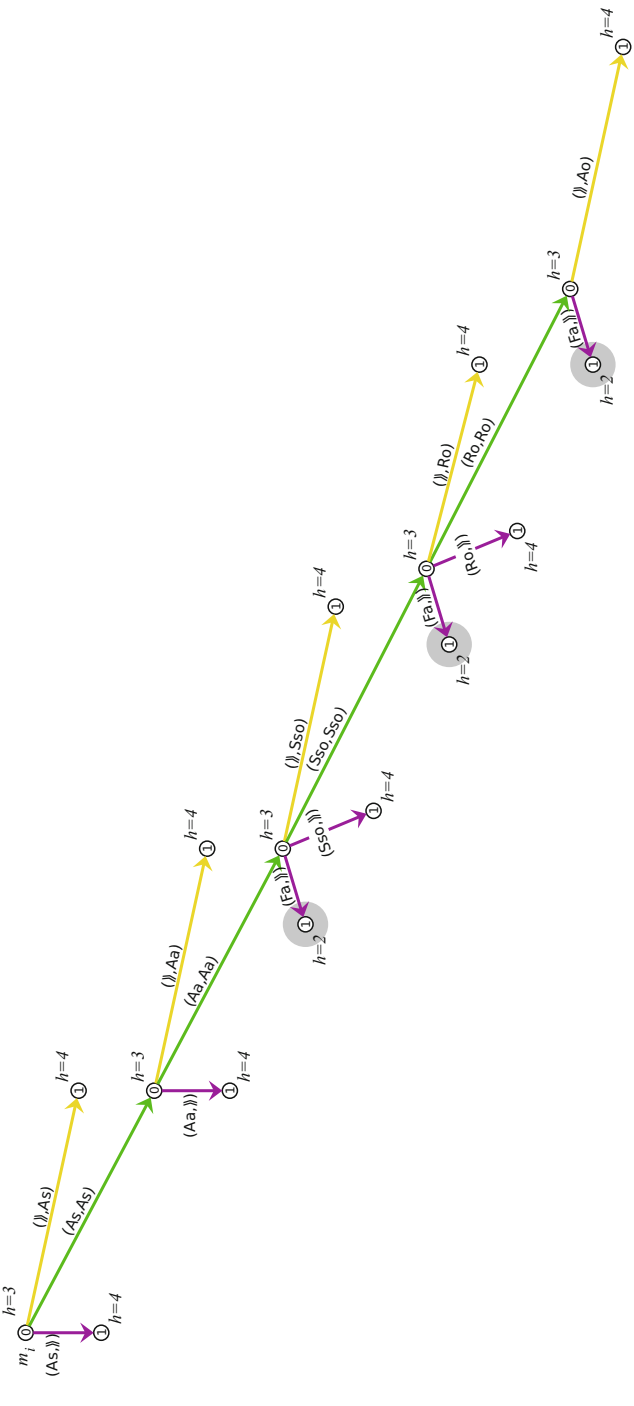


Figure 7.9 Part of the graph generated by the $A^\star$ algorithm once all markings reachable with $g(m) = 0$ have been expanded. The three highlighted markings form the head of the priority queue as they all have the same f-score

cost of reaching the final marking from any marking as follows:

$$h(m) = \begin{cases} \infty & \nexists_{\vec{x}} \ \vec{m} + \mathbf{C} \cdot \vec{x} = \vec{m_f} \\ c(\vec{x}) & \text{where } \vec{x} \text{ is the solution to} \end{cases} \qquad \begin{array}{ll} \textbf{min} & c(\vec{x}) \\ \textbf{subject to} & \vec{m} + \mathbf{C} \cdot \vec{x} = \vec{m_f} \\ & \vec{x} \geq \vec{0} \end{array}$$

We use h^I to indicate that we solve the system of linear equations for integer variables and h^R to indicate that we solve the system with real variables. Whenever we use h without superscript, both variants can be substituted.

It is easy to see that the function h indeed underestimates the cost of reaching the final marking.

Proposition 2 (Underestimation Function h Is Admissible) *Let $SP = ((P, T, F, \lambda), m_i, m_f)$ be a system net with incidence matrix $\mathbf{C}$ and let $c : T \to \mathbb{R}^+$ be a cost function and $h : \mathit{RS}(SP) \to \mathbb{R}^+$ the underestimation function following Definition 17. We prove that for any marking $m \in \mathit{RS}(SP)$ and for all $\sigma \in T^*$ with $m[\sigma\rangle m_f$ it holds that $h(m) \leq \sum_{t \in \sigma} c(t)$.*

Proof Since $m[\sigma\rangle m_f$, we know that $\vec{m} + \mathbf{C} \cdot \vec{\sigma} = \vec{m_f}$. Hence for any $\vec{x}$ minimizing $c(\vec{x})$ such that $\vec{m} + \mathbf{C} \cdot \vec{x} = \vec{m_f}$ it holds that $c(\vec{x}) \leq c(\vec{\sigma})$. Hence $h(m) \leq c(\vec{\sigma}) = \sum_{t \in \sigma} c(t)$.

□

Not only does the marking equation provide an underestimate for the remaining cost, it also provides a monotonic underestimation function.

Proposition 3 (Underestimation Function h Is Monotonic) *Let $SP = ((P, T, F, \lambda), m_i, m_f)$ be a system net with incidence matrix $\mathbf{C}$ and let $c : T \to \mathbb{R}^+$ be a cost function and $h : \mathit{RS}(SP) \to \mathbb{R}^+$ the underestimation function following Definition 17. We prove that for any marking $t \in T$ and $m, m' \in \mathit{RS}(SP)$ with $m[t\rangle m'$ it holds that $h(m) \leq c(t) + h(m')$.*

Proof We prove this by contradiction. Let m, m' and t be such that $h(m) > c(t) + h(m')$. Let $\vec{x}'$ be the solution to $\vec{m}' + \mathbf{C} \cdot \vec{x'} = \vec{m_f}$ such that $h(m') = c(\vec{x}')$. From the definition of the firing rule, we know that $\vec{m} + \mathbf{C} \cdot \vec{1}_t = \vec{m}'$, hence we know that $\vec{m} + \mathbf{C} \cdot \vec{1}_t + \mathbf{C} \cdot \vec{x'} = \vec{m_f}$. Let $\vec{x} = \vec{x}' + \vec{1}_t$, then $\vec{m} + \mathbf{C} \cdot \vec{x} = \vec{m_f}$ and $c(\vec{x}) = c(\vec{x}') + c(t) = c(t) + h(m')$.

Since $\vec{x}$ is a solution to $\vec{m} + \mathbf{C} \cdot \vec{x} = \vec{m_f}$, we know that $h(m) \leq c(\vec{x}) = c(t) + h(m')$, which contradicts the assumption.

□

Combining Proposition 2 and Proposition 3, we have shown how the marking equation can be instantiated to obtain an admissible, monotonic underestimation function for the remaining costs of an alignment. Furthermore, since the only requirement of the estimation function is that it is an underestimate, there is no need

to solve the equation system for integer variables. Instead, we can use real variables which also provide an underestimate. This might, however, worsen the quality of the solution which causes the $A^\star$ to investigate more states.

The heuristics depicted earlier in Figure 7.9 were computed using h^R. The matrix $\mathbf{C}$ has 20 rows and 27 columns, corresponding to the incidence matrix of the Petri net in Figure 7.3, and the solution found corresponds to the vector $[(As, As), (Aa, Aa), (Sso, Sso), (Ro, Ro), (Fa, \gg), (\tau, \gg), (Ao, Ao), \frac{1}{2}(Aaa, Aaa_1), \frac{1}{2}(Aaa, Aaa_2), \frac{1}{2}(\gg, Aaa_1)\frac{1}{2}(\gg, Aaa_2), (Af, \gg)]$. Note that, in general, the solution found is not guaranteed to correspond to a realizable sequence. In our example it even contains non-integer parts. Please note that this vector is highly unlikely to be the result produced by any (I)LP solver. However, it is an optimal solution which we use for illustrative purposes. It is much more likely that the LP solver returns the vector $[(As, As), (Aa, Aa), (Sso, Sso), (Ro, Ro), (Fa, \gg), (\tau, \gg), (Ao, Ao), (Aaa, Aaa_1), (\gg, Aaa_2), (Af, \gg)]$ or $[(As, As), (Aa, Aa), (Sso, Sso), (Ro, Ro), (Fa, \gg), (\tau, \gg), (Ao, Ao), (\gg, Aaa_1), (Aaa, Aaa_2), (Af, \gg)]$, which both have the same deviation cost 3.

Figure 7.8 shows the full search space expanded by $A^\star$ in the worst case. For each marking reached, the heuristic function is depicted. It is clear that $A^\star$ explores far fewer states than Dijkstra (37 instead of 95 in the worst case), and traverses fewer edges (41 instead of 180 in the worst case). However, a total of 36 heuristics have been computed for $A^\star$, each of which corresponds to a system of (integer) linear equations (note that in the final marking no heuristic needs to be computed; it's trivially 0).

The alignment found in this case is different from the one found by Dijkstra as the order in which certain moves are identified is different. The actual alignment highlighted in Figure 7.8 is:

log trace	As	Aa	Sso	Ro	$\gg$	$\gg$	Ao	Aaa	Aaa	$\gg$
execution sequence	As	Aa	Sso	Ro	Fa	τ	Ao	Aaa	$\gg$	Af

Solving a linear programming problem in each state of the search space is rather computationally intensive (recall that the heuristic is used in line 22 of Algorithm 2, hence for each visited marking and for each enabled transition). In the next section, a number of optimizations are introduced that can be employed to (1) reduce the number of linear equations solved and (2) reduce the number of states visited.

7.4 Optimizing $A^\star$ for Alignments

Finding alignments is a complex and time-consuming task, especially since (1) due to the parallelism of Petri nets, many possible sequences may exist with equal cost and, (2) in each state of the search space an (integer) linear program needs to be solved. In this section, we introduce a few optimizations to improve the performance of $A^\star$ in the context of alignments.

7.4.1 Sorting Log and Model Moves

The $A^\star$ algorithm presented is a graph-search algorithm that searches the shortest path through the state space of the synchronous product. From the structure of the synchronous product, it is clear that the model moves and log moves are independent, i.e. if an optimal alignment contains a sequence of model moves and log moves without any synchronous moves in between, these can be interleaved arbitrarily. We can exploit this fact and force the $A^\star$ algorithm to only consider subsequences of model moves followed by log moves or vice versa in between any two synchronous moves.

Proposition 4 (Log and Model Moves Can Be Sorted) *Let $SP = ((P, T, F, \lambda), m_i, m_f)$ be a synchronous product. Let $\gamma \in T^*$ be a full execution sequence of SP. We show that if $\gamma = \gamma_1 \cdot \langle (\gg, t^{LM}), (t^{MM}, \gg), \rangle \cdot \gamma_2$, then $\gamma_1 \cdot \langle (t^{MM}, \gg), (\gg, t^{LM}), \rangle \cdot \gamma_2$ is also a full execution sequence of SP and vice versa.*

Proof From the definition of a synchronous product, it is easy to see that $\bullet(\gg, t^{LM}) \cap \bullet(t^{MM}, \gg) = \emptyset$ and $(\gg, t^{LM}) \bullet \cap (t^{MM}, \gg)\bullet = \emptyset$, i.e. the transitions corresponding to log and model moves do not share any input places or output places. Hence, their firing order can be reversed without consequences.

□

Sorting model and log moves in an alignment can speed up the search, as fewer edges have to be explored in the underlying search space. The change needed in the algorithm is rather trivial, influencing only the selection of enabled transitions based on the last transition fired on the shortest path to reach the current marking. If we want to sort model moves first, we do not allow a model move to occur after a log move. If we want to sort log moves first, we do not allow a log move after a model move.

The sorting of log and model moves has the greatest effect if there are longer sequences of log and model moves in the optimal alignment. In our example, the only combination of a model move followed by a log move or vice versa in an optimal alignment is at the end, where the model move $(Af, \gg)$ and the log move $(\gg, Aaa_2)$ can follow each other directly. For our example, this optimization would reduce the search space by one state and a few edges at best.

7.4.2 Breaking Ties in the Priority Queue

The essential difference between Dijkstra's algorithm and $A^\star$ is the order in which markings are investigated. In both cases however, markings are taken from an open set X minimizing a specific function (function f in the case of $A^\star$). It is however possible to increase the efficiency of $A^\star$ by introducing second-order sorting criteria in the set X, which is usually implemented as a priority queue.

Primarily, this queue is sorted based on the total cost function f, i.e. the sum of the known cost so far, g, and the estimated remaining cost, h. If two nodes have the same total cost, the sorting is arbitrary.

However, a second-order sorting criterion is the cost function g. By preferring higher cost so far over lower costs, the $A^\star$ first expands nodes that correspond to longer prefixes of alignments, i.e. it essentially favours markings which have already incurred the costs that are inevitable and hence it tries to get to the target state more quickly. Especially when larger parallel sections of the model are not recorded in the event log and therefore have to be considered as moves on model transitions, this avoids the investigation of all possible interleavings of these transitions.

Interestingly, the alignment shown in Figure 7.11 is again the alignment found by Dijkstra's algorithm earlier, namely:

log trace	As	Aa	$\gg$	Sso	Ro	$\gg$	Ao	Aaa	Aaa	$\gg$
execution sequence	As	Aa	Fa	Sso	Ro	τ	Ao	$\gg$	Aaa	Af

Figure 7.11 shows the worst case evolution of the $A^\star$ algorithm when the selection of a marking from the open set X is not only based on minimizing function f, but also considers as second-order criteria maximizing function g. Compared to Figure 7.10, the number of states visited is reduced from 37 to 31 and the number of edges from 41 to 31.

Sorting log and model moves and breaking ties in the priority queue reduces the number of states and edges expanded by $A^\star$. However, for each state, we still need to compute a heuristic, which is also time consuming. Therefore, we now look at ways to reduce the number of (integer) linear programs solved.

7.4.3 Reusing Solutions to Linear Programs

On the one hand, the computational complexity of alignments is in the size of the search space, as the state space of any Petri net is exponential in the size of the model itself. A second element that makes alignment computations time consuming is the linear programming problems needed for the heuristic function presented in Definition 17.

The number of (integer) linear programs that need to be solved can be reduced by storing not only the value of the estimated remaining cost $h(m)$ for each marking m, but also the solution $\vec{x}$ that reaches this value. This stored solution can be used to derive a new solution for each transition enabled at m in the search space for which the corresponding element in $\vec{x}$ is greater than or equal to 1.

Proposition 5 (Solutions to the Marking Equation Can Be Reused) *Let* $SP = ((P, T, F, \lambda), m_i, m_f)$ *be a system net with incidence matrix* $\mathbf{C}$ *and let* $c : T \to \mathbb{R}^+$ *be a cost function, and* $h : RS(SP) \to \mathbb{R}^+$ *the underestimation function following Definition 17.*

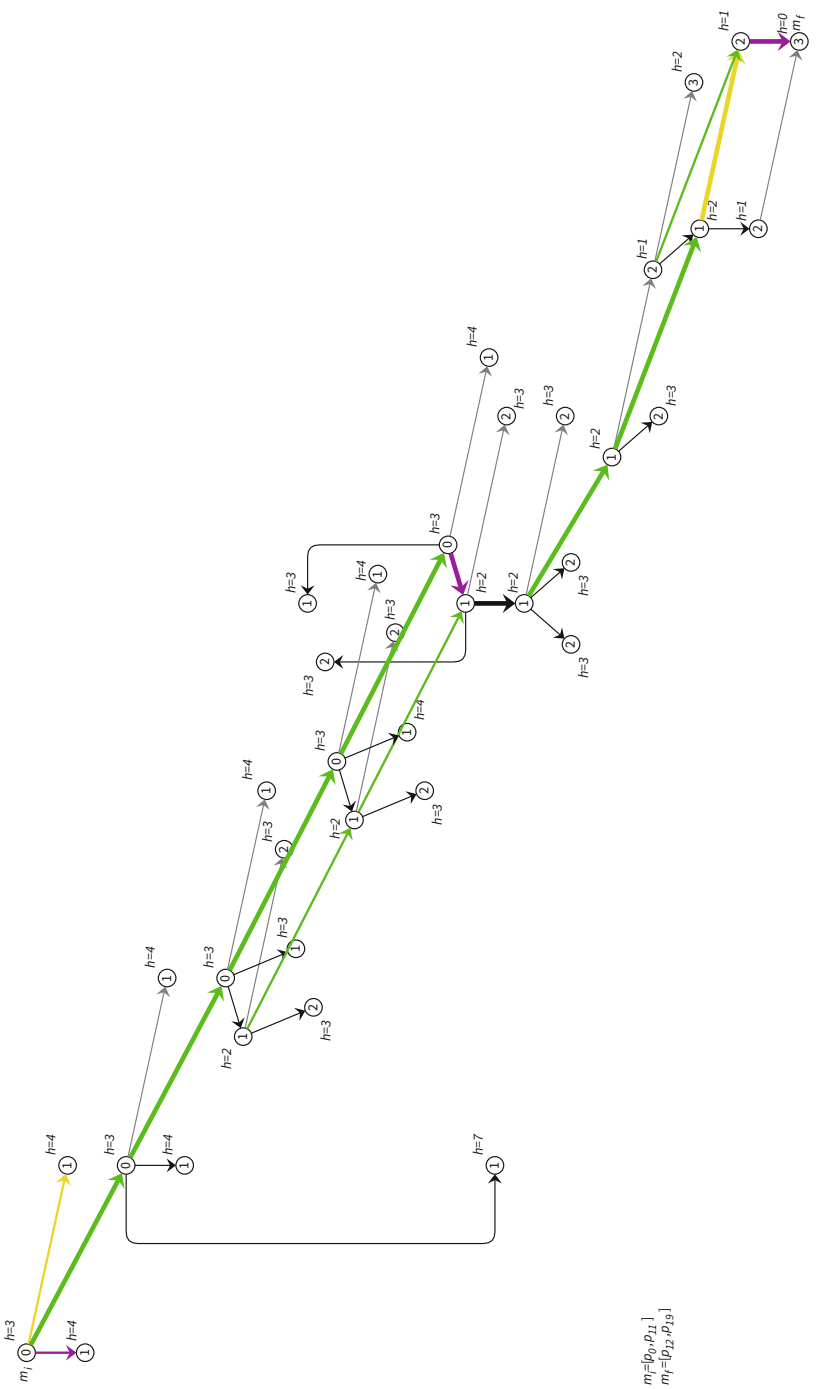


Figure 7.10 Worst case evolution of $A^\star$ on our example

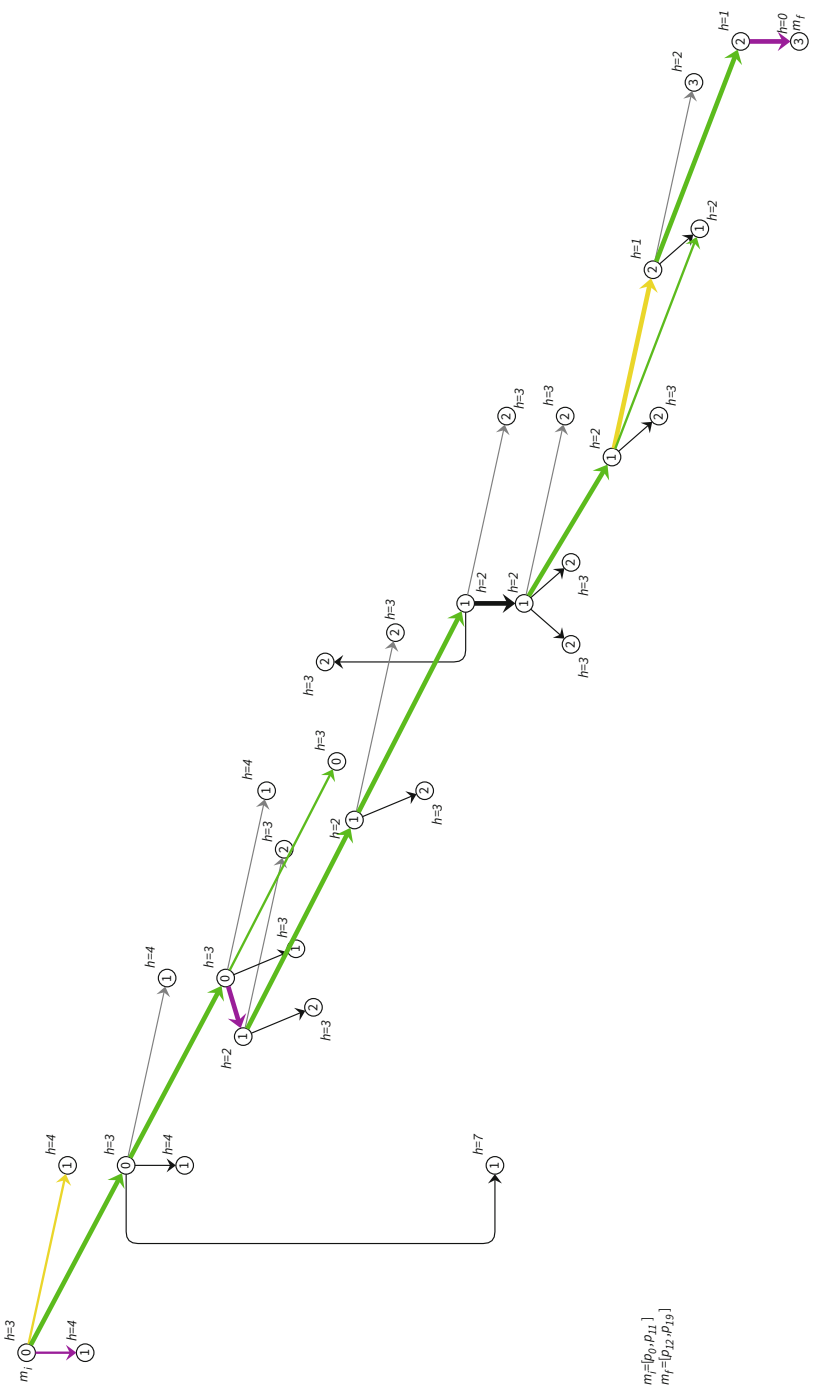


Figure 7.11 Worst case evolution of $A^\star$ on our example when the open set X uses function g as a second-order sorting criterion

We prove that for any transition $t \in T$ and markings $m, m' \in RS(SP)$ with $m[t\rangle m'$ and $\vec{x}$ a solution corresponding to $h(m)$ it holds that if $\vec{x}(t) \geq 1$ then $\vec{x}' = \vec{x} - \vec{1}_t$ is a solution corresponding to $h(m')$, i.e. $h(m') = c(\vec{x}') = h(m) - c(t)$.

Proof Since $m[t\rangle m'$, we know that $\vec{m} + \mathbf{C} \cdot \vec{1}_t = \vec{m}'$. Combining this with $\vec{m} + \mathbf{C} \cdot \vec{x} = \vec{m_f}$ yields $\vec{m}' + \mathbf{C} \cdot \vec{x} - \mathbf{C} \cdot \vec{1}_t = \vec{m_f}$. Since $\vec{x}(t) \geq 1$, we know $\vec{x}'(t) \geq 0$, hence $\vec{x}' = \vec{x} - \vec{1}_t$ is a solution to the marking equation and $h(m') \leq c(\vec{x}')$.

Assume $h(m') < c(\vec{x}')$ and let $\vec{y}'$ be the corresponding solution such that $\vec{m}' + \mathbf{C} \cdot \vec{y}' = \vec{m_f}$ and $h(m') = c(\vec{y}')$. From before, we know that $\vec{m} + \mathbf{C} \cdot \vec{1}_t + \mathbf{C} \cdot \vec{y}' = \vec{m_f}$. Let $\vec{y} = \vec{1}_t + \vec{y}'$. We know $c(\vec{y}) = c(\vec{y}') + c(t)$, hence $c(\vec{y}) < c(\vec{x}') + c(t) = c(\vec{x}' + \vec{1}_t) = c(\vec{x})$.

However, if $c(\vec{y}) < c(\vec{x})$, then x is not a solution corresponding to $h(m)$, hence $h(m') \geq c(\vec{x}')$

Combining the two cases yields $h(m') = c(\vec{x}')$, with $\vec{x}' = \vec{x} - \vec{1}_t$ the corresponding solution. □

Consider again Figure 7.9. If the heuristic in the initial marking is based on the vector $[(As, As), (Aa, Aa), (Sso, Sso), (Ro, Ro), (Fa, \gg), (\tau, \gg), (Ao, Ao), \frac{1}{2}(Aaa, Aaa_1), \frac{1}{2}(Aaa, Aaa_2), \frac{1}{2}(\gg, Aaa_1), \frac{1}{2}(\gg, Aaa_2), (Af, \gg)]$, then the heuristics in all states at distance 0 as well as the three highlighted states can be derived from this solution, indicated by the h being in italics in the figure. Hence, rather than solving all 17 (integer) linear programs, only 10 are actually solved and seven are derived. If the initial solution is based on the integer vector $[(As, As), (Aa, Aa), (Sso, Sso), (Ro, Ro), (Fa, \gg), (\tau, \gg), (Ao, Ao), (Aaa, Aaa_1), (\gg, Aaa_2), (Af, \gg)]$ instead, no other solution has to be computed as this vector corresponds to a realizable firing sequence in the synchronous product.

For the search spaces in Figure 7.10 and Figure 7.11, 10 to 11 solutions can be derived rather than computed if we store the solution vectors (10 if the original solution vector is non-integer, 11 if it is integer).

7.4.4 Estimating the Heuristic

Rather than computing heuristics for each state for which no solution can be derived from the previous one, we can delay the computation of the heuristic even further by using an estimated heuristic function.

Recall that in Proposition 5, we showed that any solution to the marking equation can be reused to compute new solutions for other markings, as long as the variable corresponding the transition executed was greater than or equal to 1. In this section, we show how to handle the case when the corresponding variable is less than 1 (i.e. 0 in the integer setting).

Suppose from some marking m, by firing transition t, we reach marking m'. We know that $\vec{x}$ is the solution corresponding to $h(m)$ and unfortunately $\vec{x}_t < 1$.

Rather than computing $h(m')$ immediately, we rely on the monotonicity of h and we compute $\hat{h}(m') = h(m) - c(t) \leq h(m')$. We then replace line 22 of Algorithm 2 by $f(m') \leftarrow g(m') + \hat{h}(m')$ and keep track of the fact that for marking m', the heuristic is not known yet, but is only an estimate.

The fact whether h is known or only an estimate is then used as a second-order sorting criterion in the priority queue, favouring markings for which the estimate is actually known, and only if the head of the priority queue is a marking m with an estimated heuristic, we compute the true heuristic, i.e. $h(m)$, and we requeue the marking m. The sorting based on function g becomes a third-order sort criterion in this setting.

Consider Figure 7.12. This figure shows part of the graph expanded by the $A^\star$ algorithm until the first time that for *none* of the markings minimizing f in the priority queue, the true heuristic h is known. The states currently minimizing f in the priority queue are highlighted. They all are at distance 2 and have an estimated heuristic $\hat{h} = 1$. All other states in the queue have the same f score ($f = 3$), but they have a lower g score ($g \leq 1$), hence they are not at the head of the priority queue.

At this stage, since all highlighted states have an equal value for g, any of them is equally likely to be selected for expansion. Since the heuristic is estimated, the algorithm then computes the true heuristic h and requeues the state if $h > \hat{h}$. In the worst case, this means that for all the highlighted states, an (integer) linear program needs to be solved. If for any state m it holds that $h(m) = \hat{h}(m)$, the $A^\star$ algorithm can continue with state m.

It is important to realize that in Figure 7.12, more states have been expanded than before in Figure 7.9. However, as these are all states with derived solutions for h, the overhead of this expansion is relatively small.

Figure 7.13 shows the worst case evolution of $A^\star$ with estimated heuristics. For all 11 highlighted nodes an (integer) linear program was solved and all states were queued again, except for the rightmost marking labelled m in Figure 7.12. For this marking it holds that $h(m) = \hat{h}(m)$ and the solution vector for $h(m)$ was used to expand the search space to the final marking. If this node would have been selected first coincidently, then only two linear programs needed to be computed to find an optimal alignment, one in marking m_i and one in marking m.

Interestingly, the use of estimated heuristics ensures that if the solution to the first (integer) linear program computed in the search corresponds to a realizable execution sequence of the model, then no other linear program is ever computed for that trace as markings that are at the head of the priority queue are always true solutions derived from this first linear program, i.e. if an integer solution would have been obtained for the initial marking m_i, then for this model solving only one linear program would have been required.

The final algorithm for computing alignments, incorporating all the improvements reported in this chapter, is described as Algorithm 3.

Table 7.1 shows an overview of the different algorithms presented in this section. It shows how many states are expanded, how many edges are traversed and how many estimates are computed, derived or estimated.

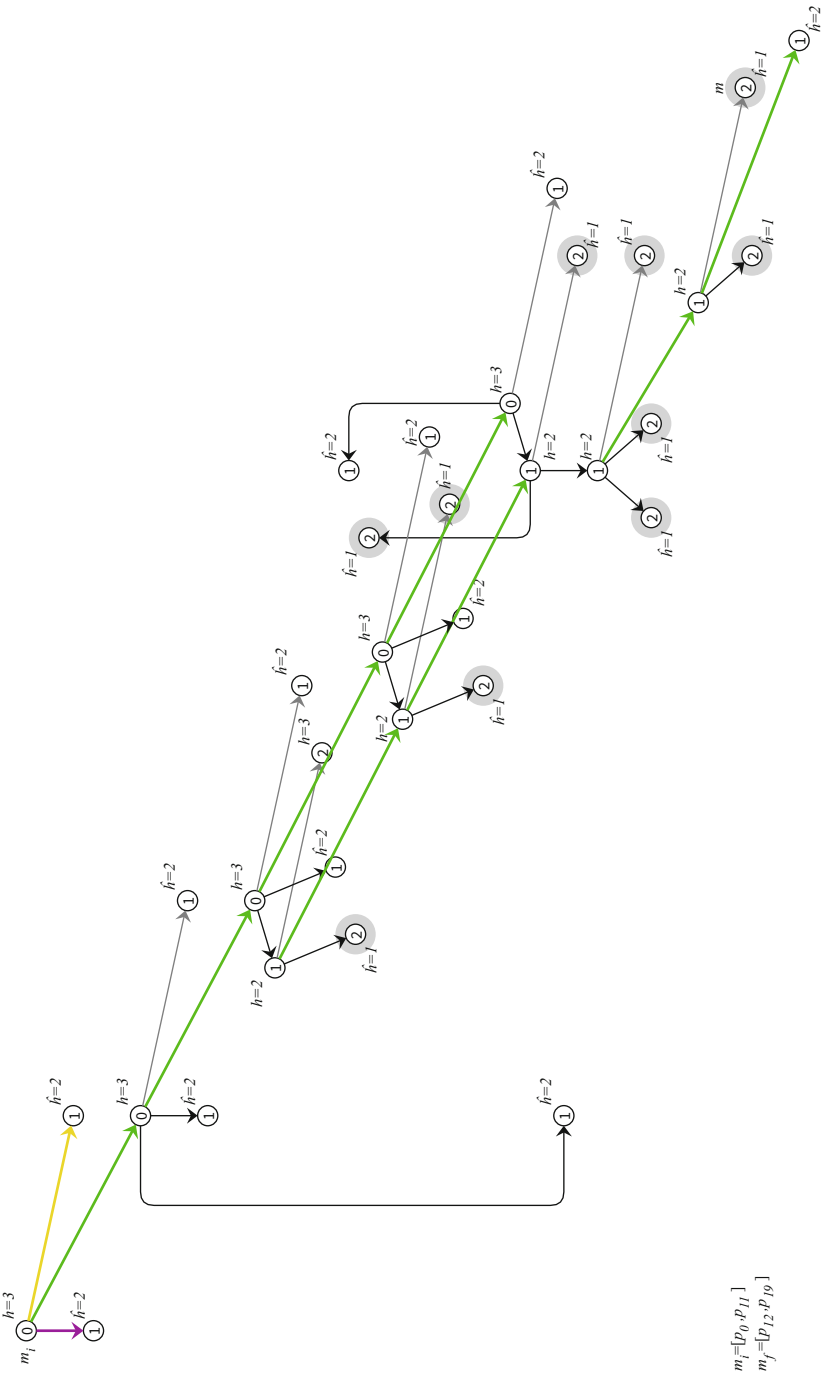


Figure 7.12 Evolution of $A^\star$ with estimated heuristic $\hat{h}$ until the point when for the first time no true estimate h is minimizing f in the open set X

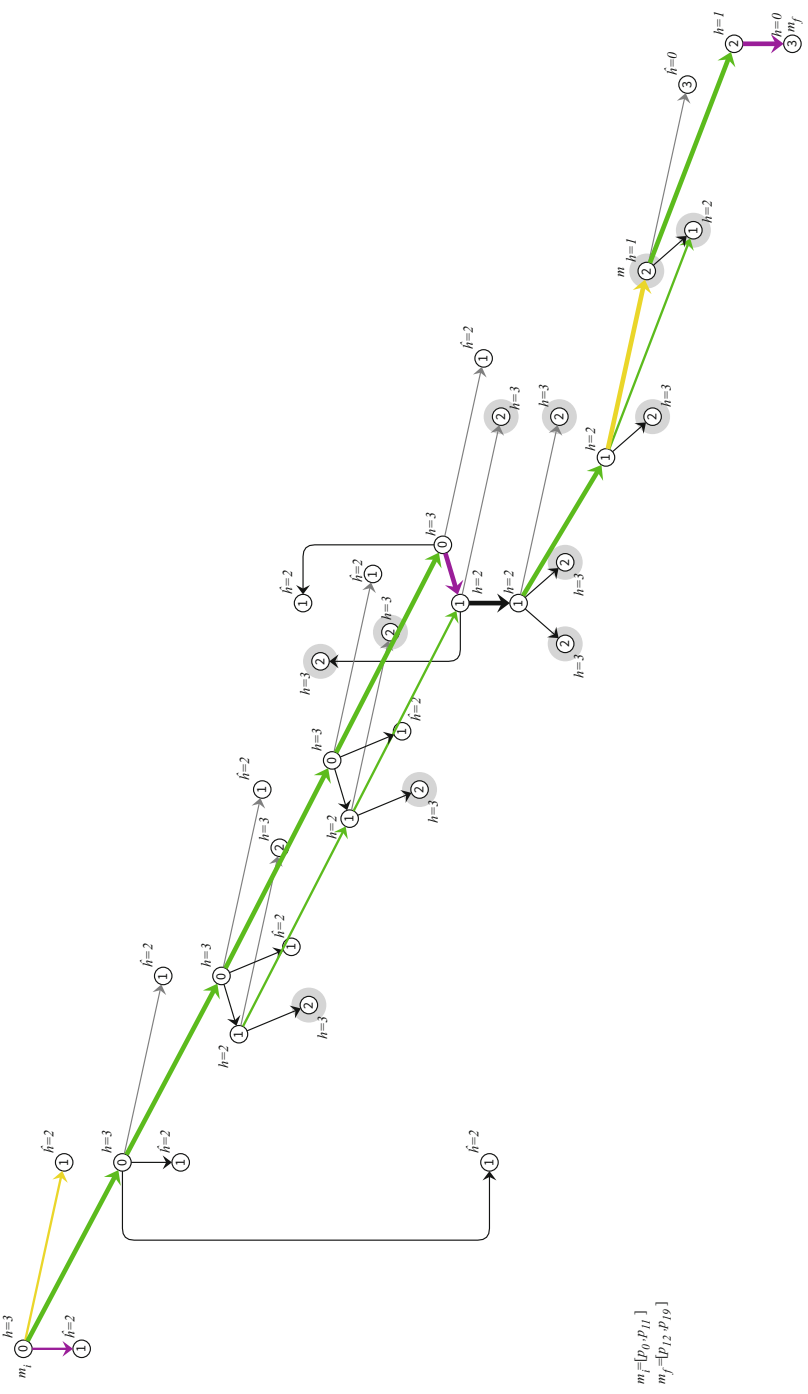


Figure 7.13 Evolution (worst case) of $A^\star$ with estimated heuristic $\hat{h}$ after completion. Note that there are many states for which $\hat{h}$ was never updated to h

Procedure 3 $A^\star$ algorithm for alignments with estimated heuristics

Let $SP = ((P, T, F, \lambda), m_i, m_f)$ be a synchronous product and let $c : T \to \mathbb{R}^+$ be a cost function and $h : \mathsf{RS} \to \mathbb{R}^+ \cup \{0\}$ be a heuristic underestimating the cost of getting from any marking to the final marking.

function ASTAR(SP, O, c)
  $A \leftarrow \emptyset$ ▷ Initialise closed set
  $X \leftarrow \{m_i\}$ ▷ Initialise open set
  $Y \leftarrow \emptyset$ ▷ Initialise estimated heuristics
  $p(m_i) = (\tau, \tau)$ ▷ Initialise predecessor function
  $\forall_{m \in \mathsf{RS}(SN)}\ g(m) = \infty$ ▷ Initialise cost so far function g
  $g(m_i) = 0$
  $\forall_{m \in \mathsf{RS}(SN)}\ f(m) = \infty$ ▷ Initialise estimated total cost function f
  $f(m_i) = h(m_i)$ ▷ Compute estimate for initial marking
  while $X \neq \emptyset$ **do** ▷ While not all states visited
    $m \leftarrow m \in X$ minimizing $f(m)$ ▷ Get the most promising marking m
    if $m = m_f$ **then** ▷ final marking reached
      break while
    end if
    if $m \in Y$ **then** ▷ Heuristic of m is not exact
      $x \leftarrow h(m)$ ▷ Compute the true estimate
      $Y \leftarrow Y \setminus \{m\}$ ▷ Remove estimated heuristic
      if $x > \hat{h}(m)$ **then** ▷ Heuristic increased
        $f(m) \leftarrow g(m) + h(m)$ ▷ Update estimated total cost function
        continue while ▷ Note: m may not be minimizing f any more
      end if ▷ If heuristic did not chance, continue with m
    end if
    $A \leftarrow A \cup \{m\}$ ▷ Add m to the closed set
    $X \leftarrow X \setminus \{m\}$ ▷ Remove m from the open set
    $T' \leftarrow T$
    if $\#_1(p(m)) \in T^{LM}$ **then** ▷ Marking m reached through a log move
      $T' \leftarrow T' \setminus T^{MM}$ ▷ Model moves not allowed after log moves
    end if
    for all $t \in T'$ with $m[t\rangle m'$ **do** ▷ For each relevant transition enabled in m
      if $m' \notin A$ **then** ▷ Reaching a marking not yet visited
        $X \leftarrow X \cup \{m'\}$ ▷ Add m' to the open set
        $a \leftarrow g(m) + c(t)$ ▷ Compute the cost so far of reaching m' via m
        if $a < g(m')$ **then** ▷ If this current cost is better than known cost so far
          $g(m') \leftarrow a$ ▷ Update cost so far function
          if $h(m')$ can be derived from $h(m)$ or $h(m')$ in cache **then**
            $f(m') \leftarrow g(m') + h(m')$ ▷ Update estimated total cost function
          else
            $Y \leftarrow Y \cup \{m'\}$ ▷ Add m' to the estimated heuristics set
            $f(m') \leftarrow g(m') + h(m) - c(t)$ ▷ Update estimated total cost function
          end if
          $p(m') \leftarrow (t, m)$ ▷ Update predecessor function
        end if
      end if
    end for
  end while
  $\gamma \leftarrow \langle t_0, \ldots, t_n \rangle$ such that $t_n = \#_1(p(m_f))$, $t_{n-1} = \#_1(p(\#_2(p(m_f))))$ etc. until the initial marking is reached recursively.
  return $f(m_f), \gamma$ ▷ Return distance and alignment
end function

Table 7.1 Overview of number of states and number of (I)LPs solved

Algorithm	States		Edges		(I)LPs					
					computed		derived		estimated	
	min	max	min	max	min	max	min	max	min	max
Full	104		244							
Dijkstra	71	95	128	180						
$A^\star$	31	37	31	41	31	36				
$A^\star$ with derived heuristics	31	37	31	41	21	25	10	11		
$A^\star$ with estimated heuristics	36		38		1	12	9	13	22	23

7.4.5 Caching Estimates

In this chapter, we focus on the problem of aligning a trace with a model, but in practice, we are interested in aligning all traces in a log with a model. When aligning an event log with a model, each trace can be considered separately. However, many traces may actually reach the same marking in the original process and may have the same activities in the trace.

In the heuristic function presented in Definition 17, the marking equation is used. This implies that it abstracts from the order in which transitions are executed in the synchronous product and therefore, we can build a cache of solutions that, for each pair of a marking in the process model and the Parikh vector of remaining activities in the trace, stores the solution to the linear program. This allows us to reuse these solutions from one trace to the next, thereby speeding up the algorithm when applied to logs at the cost of memory.

Excursion 14

Notes on implementing the open set X

In both Dijkstra's algorithm and $A^\star$, elements are taken from an open set, such that they minimize the value of a given function. To efficiently implement such a set, a priority queue is a well-known data structure. A priority queue (implemented using a balanced binary heap) keeps its elements sorted based on the given cost function, and operations like insertion (line 18 of Algorithm 2) and deletion (line 10 of Algorithm 2) are logarithmic in the size of the queue.

In $A^\star$, the function f gets updated frequently (lines 16, 33 and 36 of Algorithm 3) when new, shorter paths are found to an already scheduled, but not yet visited marking. Sometimes, the marking needs to be moved up in the queue if it was reached with lower cost g (see lines 33 and 36 of Algorithm 3). Function f may also increase, when for a certain marking m, $\hat{h}(m)$ is replaced by $h(m) \geq \hat{h}(m)$ (line 16 of Algorithm 3).

(continued)

The choice of data structure for X is therefore important. In current publicly available implementations, an array-based binary heap backed by a hash table is used. This implementation allows for most operations to be logarithmic in the size of the queue (deletion, insertion, update).

7.5 Another Example

Before discussing the complexity of alignments in more detail, let us consider another example trace $\langle As, AaFa, Aaa, Ao, Af \rangle$ for our running example. In this trace, Aaa and Ao are clearly swapped and at least Sso and Ro are missing, i.e. we expect the alignment to have cost 4. Figure 7.14 depicts the search space using the techniques in this chapter at the point where none of the markings in the priority queue have a known estimate h, i.e. they all have $\hat{h}$ values. This expanded search space is comparable to the one depicted earlier in Figure 7.12, with one important difference. The highlighted states that currently have an estimated value for f of 2, i.e. $g + \hat{h} = 2$, get an exact value for f of 4 after computing a new estimate. This means that they will be pushed down the priority queue one by one until for

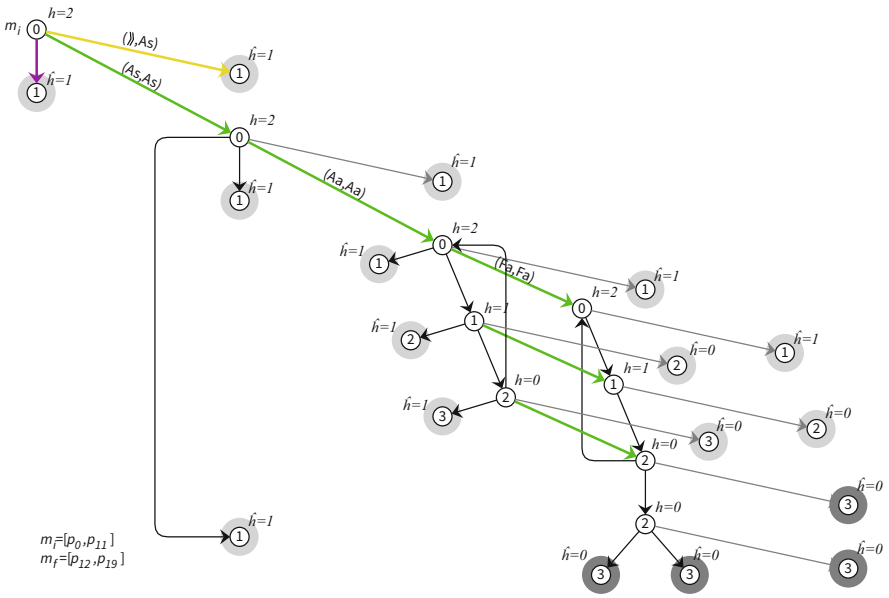


Figure 7.14 Evolution of $A^\star$ with estimated heuristic $\hat{h}$, at the first point when no true h values are known. Highlighted markings are in the priority queue, darker implies a higher f-score

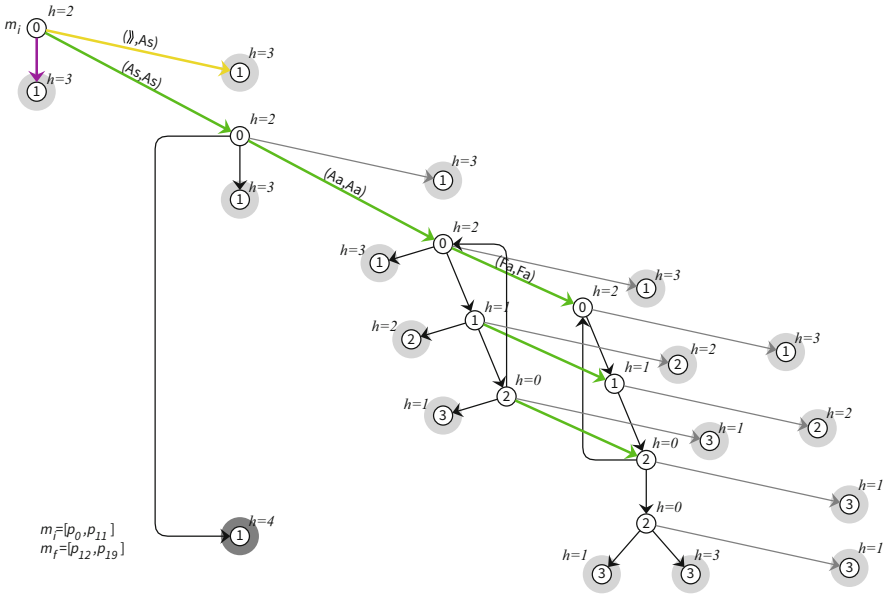


Figure 7.15 Evolution of $A^\star$ after computing all exact h values. Highlighted markings are in the priority queue, darker implies a higher f-score

each highlighted node the exact h value is obtained. These exact values are shown in Figure 7.15.

Only then the search continues, which leads to a worst-case expansion of the search space depicted in Figure 7.16. For this $A^\star$ search, a total of 18 linear problems were solved to find the following optimal alignment:

log trace	As	Aa	Fa	$\gg$	$\gg$	$\gg$	Aaa	Ao	$\gg$	Af
execution sequence	As	Aa	Fa	Sso	Ro	τ	$\gg$	Ao	Aaa	Af

The reason that $A^\star$ expands so many markings for the trace $\langle As, AaFa, Aaa, Ao, Af\rangle$ lies in the fact that the marking equation is unable to detect the swapped activities at the end of the trace, i.e. in the marking equation these events are both estimated to correspond to synchronous moves, which is clearly not possible.

7.6 Complexity Results

In Petri net theory, the problem of reachability is the problem of answering the question: Given a Petri net, an initial marking and a target marking, is the target marking reachable from the initial marking by firing a sequence of transitions? And if so, through which sequence? The problem of reachability can easily be translated

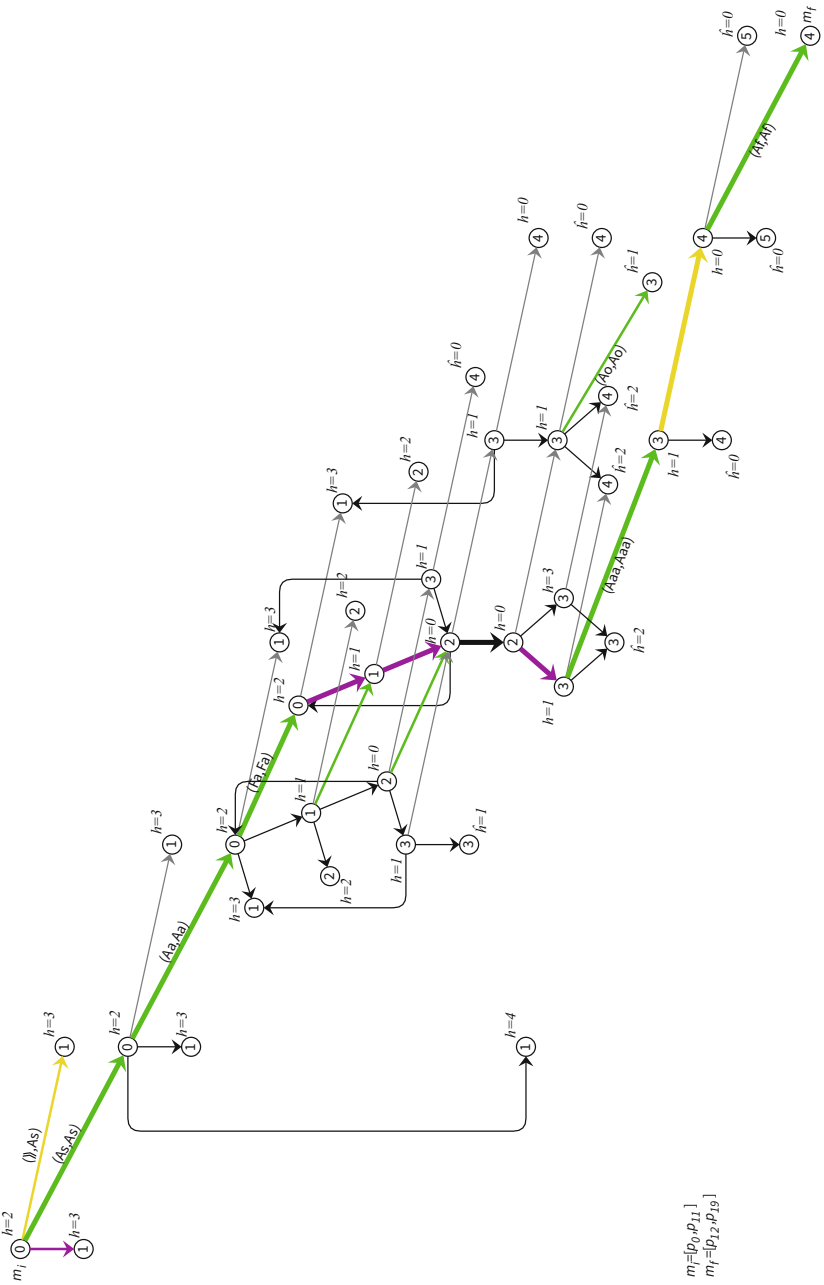


Figure 7.16 Worst case evolution of $A^\star$ for the trace $\langle As, AdFa, Aaa, Ao, Af \rangle$

into an alignment problem, i.e. find an optimal alignment of the empty trace and the given model from the initial marking to the target marking, with a cost function that assigns identical costs (> 0) to any transition.

If the target marking is reachable, then an optimal alignment can be found; if the target marking is not reachable, then an optimal alignment does not exist. Any algorithm that returns alignments is therefore guaranteed to be at least as complex as reachability. Hence, the theoretical bounds of the *optimal* alignment problem are in general higher than those of reachability.

However, as discussed earlier in this chapter, we explicitly assume easy soundness, which implies reachability of the final marking from the initial one. Hence the algorithm presented in this chapter does not cover the situation in which the target marking (of the synchronous product) is not reachable, i.e. we assume that the target marking is reachable. This assumption is needed to guarantee termination of the $A^\star$-based technique.

More specifically, the $A^\star$-based techniques presented in this chapter require a worst-case storage space linear in the number of reachable markings that are at a distance less or equal to the optimal distance from the initial marking. Furthermore, the worst-case time complexity is linear in the number of transitions between these states. The size of this subgraph is, unfortunately, worst-case exponential in the size of the Petri net (the well-known state space explosion problem in concurrent systems).

Due to the aforementioned limitations, the $A^\star$-based technique may have problems in dealing medium or large instances. The reader is encouraged to wait till Chapter 9, where alternative techniques to compute alignments are presented. By decomposing the problem or by symbolically exploring the search space, the problem of computing alignments is alleviated considerably, sometimes at the expense of losing some of the guarantees provided by the technique explained in this chapter.

To finish the content of this chapter, in the next excursion we informally discuss an interesting extension of the alignment techniques presented so far: in general, not one but several optimal alignments may exists for a given trace. How to compute them all?

Excursion 15

Looking for all explanations

The alignment techniques presented in this chapter produce a single, optimal alignment. However, for any given cost function, many alignments may be optimal. Hence, the question arises if there is a way to obtain *all optimal alignments*. This turns out to be a complex question.

Using Dijkstra's algorithm (Algorithm 1), finding all optimal alignments is possible by not stopping when the final marking is reached (line 10), but by stopping when the final marking is reached with sub-optimal cost for the first

(continued)

time (i.e. with cost higher than with which it was reached the first time). This, however, does not guarantee that all optimal alignments are found, since only a single path from the initial marking is stored. Post-processing is needed to find all possible sequences.

Also $A^\star$ can be changed to obtain all optimal alignments. In line 12 of Algorithm 2 the algorithm terminates the first time the final marking is reached. However, to obtain all optimal alignments, the algorithm should continue until the head of the priority queue has an *f-score* higher than that of the optimal path identified. This ensures again that all states are reached that are on a shortest path and the same post-processing is needed as before to find all sequences.

Algorithm 3 cannot be changed to find all optimal alignments as certain parts of the search space are not visited by definition.

Bibliographic Notes

The theory of this chapter is taken from the seminal work on alignments from Arya Adriansyah's PhD thesis [1]. The idea of alignments was first introduced by Adriansyah et al. in [6], where it was presented as a technique to get conformance insights into "flexible models". Only later, the synchronous product was introduced in [5] as a basis for formal alignment computations on Petri nets. In [120] further improvements to the algorithm in Section 9.4 have been made using Petri net theory.

Over time, the fundamental algorithm for computing alignments did not change, but many tweaks were added to the implementation which is available in the open source framework ProM, which we discuss in detail in Chapter 12. In [124] a first-ever empirical evaluation was performed on the effect of various of these tweaks on runtime and memory use of the $A^\star$ algorithm.

7.7 Exercises

7.A) Synchronous product

Draw the trace model for the trace $\langle As, Da \rangle$.
Next, draw the synchronous product for this trace and the process model given in Figure 7.1.

7.B) Effect of the heuristic function

Consider the trace $\langle As, Aa, Fa, Sso, Co, Da, Af \rangle$ and an integer heuristic function h^I. If we solve the linear equation system for the initial marking of the corresponding synchronous product, we get one of the two following solution vectors, both with identical value for $h = 4$:

$[(As, As), (\gg, Aa), (\gg, Fa), (\gg, Sso), (\gg, Co), (Da_1, Da), (Af, Af)]$
$[(As, As), (Aa, Aa), (Fa, Fa), (Sso, Sso), (Ro, \gg), (Co, Co), (Sso, \gg), (Ro, \gg), (\tau, \gg), (Do, \gg), (Da_2, Da), (Af, Af)]$

Explain what the corresponding alignment looks like for both vectors when using $A^\star$ in Algorithm 3. What is the difference between the two alignments and why are these differences there.

7.C) $A^\star$ search

Illustrate the execution of the $A^\star$ algorithm for the example of Exercise 7.A.

7) In the lab: tool support for aligning event logs and process models

Check out the lab session to practise with tools that enable aligning event logs and process models.

http://www.conformancechecking.com/CC_book_Chapter_7

Chapter 8
Interpreting Alignments

The computation of optimal alignments is a challenging task, which demands efficient techniques such as the ones seen in the previous chapter. For alternative techniques, the reader can have a look at Chapter 9, where the challenge of computing alignments is explored further, by considering different variations of the problem and alternative representations.

But is it really worth the *pain* of computing alignments? This chapter is devoted to reflecting on the principal role alignments have in the area of conformance checking. We show that the notion of an optimal alignment poses some considerations that need to be realized upfront, with the non-uniqueness being the most critical (Section 8.1). Then, in Section 8.2 the importance of visualizing alignments is reported, since human interpretation is often crucial in a conformance checking project. Section 8.3 delves into the properties any quality metric should have. We then dive into the interpretations that can be extracted when optimal alignments are available, mainly for the two important conformance checking metrics: fitness (Section 8.4) and precision (Section 8.5).

8.1 Types of Alignments

Non-synchronous moves in an alignment reflect deviations between modelled and recorded behaviour. The different types of deviations are assigned a cost (cf. Definition 15 in the previous chapter), which then enables to compare alignments with respect to the sum of costs corresponding to the deviations of each of them.

J. Carmona et al., *Conformance Checking*,
https://doi.org/10.1007/978-3-319-99414-7_8

Excursion 16

What is the interpretation of costs in an alignment?

Given a deviation (e.g., a move on model), the cost assigned to it may have alternative interpretations. This cost is often assumed to be assigned by a domain expert, and clearly influences the final alignment obtained.

On the one hand, the *penalty interpretation* assigns a cost to deviations with the aim at having an explanation of the recorded behaviour that is less harmful with respect to the severity of the penalties. For instance, a move on model representing a crucial activity for the process (i.e., an activity that was required by the model but was not executed in reality) may be assigned a high cost so that, if possible, the model explanation of the recorded behaviour finds an alternative way which incurs a smaller cost by avoiding incorporating that particular deviation. As an example, let us assume that the trace $\langle As, Aa, Fa, Sso, Ro, Af \rangle$ is recorded for the model of Figure 7.1, and all deviations have assigned a unit cost except for activities "Decline Offer" and "Accept Offer", whose deviation incurs a cost of 10 units. In the aforementioned trace, the cheapest explanation according to the considered cost would be the execution sequence $\langle (As, As), (\gg, Aa), (\gg, Fa), (\gg, Sso), (\gg, Ro), (Da, \gg), (Af, Af) \rangle$, which has cost 5 and represents an immediately declined application, in spite of the trace suggesting that the process continued for a while.

On the other hand, the *likelihood interpretation* is centered around assigning a cost associated to deviations in order to mimic the probability of the deviations to happening in reality. For instance, one may regard model moves as less likely than log moves, or even in a more granular perspective, to consider a probability distribution over the set of activities that form the alphabet of the synchronous product.

Finally, in the *trust interpretation*, costs would represent the trust we have on each of the two objects confronted: process model and log [91]. For instance, if there is a high confidence in the log, then an alignment that minimizes log moves is preferred since the model strives to explain as much as possible the behaviour observed, even if several model moves are required.

For the rest of the chapter, let us assume the default cost function (cf. Definition 15) that assigns unitary costs to deviations and null costs to synchronous moves. In the search for an optimal alignment, a crucial observation that needs to be addressed is the existence of several solutions with the same optimal cost, i.e., optimal alignments are not unique. Let us now get back to the running example of this book to highlight this fact; given the trace $\langle As, Aa, Fa, Sso, Ro, Af \rangle$ and the Petri net of our running example shown in Figure 7.1, two possible alignments exist with the same optimal cost of 2 units:

log trace	As	Aa	Fa	Sso	Ro	$\gg$	$\gg$	$\gg$	Af
execution sequence	As	Aa	Fa	Sso	Ro	τ	Do	Da	Af

log trace	As	Aa	Fa	Sso	Ro	$\gg$	$\gg$	$\gg$	Af
execution sequence	As	Aa	Fa	Sso	Ro	τ	Ao	Aaa	Af

In the first alignment, the trace is regarded as a declined offer, while in the second, the offer is considered accepted. Notice the importance of the previous observation on the interpretation of costs, in order to guide the search for optimal alignments in the most meaningful way for obtaining the desired interpretation. For instance, one may simply assign a higher cost to model moves that consider the action "Decline offer", in order to untie the previous two optimal alignments, causing that the first alignment is no longer optimal. Another possibility, as we illustrate in Section 9.1, is to consider other perspectives as well (e.g., the amount offered) to determine the most likely alignment.

The previous example illustrates an important situation that needs to be considered when using alignments as core elements to relate modelled and recorded behaviour: there may be many optimal ways to relate these two views on the process (in general, the number of optimal alignments is not known a priori). Clearly, the selection of the optimal alignment among the set of possible options has an influence on the analysis that is done afterwards, e.g., in the previous example, it is not the same to regard the trace as an accepted or declined offer.

An alternative is to consider not one but all optimal alignments. This option, although being more general, may blur the later analysis since there are not one but several correspondences between modelled and recorded behaviour, and only aggregated factors can be used in the analysis. For instance, for the previous example, two optimal alignments exist for the given trace $\langle As, Aa, Fa, Sso, Ro, Af\rangle$ (assuming the default cost function). When analysing the likelihood of an unfinished trace $\langle As, Aa, Fa, Sso, Ro\rangle$ being accepted (i.e., *prediction*, a well-known use case of conformance checking we discuss in Chapter 10), we may conclude that one of them is more likely than another, for example because the data shows that an application for which an offer has already been sent to and received by the customer is more likely to be accepted.

8.2 Visualization of Alignments

When using alignments in practice, we are often confronted with a considerable number of traces and trace variants (traces that represent the same sequence of activity executions are said to be of the same variant). It can be tedious to look at long tabular representations of alignments. To assist users in reading and interpreting one alignment, a visualization with colour coding was proposed to enable analysts and researchers to spot and quantify the deviations quicker than

by comparing tables. In this way, we can encode the following alignments

log trace	As	Aa	Sso	Ro	$\gg$	$\gg$	Do	Da	Af
execution sequence	As	Aa	Sso	Ro	Fa	τ	Do	Da	Af

log trace	As	Aa	Sso	Ro	$\gg$	$\gg$	Ao	Aaa	Aaa	$\gg$
execution sequence	As	Aa	Sso	Ro	Fa	τ	Ao	Aaa	$\gg$	Af

in a representative visualization shown in Figure 8.1. This visualization helps to spot single deviations in a single trace (variant) of an event log. Most real life event logs, however, consist of hundreds or thousands of variants. Thus, to evaluate the conformance of an entire event log with respect to the model, the analyst still needs to browse through a long list of single alignment visualizations.

Instead of projecting one alignment to one trace at a time, we can project all the alignment results into the model through aggregation. Using this technique, we can quickly locate deviations, if the misalignments are concentrated at certain tasks in the process model. Different aggregations of the alignment components (log move, model move, and synchronous move) are possible. Let us look at the simplest aggregation: the ratios of synchronous-, model-, and log moves of an activity in the model. Figure 8.2 depicts such an aggregation and we can easily locate the areas of non-conformance by checking which activities are not completely green. This overview visualization serves as indicator to see where in the process deviations

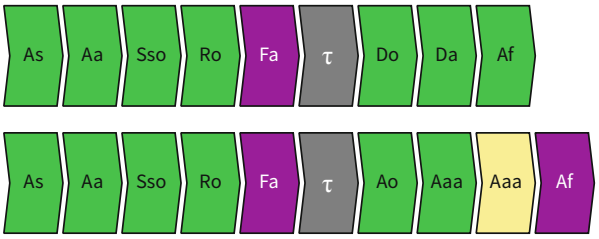


Figure 8.1 Visualization of two alignments using chevrons and colours. Green indicates synchronous moves, magenta is used for model moves (except for model moves on τ-labelled transitions which are grey) and yellow indicates log moves

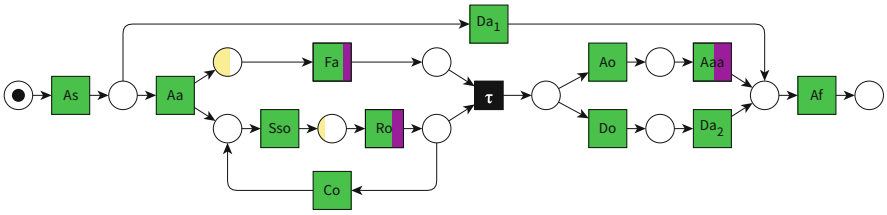


Figure 8.2 Visualization of all the alignments of a log on a model. Green indicates synchronous moves and magenta is used for model moves. The colour ratio represents the relative occurrence of deviations. Yellow-filled places indicate that log moves occurred while these places were marked

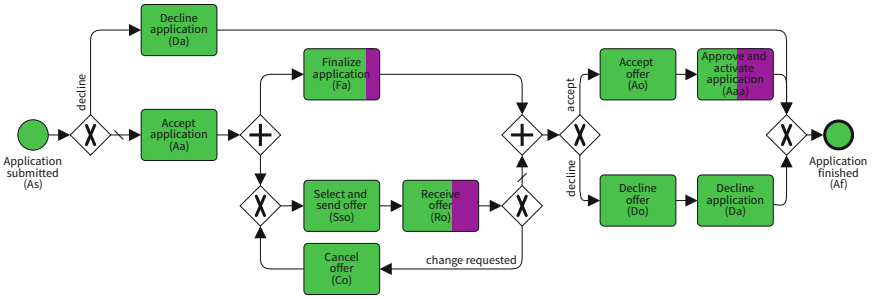


Figure 8.3 Visualization of all the alignments of a log on a model. Green indicates synchronous moves and magenta is used for model moves. The colour ratio represents the relative occurrence of deviations

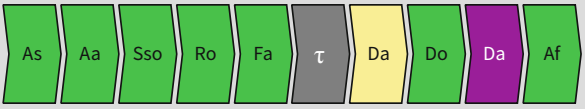
happen. Note that log moves are not easy to handle when projecting alignment results to models as log moves occur in certain states and are not linked to transitions in the Petri net. Figure 8.3 shows that such a visualization is not limited to Petri nets, but can be used on other model types as well.

While measures like fitness and precision are computed on an aggregated level on the complete log and the corresponding model, the projection of the alignment results into the model allows us to get fine-grained information about the tasks, where events are missing, or where events are in excess with respect to the model. In Excursion 17, we look at a particular combination of log and model moves that also allow a joint interpretation.

Excursion 17

Swapped behaviour

Looking at single alignment visualizations as depicted in Figure 8.1, sometimes we see patterns. One common pattern is that we encounter a log move of an event and a model move of the corresponding task later on. Obviously, the same pattern also works the other way around, meaning that the model move comes first and the log move of the corresponding event follows. Consider the following example:



(continued)

Here, we can infer that the original trace is $\langle As, Aa, Sso, Ro, Fa, Da, Do, Af \rangle$. The two events *Da* and *Do* are *swapped* in their execution order, which in turn produces an alignment with one model move and a log move of *Da*. Assuming equal positive costs for model moves and log moves of all activities in this example, would there be another equally optimal alignment of the trace?

To conclude this section, visualizations like the ones discussed can help to easily spot pain points in the process. These tasks in the model that show the highest ratio of non-conforming behaviour are good candidates to start the investigations for the root causes of misalignments.

8.3 Properties of Quality Metrics

Before digging into different metrics for fitness and precision, we discuss the properties a metric should have. These properties have arisen as important features in practice, but the list provided below is not meant to be complete. Also, it is important to emphasize that current metrics often do not satisfy all the properties, but only a subset. A deep discussion on metrics for precision can be found in [106].

First of all, a *deterministic* metric is often preferred, i.e., a metric computes the same value given the same inputs. This is an important property, since it provides certain reliability when using the metric on a regular basis. Instead, the use of non-deterministic metrics may blur the analysis, since insights extracted from them (e.g., a fitness problem) may be not meaningful. In general, alignment-based metrics tend to be deterministic. In contrast, as the following example illustrates, heuristic techniques to relate recorded and model behaviour like token replay, applied on non-deterministic models, may provide different values depending on the non-deterministic choices made.

For instance, consider the process model shown in Figure 8.4, and let us assume the following trace: $\langle As, Aa, Af \rangle$. In case of token-based replay, when replaying the event *Aa*, a decision has to be made about which one of the two paths to follow. If the upper branch is selected, then no problem in replaying the trace is encountered.

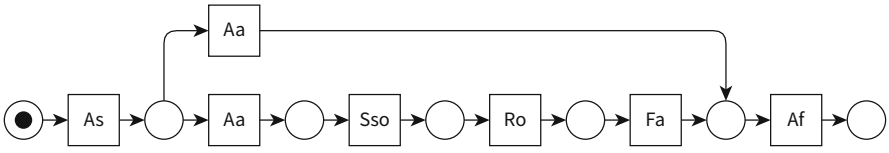


Figure 8.4 Process model with a non-deterministic decision

If, however, the lower branch is followed, then some deviations (manifested as missing/remaining tokens) are encountered, thus incorrectly assessing the trace as non-fitting. Since the decision on going to the upper or lower branch is not deterministic, token replay-based fitness is not a deterministic metric.

Second, a metric should be *monotonic* with respect to the addition or removal of additional behaviour: adding (removing) behaviour to (from) a model should only contribute to having at least the same fitness (same precision), or even higher fitness (lower precision). Symmetrically, the addition (removal) of traces to the log should only contribute to having at least the same fitness (same precision), or even lower fitness (higher precision). For instance, let us now consider a variation of the process model of Figure 8.4, where a new branch with additional checks from a senior member has been added, namely "Senior check 1 (*Sc1*)" and "Senior check 2 (*Sc2*)".

Clearly, the process model in Figure 8.5 is the result of adding behaviour to the process model of Figure 8.4. However, if confronted with the trace $\langle As, Aa, Af \rangle$, token-based replay on this trace for the process model in Figure 8.5 can take the lower branch, providing the trace $\langle As, Aa, Sso, Ro, Fa, Sc1, Sc2, Af \rangle$, whose replay incurs more violations, thus decreasing the fitness with respect to any of the two other possible branches, and therefore, violating the monotonicity property. Instead, alignment-based metrics tend to avoid this problem, since always the best (cost-minimal) model trace is returned before of evaluating the metric.

Third, a metric should be *behavioural*, i.e., two models having the same behaviour (but perhaps different structure) should be assigned the same value by the metric. Behavioural metrics are invariant to the modelling notation or primitives used, and focus on the underlying behaviour.

Finally, a metric is expected to be provided in $\mathbb{R}$, in the $[0 \ldots 1]$ range, so it can be understood easily. Moreover, it is expected that at least one input exists for which the metric provides the extreme values (0 or 1). For instance, for the running example of this book, one may expect that the flower model of Figure 3.12 should have fitness 1 and precision 0 for any given log.

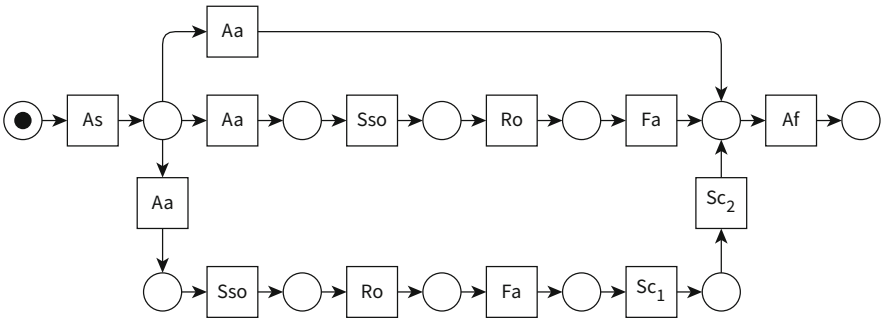


Figure 8.5 Process model with a non-deterministic decision

8.4 Calculating Fitness

Section 3.2 introduced fitness as a crucial metric to asses the model's capability in reproducing recorded behaviour. In this section, a summary of the most important variants to measure fitness is provided.

8.4.1 Token Replay Fitness

In Section 4.2 we described the token replay method as a light way of relating modelled and recorded behaviour. Consequently, this method can be used to provide a means to compute fitness.

Recall that in token replay the trace is used as a basis for executing activities in the model and cases are considered one by one. If we translate this into Petri nets, then for each event in a trace, a transition labelled with the corresponding activity is fired. Let us consider our running example (Figure 8.6) and the non-fitting trace $\langle As, Aa, Sso, Ro, Ao, Da, Af \rangle$.

We start by firing the transition labelled with *As*, since this is the only transition enabled and it also corresponds to the first event in the trace. After executing the subsequence $\langle As, Aa, Sso, Ro \rangle$, we reach a point where the model and the trace deviate. The trace shows activity *Ao*, but the only enabled transitions in the model are labelled with *Fa* and *Co*. The corresponding marking in the Petri net is depicted in Figure 8.7.

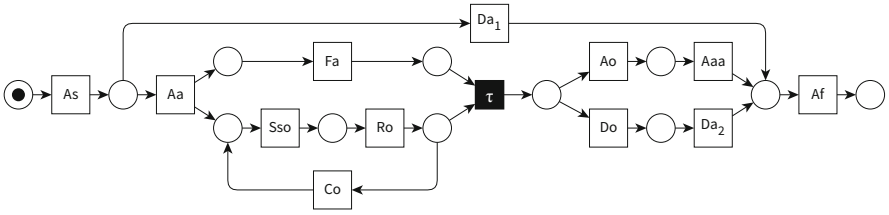


Figure 8.6 Petri net for the running example

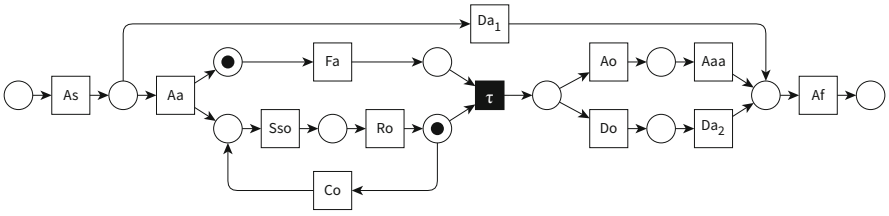


Figure 8.7 Petri net for the running example showing the marking at the point where the trace starts deviating from the model

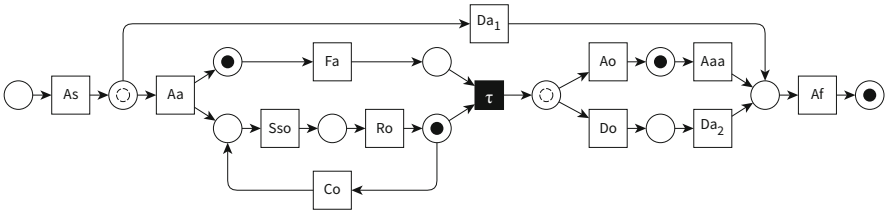


Figure 8.8 Petri net for the running example showing the marking after token replay

At this point, the trace shows activity *Ao* and in the model, the corresponding transition is fired. However, since it is not enabled, a missing token is recorded. The last event in the trace is the activity *Da* and there are actually two equally labelled transitions in the model. Since both are not enabled, a random decision is taken to execute one of them. In this case, we select Da_1 on the top and consequently, after firing the enabled transition *Af*, we obtain the final state depicted in Figure 8.8, where the dashed circles represent missing tokens and the closed tokens represent the remaining tokens.

After replaying this trace, there are two missing tokens recorded and three remaining tokens (note that the token in the output place does not count as we expect this token to be there. If it wouldn't be there, it would be counted as a missing token). In total, eight tokens were consumed (seven by the transitions that fired and one from the final place after completing the trace) and nine produced (one for the initial marking and eight by the transitions that fired). The token-based fitness for a trace is therefore $f_t^t = \frac{1}{2} \cdot (1 - \frac{\text{missing}}{\text{consumed}}) + \frac{1}{2} \cdot (1 - \frac{\text{remaining}}{\text{produced}}) = \frac{1}{2} \cdot (1 - \frac{2}{8}) + \frac{1}{2} \cdot (1 - \frac{3}{9}) \approx 0.7083$. For an entire log, the token-based fitness sums all missing/consumed/remaining and produced tokens as shown in Equation 4.4.

From this small example, it is already clear that there are several issues with token-based replay. First of all, in the example above, the choice of which of the transitions labelled *Da* to fire is non-deterministic (see Section 8.3). Hence token-based replay violates the first property: determinism. Especially when multiple transitions with the same label have different numbers of incoming and outgoing arcs, this may lead to different outcomes each time. There are ways to circumvent this issue partially. For example, when firing *Ao* if it is not enabled (i.e. in the marking depicted in Figure 8.7) we could identify a sequence of τ-labelled transitions that enables *Ao*. However, in this example, such a τ-labelled transition is not enabled. Furthermore, firing it would lead to a different fitness value, 0.8.

Second, token-based replay suffers from so-called "token flooding". In the model in Figure 8.8 transitions *Fa*, *Co* and *Aaa* are all enabled. Adding any one of them to the trace would increase both the consumed and produced tokens by one, without an effect on the remaining or missing tokens. Hence, the fitness of the trace would *increase*, while adding a non-enabled transition to the trace would *decrease* the fitness. Hence, the token-based replay metric is non-monotonous.

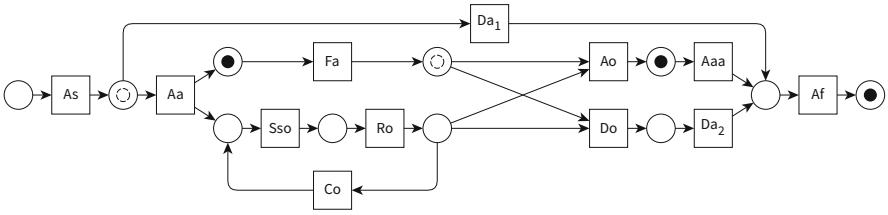


Figure 8.9 Petri net for the running example without τ showing the marking after token replay

Third, the τ-labelled transition in the model is never fired since it does not appear in the log, i.e. token-based replay cannot handle complex routing constructs that appear in most real-life processes. Since there are many ways to express routing in Petri nets using τ-labelled transitions, the token-based replay metric is non-behavioural. It is for example possible to eliminate the τ transition in this model by connecting both *Do* and *Ao* to the input places of τ as depicted in Figure 8.9, thereby improving fitness without changing behaviour (fitness would be 0.7778 in this case).

In conclusion, token-based fitness is in no way a fitness metric that satisfies the properties outlined in Section 8.3. Therefore, by using alignments, a more robust fitness metric can be devised. Sometimes, however, the computational challenge that represents the derivation of alignments makes techniques like token-based fitness the only choice when dealing with large instances.

8.4.2 Alignment-Based Fitness

To overcome the issues with token-based replay, alignment-based fitness has been developed. As discussed in Chapter 7, alignments provide a way to compare event logs and models by providing the most likely execution of a model for a given trace. In token-based replay, the decision which transition to fire given the recorded activity is made locally, while alignments are computed on the entire state space of the synchronous product, i.e. decisions on which transition to fire are made globally.

The way alignments are constructed, i.e. by looking for a shortest path through the state space of the synchronous product, is not necessarily deterministic. There may be more than one shortest path. However, the final cost of the alignment is minimal and therefore deterministic. On the basis of this cost, alignment-based fitness is defined as:

$$f_a = 1 - \frac{\text{cost of the optimal alignment}}{\text{cost of worst-case alignment}}$$

Again, for any log L we have:

$$f_a^l = 1 - \frac{\sum_{\sigma \in L} \text{cost of the optimal alignment for } \sigma}{\sum_{\sigma \in L} \text{cost of worst-case alignment for } \sigma}$$

For alignment-based fitness, two costs are of interest, namely the cost of the optimal alignment and the cost of the worst-case alignment. The former is obtained by the alignment algorithm defined in Chapter 7. The latter is simply defined as the cost of aligning the empty trace in the model plus the cost of treating all events as log moves.

As an example, let's again consider Figure 8.6 and the trace $\langle As, Aa, Sso, Ro, Ao, Da, Fa \rangle$ with the default cost function (assigning cost 1 to deviations and 0 to synchronous moves and τ-labelled transitions). An optimal alignment for this trace is:

log trace	As	Aa	Sso	Ro	$\gg$	$\gg$	Ao	$\gg$	Da	Fa
execution sequence	As	Aa	Sso	Ro	Fa	τ	$\gg$	Do	Da_2	Fa

The cost of this alignment equals 3 (execution of the τ step as a move on model has cost 0). The worst-case alignment for this trace is:

log trace	$\gg$	$\gg$	$\gg$	As	Aa	Sso	Ro	Ao	Da	Fa
execution sequence	As	Da_1	Fa	$\gg$	$\gg$	$\gg$	$\gg$	$\gg$	$\gg$	$\gg$

This worst-case alignment has a cost of 10, i.e. the cost of the 3 move on model steps at the beginning and the 7 move on log steps at the end. The fitness of this trace in this model is therefore $f_a = 1 - \frac{\text{cost of the optimal alignment}}{\text{cost of worst-case alignment}} = 1 - \frac{3}{10} = 0.7000$.

It is easy to see that this metric is deterministic since any other optimal alignment of this trace would yield the same cost for the optimal alignment. Furthermore, the metric is monotonous *given the model*. The alignment-based fitness metric is not monotonous when the model is changed.

If two models allow for the same behaviour in different ways, the alignment-based fitness metric also does not change. The actual alignment might contain more moves on models with τ-labelled transitions, but these do not add to the cost of the alignment and hence the cost does not change even if the optimal alignment does. The model depicted in Figure 8.9 for example has the same behaviour as the one with the τ transition. The alignment-based fitness is therefore the same. Finally, it is easy to see that alignment-based fitness is indeed a metric, i.e. it has a value between 0 and 1 (fitness 0 can be reached by using a model and a log with two disjoint sets of labels).

8.4.3 Move-Based Fitness

The fact that alignment-based fitness is not monotonous when the model changes makes this metric less useful when comparing models against a single log. For that purpose, two other metrics can be based on alignments, namely move-model fitness and move-log fitness.

Both move-model and move-log fitness are defined purely on the alignment, i.e. they do not require a worst-case alignment in the model. They both relate the cost of the model moves (or log moves) to the cost of the alignment assuming that all synchronous moves are also model moves (or log moves).

Consider again the example trace $\langle As, Aa, Sso, Ro, Ao, Da, Fa\rangle$ and an optimal alignment:

log trace	As	Aa	Sso	Ro	$\gg$	$\gg$	Ao	$\gg$	Da	Fa
execution sequence	As	Aa	Sso	Ro	Fa	τ	$\gg$	Do	Da_2	Fa

The move-model fitness for this trace is defined as

$$f^t_{mm} = 1 - \frac{\text{model move cost}}{\text{model move cost} + \text{cost of synchronous moves interpreted as model moves}}$$

which then for the example would be:

$$1 - \frac{2}{2+6} = 0.7500$$

The move-log fitness for this trace is defined as

$$f^t_{ml} = 1 - \frac{\text{log move cost}}{\text{log move cost} + \text{cost of synchronous moves interpreted as log moves}}$$

which then for the example would be:

$$1 - \frac{1}{1+6} = 0.8571$$

Unfortunately, these two metrics are not deterministic since a different alignment for the same trace might yield more model moves in favour of log moves or vice versa. Consider for example the trace $\langle As, Ro, Co, Sso, Da, Fa\rangle$. There are at least two optimal alignments, namely:

log trace	As	Aa	$\gg$	Ro	Co	Sso	$\gg$	$\gg$	$\gg$	$\gg$	Da	Fa
execution sequence	As	Aa	Sso	Ro	Co	Sso	Ro	Fa	τ	Do	Da_2	Fa

and

log trace	As	Aa	Ro	Co	Sso	Da	Fa
execution sequence	As	$\gg$	$\gg$	$\gg$	$\gg$	Da_1	Fa

The move model fitness for the first alignment is $f^t_{mm} = 1 - \frac{4}{4+7} = 0.6364$ and for the second is 1. The move log fitness is 1 for the first alignment and $f^t_{ml} = 1 - \frac{4}{4+3} = 0.4286$ for the second.

Because of the non-determinism in these metrics, the generally accepted standard for expressing fitness of a log with respect to a model is alignment-based fitness. In the remainder of this chapter, when we mention fitness, we mean alignment-based fitness.

8.4.4 The Role of Fitness

When comparing event logs and process models we prefer to look at the behaviour in both. In Chapter 3, we presented behaviour in a relatively abstract way and we introduced the intuition behind fitness and precision.

When looking at alignments, we see that they provide a natural interpretation of fitness since alignments pinpoint the minimal changes needed to fit recorded behaviour into a process model. In essence we accept the model as the ground truth and we identify deviations in behaviour in terms of "behaviour that was recorded, but was unexpected" (move on log) and behaviour that is "assumed to have taken place, but was not recorded" (move on model). The more such deviations exist, the less fitting a log is given the model.

For precision and other conformance metrics, we are interested in the behaviour of a model that did not manifest yet (the more behaviour a model allows for, the less precise it is) and hence we need to know what part of the model's behaviour was in fact observed. This is where alignments play an important role.

By computing fitness through alignments, we "fix the log", i.e. the alignments provide us with a reasonable assumption on the *actual behaviour* of the model that occurred and was recorded in the log. In other words, by assuming that the move on log events indeed were erroneously recorded and that the move on model events did in fact happen and were not recorded, we allow for a clean separation of concerns between fitness and other conformance metrics.

In the remainder of this chapter, the assumption is generally made that the recorded behaviour fits the model. This should be read in such a way that when an event log does not fit the model, the non-fitting traces are first aligned and then the aligned traces are considered to be the behaviour against which other metrics are computed.

8.5 Calculating Precision

In Section 3.3 we already discussed the importance of having a metric to quantify how precise a model is in describing recorded behaviour. This section focuses on the operational problem of computing precision. In particular, we provide two possible precision metrics, and reason on their adherence to the properties referred to in Section 8.3. To explain the metrics in this section, we use the τ-reduced Petri net for our running example, shown in Figure 8.10.

8.5.1 Precision Based on Escaping Arcs

As has been pointed out in Section 3.3, the need to explore the state space of a process model in order to compute precision is a limiting factor in providing an accurate metric. In spite of this, some techniques exist that try to limit this exploration while capturing the underlying precision. The work in [72] and later extended in [4] presented the simple idea of limiting the state space of the model to the part that is visited by sequences in the log, and penalizing the escaping points, i.e., arcs that are possible in the model and deviate from the log behaviour.

Let us denote by $\mathcal{L}_{M_{esc(L)}}$ the language corresponding to escaping arcs of model M over log L, that is,

$$\mathcal{L}_{M_{esc(L)}} = \{\sigma \cdot \langle a \rangle \mid a \in A,\ \sigma \in \text{Pref}(L \cap \mathcal{L}_M),\ \sigma \cdot \langle a \rangle \in \text{Pref}(\mathcal{L}_M \setminus L)\}$$

where Pref(X) denotes the set of all prefixes of a language X. Clearly, when the model is fitting, $\text{Pref}(L) = \text{Pref}(L \cap \mathcal{L}_M) \subseteq \text{Pref}(\mathcal{L}_M)$ holds. Remarkably, since the event log and the alphabet of activities A are finite, so is $\mathcal{L}_{M_{esc(L)}}$. In this context, the formula

$$precision_{approx} = \frac{|\text{Pref}(L \cap \mathcal{L}_M)|}{|\text{Pref}(L \cap \mathcal{L}_M)| + |\mathcal{L}_{M_{esc(L)}}|} \tag{8.1}$$

provides an approximation to the real precision of M with respect to L.

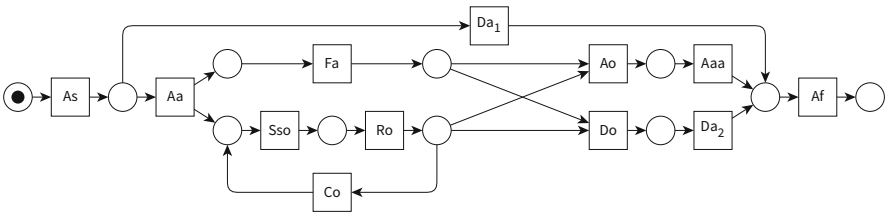


Figure 8.10 Petri net for the running example without τ-labelled transitions

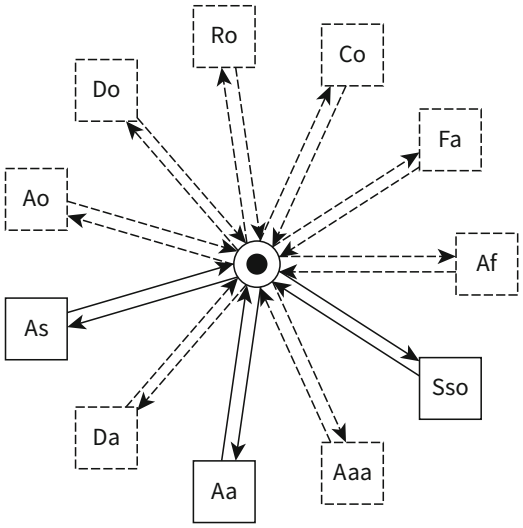


Figure 8.11 Escaping arcs (dashed activities) in the flower model for the prefix of the log trace $\langle As, Aa \rangle$

Let us consider the example of the flower model. We presented the BPMN version of the flower model before in Figure 3.12 and we acknowledged that this model is very imprecise in describing the event log of the running example.

Consider the following event log for the running example, containing only the following three fitting traces:

$$T_1 = \langle As, Aa, Sso, Fa, Ro, Ao, Aaa, Af \rangle$$
$$T_2 = \langle As, Aa, Sso, Fa, Ro, Co, Sso, Ro, Do, Da, Af \rangle$$
$$T_3 = \langle As, Da, Af \rangle$$

Figure 8.11 shows the Petri net version of the flower model.[1] Using dashed lines to indicate escaping arcs, Figure 8.11 highlights the imprecision for the prefix $\langle As, Aa \rangle$. In the state reached in the model, several escaping arcs exist, e.g.:

$$\langle As, Aa \rangle \xrightarrow{Af} \langle As, Aa, Af \rangle$$
$$\langle As, Aa \rangle \xrightarrow{Aaa} \langle As, Aa, Aaa \rangle$$
$$\langle As, Aa \rangle \xrightarrow{Co} \langle As, Aa, Co \rangle$$
$$\ldots$$

[1]The Petri net of Figure 8.11 illustrates where the name "flower model" comes from. The marked place is the heart of the flower and the transitions form the leaves.

Actually, only $\langle As, Aa\rangle \xrightarrow{Sso} \langle As, Aa, Sso\rangle$ is not an escaping arc since $\langle As, Aa, Sso\rangle$ is a prefix of a log trace. By encountering several escaping arcs in almost every reachable state of the flower model, Equation 8.1 clearly has a low (close to 0) value.

For the Petri net of the running example, illustrated in Figure 8.10, very few escaping arcs can be found with respect to a fitting event log (an event log for which all traces correspond to execution sequences of the model). For our small example log, this concerns only execution sequences that correspond to many iterations of the loop and those execution sequences where *Fa* is executed before *Sso* or after *Ro*. Hence, this model is expected to be fairly precise.

The only five escaping arcs for the traces above and the model of Figure 8.10 are:

$$e_1 : \langle As, Aa, \rangle \xrightarrow{Fa} \langle As, Aa, Fa\rangle$$

$$e_2 : \langle As, Aa, Sso\rangle \xrightarrow{Ro} \langle As, Aa, Sso, Ro\rangle$$

$$e_3 : \langle As, Aa, Sso, Fa, Ro\rangle \xrightarrow{Do} \langle As, Aa, Sso, Fa, Ro, Do\rangle$$

$$e_4 : \langle As, Aa, Sso, Fa, Ro, Co, Sso, Ro\rangle \xrightarrow{Co} \langle As, Aa, Sso, Fa, Ro, Co, Sso, Ro, Co\rangle$$

$$e_5 : \langle As, Aa, Sso, Fa, Ro, Co, Sso, Ro\rangle \xrightarrow{Ao} \langle As, Aa, Sso, Fa, Ro, Co, Sso, Ro, Ao\rangle$$

The set $\text{Pref}(L \cap \mathcal{L}_M)$ for the previous log is:

$\langle\rangle$

$\langle As\rangle$

$\langle As, Aa\rangle$

$\langle As, Da\rangle$

$\langle As, Aa, Sso\rangle$

$\langle As, Da, Af\rangle$

$\langle As, Aa, Sso, Fa\rangle$

$\langle As, Aa, Sso, Fa, Ro\rangle$

$\langle As, Aa, Sso, Fa, Ro, Ao\rangle$

$\langle As, Aa, Sso, Fa, Ro, Co\rangle$

$\langle As, Aa, Sso, Fa, Ro, Ao, Aaa\rangle$

$\langle As, Aa, Sso, Fa, Ro, Co, Sso\rangle$

$\langle As, Aa, Sso, Fa, Ro, Ao, Aaa, Af\rangle$

$\langle As, Aa, Sso, Fa, Ro, Co, Sso, Ro\rangle$

$$\langle As, Aa, Sso, Fa, Ro, Co, Sso, Ro, Ao \rangle$$

$$\langle As, Aa, Sso, Fa, Ro, Co, Sso, Ro, Ao, Aaa \rangle$$

$$\langle As, Aa, Sso, Fa, Ro, Co, Sso, Ro, Ao, Aaa, Af \rangle$$

i.e., a total number of 17 prefixes of execution sequences. Hence, the precision approximation according to Equation 8.1 would be:

$$precision_{approx} = \frac{|\text{Pref}(L \cap \mathcal{L}_M)|}{|\text{Pref}(L \cap \mathcal{L}_M)| + |\mathcal{L}_{M_{esc(L)}}|} = \frac{17}{17+5} = 0.7727$$

As a final example, let us consider the same event log $\{T_1, T_2, T_3\}$ used in the previous example, and a slight variation of the process model, shown in Figure 8.12, where an additional branch has been added in the last stage of the process allowing for activities "Cancel application (Ca)" and "Archive reason (Ar)" to be executed in sequence. Apart from the escaping arcs mentioned before, this variation of the running example has the following new escaping arcs:

$$e_6 : \langle As, Aa, Sso, Fa, Ro \rangle \xrightarrow{Ca} \langle As, Aa, Sso, Fa, Ro, Ca \rangle$$

$$e_7 : \langle As, Aa, Sso, Fa, Ro, Co, Sso, Ro \rangle \xrightarrow{Ca} \langle As, Aa, Sso, Fa, Ro, Co, Sso, Ro, Ca \rangle$$

Both escaping arcs correspond to the activity Ca only, since, in the event log with three traces, this activity was not observed in the corresponding prefixes of execution sequences of the model. Hence, the precision of this new model is slightly worse than without the additional branch in Figure 1.3:

$$precision_{approx} = \frac{|\text{Pref}(L \cap \mathcal{L}_M)|}{|\text{Pref}(L \cap \mathcal{L}_M)| + |\mathcal{L}_{M_{esc(L)}}|} = \frac{17}{17+7} = 0.7083$$

The attentive reader may already notice that considering fitting models is very restrictive and perhaps not realistic. However, the metric mentioned above can be extended if alignments are used. For a given trace σ, an optimal alignment $\Gamma(M, \sigma)$ to the model is computed. The execution sequence of the optimal alignment is then

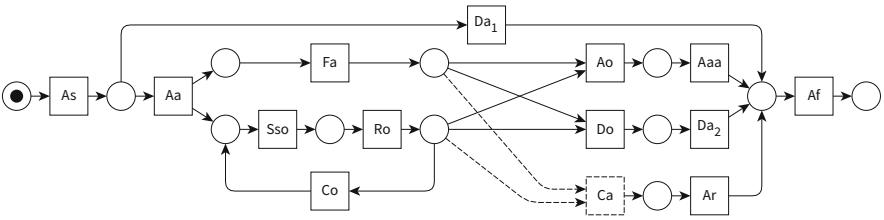


Figure 8.12 Escaping arc for a variation of the running example

used as a basis for computing escaping arcs precision, i.e. the language $L \cap \mathcal{L}_M$ in Equation 8.1 is redefined in the following way:

$$\mathcal{L}_{M_L^a} = \{\gamma \mid \sigma \in L, \gamma \in \pi_2(\Gamma(M, \sigma))\}$$

In other words, for an unfitting trace σ, one execution sequence $\Gamma(M, \sigma)$ that best resembles σ is used to build the joint language between model and log. For instance, for the unfitting trace $\sigma = \langle As, Af \rangle$, the trace $\langle As, Da, Af \rangle$ that corresponds to the optimal alignment of σ with the model in Figure 7.1 is used.

Notice that by relying on an alignment, and not the fitting prefix of a trace (as done in Equation 8.1), potentially many more reachable states of the model are explored, thus improving the accuracy of the metric with respect to the previous one that only considers the fitting part of a trace. However, as optimal alignments are non-deterministic, so is this metric.

This allows us to redefine, using alignments, the language corresponding to escaping arcs of model M over log L, as:

$$\mathcal{L}_{M^a_{esc(L)}} = \{\gamma \cdot \langle a \rangle \mid a \in A,\ \gamma \in \text{Pref}(\pi_2(\Gamma(M, \sigma))),\ \gamma \cdot \langle a \rangle \in \text{Pref}(\mathcal{L}_{M_L^a} \setminus L)\}$$

The precision metric considering alignments is:

$$precision^a_{approx} = \frac{|\text{Pref}(\mathcal{L}_{M_L^a})|}{|\text{Pref}(\mathcal{L}_{M_L^a})| + |\mathcal{L}_{M^a_{esc(L)}}|} \tag{8.2}$$

What is important to realize is that the existence of τ-labelled transitions in the model requires special care. Essentially, these transitions do appear in the alignments, but are filtered out when computing the escaping edges.

In general, and due to the potential existence of not one but many optimal alignments (cf. Section 8.1), the language corresponding to escaping arcs can incorporate all the optimal alignments for every trace. Also, it can incorporate the frequency of every prefix of a trace in the log, so that escaping arcs can be ranked depending on the frequency of their source prefix.

Let us now reason about the adherence of the metric presented to the properties enumerated in Section 8.3. We consider the alignment-based precision metric, since it is more general. By associating an execution sequence with an optimal alignment (which is not deterministic) when constructing the language corresponding to escaping arcs, the metric is not deterministic (this may be changed if all optimal alignments are considered instead of just one). The metric provides a real value in the [0..1] range. The metric is however non-monotonic: A new trace in the log can explore new imprecise parts of the model that where not explored before, so the model's precision can decrease.

8.5.2 Precision Based on Anti-Alignments

In the previous section, the precision based on escaping arcs was presented. One of the main strengths of the metric is its ability to cut the model exploration at most one (or in general, k) steps away from log behaviour, thus bounding at will the exploration of the model state space. In fact, this can be seen as a bounded breath-first-search exploration of the model behaviour. Due to this, the metric may be *short-sighted* in some situations: by only considering the model behaviour that is one step ahead from log behaviour, one can regard as very precise models that really have very deviating traces, since only a prefix of these traces is considered.

In this section we consider a metric for precision that, in contrast, considers the exploration of the model behaviour in a depth-first-search way. We show that this metric compensates the weakness of the previous metric, thus providing a more accurate evaluation of precision. The metric is based on the notion of *anti-alignment*.

An anti-alignment is an execution sequence of a model which differs sufficiently from all the traces in a log. In order to measure how much an execution sequence differs from an trace, one needs a notion of *distance*; actually, a mapping $d : \Sigma^* \times \Sigma^* \rightarrow [0..1]$ is sufficient to define anti-alignments: the other axioms of distance functions (symmetry, triangle inequality, ...) are not required for the definition of anti-alignments. For a log L, we write $d(\sigma, L) = \min_{\sigma' \in L} d(\sigma, \sigma')$. If $L = \emptyset$, then $d(\sigma, L) = 1$.

Definition 18 (Anti-Alignment) A (n, δ)-anti-alignment of a model M w.r.t. a log L and a distance function d is an execution sequence $\sigma \in \mathcal{L}_M$ such that $|\sigma| = n$ and $d(\sigma, L) \geq \delta$.

Notice that the distance function d is a parameter of the previous definition. Typically, Levenshtein's distance (or edit distance) is applied to count how many replacements, deletions and insertions of symbols are needed to obtain $\sigma' \in L$, projected to labelled transitions, starting from anti-alignment $\sigma \in \mathcal{L}_M$. This count is finally divided by the length of the longest of the two sequences (to obtain a value between 0 and 1).

Consider the Petri net shown in Figure 8.13, and the log of Table 8.1. The trace $\langle A,C,G,H,D,F,I \rangle$ is a $(7, \frac{1}{7})$ anti-alignment when considering edit-distance as a distance metric: It can be obtained by inserting G in the trace $\langle A,C,H,D,F,I \rangle$; and the length of the longest trace is 7. Notice that for $\delta > \frac{1}{7}$ there are no anti-alignments for this example, because all other execution sequences of the model have corresponding traces in the log.

The intuition behind the metric based on anti-alignments is as follows. A very precise process model allows for exactly the traces to be executed and not more. Hence, if one trace is removed from the log, this trace becomes the anti-alignment for the remaining log as it is the only execution of the model that is not in the log. This property is used to estimate precision.

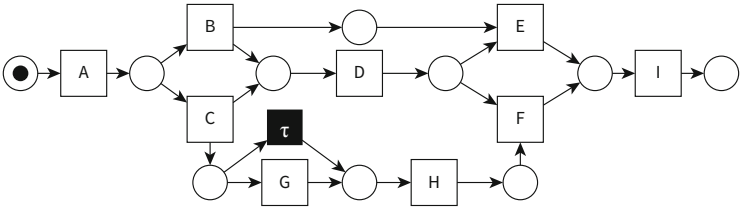


Figure 8.13 An example model

Table 8.1 An event log

Trace	Frequency
⟨A,B,D,E,I⟩	1207
⟨A,C,D,G,H,F,I⟩	145
⟨A,C,G,D,H,F,I⟩	56
⟨A,C,H,D,F,I⟩	23
⟨A,C,D,H,F,I⟩	28

Definition 19 (Trace-Based Precision) Let L be an event log and M a model, such that $L \subseteq \mathcal{L}_M$. Trace-based precision can be defined as follows:

$$P_t(M, L) = 1 - \frac{1}{|L|} \cdot \sum_{\sigma \in L} d(\sigma, \gamma^{d,mx}_{|\sigma|}(M, L \setminus \{\sigma\}))$$

where $\gamma^{d,mx}_n(M, L)$ is one complete anti-alignment, such that $d(\gamma^{d,mx}_n(M, L), L)$ is maximal and the length is less than or equal to n.

Informally, for each trace σ in the log, a maximal anti-alignment γ for the model M and the log without that trace $L \setminus \{\sigma\}$ is computed. This anti-alignment is guaranteed to reach the final marking m_f and hence represents an element of $\mathcal{L}_M$. Then, the distance between σ and γ is computed, which is then averaged over the log, *not* taking into account the relative frequencies of the traces in the log. If the language of the model equals the log, then the anti-alignment γ is equal to σ for every σ, hence the precision is 1. If for every trace σ, an anti-alignment can be produced which has maximal distance from σ, the precision is 0.

A perfectly fitting log is assumed in Definition 19, i.e. $\sigma \in \mathcal{L}^{|\sigma|}_M$ and hence $\gamma^{d,mx}_{|\sigma|}(M, L \setminus \{\sigma\})$ exists. This can be ensured by obtaining an alignment between the log and the model first and then taking the projection of the optimal alignment to the model, rather than the trace itself. As for the escaping arcs precision presented in the previous section, the same discussion on the use of alignments applies here.

Frequencies of traces are not considered as the comparison is between the language of the model and the recorded traces. Observing one trace more frequently than another should not influence the precision of the model as the amount of unrecorded behaviour does not change.

In trace-based precision, the length of the anti-alignment considered is bounded by the length of the removed trace σ. This guarantees that an anti-alignment exists in

the log without trace σ, but also limits the possibility to see imprecise executions of the model that are much longer than the lengths of the traces. Therefore, a log-based precision metric can be defined, which uses an anti-alignment of the model with respect to the entire log of a much greater length than the longest trace recorded in the log.

Definition 20 (Log-Based Precision) Let L be an event log and M a model. Log-based precision can be defined as follows:

$$P_l^n(M, L) = 1 - d(\gamma_n^{d,mx}(M, L), L)$$

where n represents the maximal length of the anti-alignment, typically of the order of several times the length of the longest trace in the log.

The log-based precision metric uses a single anti-alignment of considerable maximum length to determine the amount of behaviour allowed by the model, but not recorded in the event log. The final precision metric is a weighted sum of log- and trace-based precision.

Definition 21 (Precision) Let L be an event log and M a model. Anti-alignment-based precision can be defined as follows:

$$P(M, L) = \alpha P_t(M, L) + (1 - \alpha) P_l^n(M, L)$$

The precision metric has two parameters, α, indicating the relative importance of the trace-based vs. the log-based part, and, n, indicating the maximum length of the log-based anti-alignment. Typically, one may choose $\alpha = 0.5$ and $n = 2 \cdot \max_{\sigma \in L} |\sigma|$.

Let's consider the Petri net shown in Figure 8.13 again, with the log of Table 8.1. Earlier, we identified the trace $\langle A, C, G, H, D, F, I \rangle$ as a $(7, \frac{1}{7})$ anti-alignment. Furthermore, when leaving one trace out, we get the following anti-alignments[2]:

σ	$\gamma_{\lvert\sigma\rvert}^{d,mx}(M, L \setminus \{\sigma\})$	γ projected	$d(\gamma, \sigma)$
$\langle A,B,D,E,I \rangle$	$\langle A,B,D,E,I \rangle$	$\langle A,B,D,E,I \rangle$	0
$\langle A,C,D,G,H,F,I \rangle$	$\langle A,C,G,H,D,F,I \rangle$	$\langle A,C,G,H,D,F,I \rangle$	$\frac{2}{7}$
$\langle A,C,G,D,H,F,I \rangle$	$\langle A,C,G,H,D,F,I \rangle$	$\langle A,C,G,H,D,F,I \rangle$	$\frac{2}{7}$
$\langle A,C,H,D,F,I \rangle$	$\langle A,C,\tau,H,D,F,I \rangle$	$\langle A,C,H,D,F,I \rangle$	0
$\langle A,C,D,H,F,I \rangle$	$\langle A,C,\tau,D,H,F,I \rangle$	$\langle A,C,D,H,F,I \rangle$	0

The trace-based precision is $P_t(M, L) = \dfrac{1 + \frac{5}{7} + \frac{5}{7} + 1 + 1}{5} = \frac{31}{35} \approx 0.886$ and the log-based precision is $P_l^{14}(M, L) = 1 - \frac{1}{7} = \frac{6}{7} \approx 0.857$, hence overall precision

[2]Note that for the edit distance between the anti-alignment and the removed trace, the trace is first projected onto labelled elements, i.e. the τ transition is removed first.

with $\alpha = 0.5$ for this model and log is $P(M, L) = 0.5 \cdot \frac{31}{35} + 0.5 \cdot \frac{6}{7} \approx 0.871$. In contrast, the escaping arc precision metric introduced in the previous section evaluates to 0.96 for this instance, since very few escaping arcs are detected.

The anti-alignment trace-based metric for precision satisfies the properties enumerated in Section 8.3. It is deterministic whenever a deterministic choice is made among the set of maximal complete anti-alignments queried. Also, the metric provides a real value between 0 and 1. Finally the metric is monotonic: Adding traces to the log (in the model) can only increase (decrease) or retain the precision metric (for a sufficiently large value of n).

Bibliographic Notes

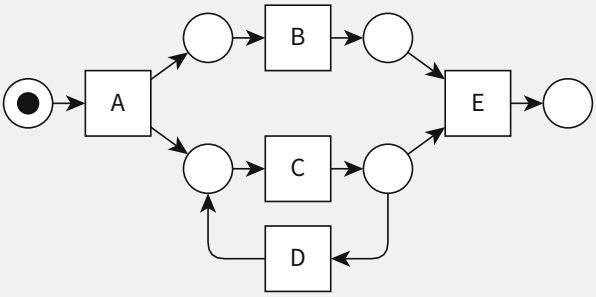
The two metrics introduced in this chapter to estimate the precision of a process model with respect to an event log were originally presented in [4] (escaping arcs-based) and [121] (anti-alignment-based). There are other works related to the evaluation of precision; the work in [126] uses the notion of *artificial negative events* as a strategy to enrich every prefix of a trace in the event log with those events that are not expected to occur, and query the process model in the corresponding state. When event logs fall into the category of big data, a scalable but less accurate way to estimate precision was presented in [58].

8.6 Exercises

8.A) Computing all optimal alignments

Given the process model shown below, and the trace $\langle A, D, E \rangle$:

- What is the cost of an optimal alignment?
- Provide several optimal alignments.



8.B) Fitness calculation metrics

Given the process model and trace of the previous exercise:

- Compute the fitness of the trace using token replay fitness. For that, transform first the process model into a Petri net.
- Compute the alignment-based fitness.
- Compute the move-model fitness.
- Compute the move-log fitness.

8.C) Precision calculation based on escaping arcs

Given the process model of Figure 8.12 and the event log corresponding to the following three fitting traces:

$$T_1 = \langle As, Aa, Sso, Fa, Ro, Do, Da, Af \rangle$$
$$T_2 = \langle As, Aa, Sso, Fa, Ro, Co, Sso, Ro, Ao, Aaa, Af \rangle$$
$$T_3 = \langle As, Da, Af \rangle$$

Compute the precision based on escaping arcs.

8.D) Precision calculation based on anti-alignments

Given the process model of Figure 8.13 and the event log corresponding to the following three fitting traces:

$$T_1 = \langle A,B,D,E,I \rangle$$
$$T_2 = \langle A,C,G,D,H,F,I \rangle$$
$$T_3 = \langle A,C,D,H,F,I \rangle$$

Compute the precision based on anti-alignments.

8) In the lab: tool support for metric computation and alignment visualization

Check out the lab session to practise with tools that compute metrics relating event logs and process models. Also, tools for visualizing alignments are used.

http://www.conformancechecking.com/CC_book_Chapter_8

Chapter 9
Advanced Alignment Techniques

As already acknowledged in previous chapters, alignments became a pivotal element for conformance checking. The techniques seen in Chapter 7 represent the basis for alignment computation, but the attentive reader will realize that the aforementioned techniques work on the basis of a particular algorithmic perspective. In this chapter, we aim at describing several alternatives for computing alignments in order to complement the algorithmic perspective assumed in Chapter 7.

In particular, the following variations of alignment computation are considered. First, in Section 9.1 we show how alignments can be made multi-perspective, so that not only control flow but also other perspectives are considered. Then, we show in Section 9.2 how to compute alignments in an online scenario. The chapter then turns the focus onto how the complexity of computing alignments can be alleviated: in Section 9.3 a decomposition technique to divide and conquer the problem of alignments is reported. Then, in Section 9.4 the computation of alignments using structural theory is described. Finally, the computation of alignments from two different process representations is presented in Section 9.5.

9.1 Incorporating Other Perspectives

In the discussed techniques in this book so far, we have focused on the control flow aspect of processes. Besides the correct ordering of activities inside a process, however, we can also capture additional information in the process model in the form of business rules that further restrict the execution. In general, we distinguish between the data, cost, resource, and time perspectives. Failing to take into account these perspectives in conformance checking can be problematic, since violations

J. Carmona et al., *Conformance Checking*,
https://doi.org/10.1007/978-3-319-99414-7_9

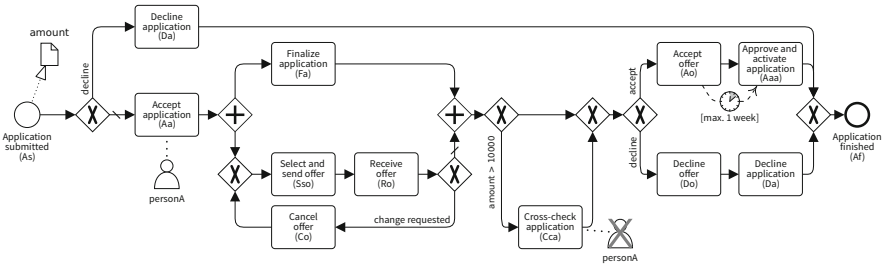


Figure 9.1 BPMN model of our running example with additional data, resource and temporal dependencies

of rules considering these perspectives cannot be detected by only analysing the control flow of a process.

Broadly speaking, *process data* of individual cases is one perspective. This corresponds to the event attributes, as we defined them in Definition 1. In our example, the requested *loan amount* might trigger certain business rules. Also, a business rule may be that when a received document is not signed, we need to send it back to collect the signature. The existence of a signature in a form can be stored and processed by current technology, and fed back to event data. This way, process data can play a decisive role in the execution of processes.

In the following, we look at an enriched process model to reflect additional data constraints. We see a rule capturing data conditions annotated on the arc to the new "Cross-check application" activity in Figure 9.1. Here, the optional activity "Cross-check application" (Cca) should be always executed, when the amount is higher than 10,000.

Additionally, there might be compliance regulations in place that impose certain rules on the resource allocation inside a process. One well-known rule is the *four-eyes-principle* specifying that two related activities should be performed by different people. We see an example of the rule in our model in Figure 9.1. The process participant who performs "Accept application" (*Aa*) is called *personA*. Later in the process, "Cross-check application" (Cca) must be performed by someone other than *personA*.

Besides data and resource rules, the temporal aspect of processes plays an important role when facing time-critical processes. Checking conformance with respect to temporal rules is required when we need to ensure that we reach a certain state in a reasonable time frame. Many service-level agreements (SLAs) specify time constraints. A common SLA could be that we want to guarantee a response time of less than two weeks. To help achieve that SLA, we might specify that it is not allowed to waste time after accepting an offer (*Ao*). Instead, we want to approve and activate the application (*Aaa*) within 1 week after the acceptance; cf. Figure 9.1.

Following this line of reasoning, the work by Mannhardt et al. [64] addresses the need to cater for multiple perspectives in conformance checking. Therefore, they extended the $A^\star$ algorithm to work on Petri nets with data (DPN-net). We omit

the mathematical formalism here and focus on the motivation, the intuition of the approach, complexity implications, and open research challenges in this area.

The extension of conformance checking to data, resource constraints, or time constraints is natural, as these additional dimensions are also part of the process. Why would a process model that precisely states that activity "Finalise application" should be done after "Accept application" not be able to express *who* should be doing it (maybe the same person that accepted it), *when* (maybe latest 1 week later), or *what* the permissible data ranges are for a particular data entry?

In fact, there are language extensions that cover all these dimensions in models to gain more expressiveness. The additional expressiveness comes at a price of understandability. The question of how we can best enrich process models with more information, without impacting their readability and understandability, is an ongoing challenge for researchers and practitioners. Currently, much of the burden to make these expressive models understandable rests on the shoulders of tool vendors. Techniques like selecting only the desired part and applying filters to different perspectives are helpful to divide and conquer the complexity.

Conformance checking with multiple perspectives is agnostic of the modelling notation and the visual representation of the additional constraints. The algorithms only need to have access to the rules as input to the search. The output is an alignment that balances between violations of the different perspectives considered. Control flow and data (also capturing time and resources) are typical examples to consider in multi-perspective conformance checking. Exactly as it is possible to penalise different log moves and model moves, we can also set the costs of violations to data rules.

Let us consider again the model in Figure 9.1. We observe the trace described in Table 9.1. Tracking the path along the model, we see that the recorded case reads $\langle As, Aa, Fa, Sso, Ro, Ao, Aaa, Af \rangle$. This trace if fully compliant to the *control flow* of the model, if we disregard the additional perspectives. Taking into account the other perspectives, however, deviations arise.

The model specifies that a cross-check of the application should be done if the amount is higher than 10,000 and also that the person performing this additional check should not be the person that performed the "accept application" activity. This is not met in the trace of application "A0054", as the cross-check is missing. Further, consider the time constraint of a maximum 1 week delay in the model between the "Accept offer" activity and the "Approve and activate application" activity. The recordings in Table 9.1 show that it took almost 8 days to approve and activate the application after acceptance.

These violations with regard to the model's additional perspectives need to be handled in the alignment. The question is, how can we best align the observations with the rules that regard data, in combination with the alignment in the control flow perspective? In comparison with the control flow alignment, when aligning the data perspective, we reconsider the data values or application of rules so that the trace is compliant with the enhanced model. It turns out that the machinery we have for computing alignments on the control flow level can be adjusted to also take into account the other perspectives.

Table 9.1 Example of a case in the loan application process that is compliant to our running example in Figure 1.3, but not compliant to the data and resource-aware model in Figure 9.1

Event	Application	Offer	Activity	Amount	Signed	Resource	Timestamp
e_{13}	*A*0054		Application submitted	€12,000		<user>	Feb. 1, 12:31
e_{14}	*A*0054		Accept application			John	Feb. 1, 12:32
e_{22}	*A*0054		Finalise application			John	Feb. 3, 09:00
e_{37}	*A*0054	5401	Select and send offer	€11,000		John	Feb. 5, 12:32
e_{42}	*A*0054	5401	Receive offer		YES	John	Feb. 10, 10:00
e_{54}	*A*0054	5401	Accept offer			John	Feb. 10, 10:04
e_{64}	*A*0054		Approve and activate application			John	Feb. 18, 09:05
e_{65}	*A*0054		Application finished				Feb. 18, 09:06

In the example trace for application "A0054" in Table 9.1, we can make the case compliant to the cross-check rule by assuming that the offer selected and sent (*Sso*) was actually about 10,000. Then, the data rule referring to the amount being > 10,000 would be false. Therefore, the absence of the "Cross-check application" activity would not be an issue.

Hence, saying that the data value is incorrectly recorded as 11,000 instead of the true value of 10,000 is one possible explanation of the detected deviations. Another explanation is that the value is correct, but the additional "Cross-check" activity was omitted in the case, which results in a model move in the control flow alignment. Thus, the alignment needs to balance between these explanations. We can again use the concept of costs to capture the deviations and their relation to one another. If the analyst has a good reason to believe that data values are correct, the costs for changing data values can be set much higher than the cost of additional model moves or log moves in the multi-perspective alignment.

Table 9.2 shows two possible explanations side by side in a multi-perspective alignment of trace "A0054". The deviations are highlighted with boldface font. The cost function that evaluates the optimality of these alignments can be defined to balance the perspectives. For example, we can specify that a cost function c penalises a non-conformance in a data value with 1, and also an existence of a log move or model move with a cost of 1. Let (s_L, s_M) denote an element of an

Table 9.2 Two possible alignments explaining the trace for application "A0054" in Table 9.1 with the model Figure 9.1

Alignment a_1	
log trace	execution sequence
As *@Feb. 1, 12:31* (Res: <user>, Amount: €12,000)	*As* *@Feb. 1, 12:31* (Res: <user>, Amount: €12,000)
Aa *@Feb. 1, 12:32* (Res: John)	*Aa* *@Feb. 1, 12:32* (Res: John)
Fa *@Feb. 3, 09:00* (Res: John)	*Fa* *@Feb. 3, 09:00* (Res: John)
Sso *@Feb. 5, 12:32* (Amount: €11,000)	*Sso* *@Feb. 5, 12:32* (Amount: €**10,000**)
Ro *@Feb. 10, 10:00* (Signed: YES)	*Ro* *@Feb. 10, 10:00* (Signed: YES)
Ao *@Feb. 10, 10:04* (Res: John)	*Ao* *@Feb. 10, 10:04* (Res: John)
Aaa *@Feb. 11, 09:05* (Res: John)	*Aaa* ***@Feb. 10, 11:03*** (Res: John)
Af *@Feb. 11, 09:06*	*Af* *@Feb. 18, 09:06*

Alignment a_2	
log trace	execution sequence
As *@Feb. 1, 12:31* (Res: <user>, Amount: €12,000)	*As* *@Feb. 1, 12:31* (Res: <user>, Amount: €12,000)
Aa *@Feb. 1, 12:32* (Res: John)	*Aa* *@Feb. 1, 12:32* (Res: John)
Fa *@Feb. 3, 09:00* (Res: John)	*Fa* *@Feb. 3, 09:00* (Res: John)
Sso *@Feb. 5, 12:32* (Amount: €11,000)	*Sso* *@Feb. 5, 12:32* (Amount: €11,000)
Ro *@Feb. 10, 10:00* (Signed: YES)	*Ro* *@Feb. 10, 10:00* (Signed: YES)
≫	**Cca** ***@Feb. 10, 10:01*** (Res: **Carla**)
Ao *@Feb. 10, 10:04* (Res: John)	*Ao* *@Feb. 10, 10:04* (Res: John)
Aaa *@Feb. 11, 09:05* (Res: John)	*Aaa* ***@Feb. 10, 11:03*** (Res: John)
Af *@Feb. 11, 09:06*	*Af* *@Feb. 18, 09:06*

alignment. A possible cost function can be [64]:

$$c(s_L, s_M) = \begin{cases} 1 & \text{if}(s_L \neq \gg \wedge s_M = \gg) \text{ (move in log)} \\ 1 + |\textit{written variables}| & \text{if}(s_L = \gg \wedge s_M \neq \gg) \text{ (move in model)} \\ |\textit{unequal written variables}| & \text{if}(s_L \neq \gg \wedge s_M \neq \gg) \text{ (synch. move)} \\ 0 & \text{otherwise} \end{cases}$$

That is, the cost function extends the standard cost function by penalizing:

- Model moves that write to variables, where the number of written values adds to the cost, and
- Synchronous moves that refer to different variables, where the number of different written variables adds to the cost.

If we apply this cost function, we get the costs for the two alignment alternatives a_1 and a_2 as follows. Alignment a_1 has two data variables changed in the alignment to make it fit to the modelled constraints:

1. The amount was reduced to allow skipping the required cross-check activity. (+1)
2. The timestamp of the approve and activate application activity was adjusted to fit to the time constraint of less than one week. (+1)

In contrast, the alignment a_2 has the following deviations:

1. A model move for the mandatory activity to cross-check the application was added, to reflect the four-eyes-principle before accepting the offer. (+1)
   - A time value was added to the model move. (+1)
   - A resource value was added to the model move. (+1)
2. The timestamp of the activity to approve and activate the application was adjusted to fit to the time constraint of less than one week. (+1)

In total, we get cost 2 for alignment a_1 and cost 4 with alignment a_2 in this example. We could, however, say that, for compliance reasons, the cost of changing data values in this process would be significantly higher, e.g., 10. In that case, a_1 would have cost 11, and alignment a_2 would still have cost 4. *As* the cost function is flexible, we can adjust it to balance corrections to time, resources, data (on a variable-based granularity), and also to the costs of different log and model moves, as in the cost-based alignment discussed in Chapter 7.

This extension of the control flow-based alignments to multiple perspectives is a generalization of the concepts to find the cost-optimal alignment for a case. Indeed, if we set the costs for data, resource, and time deviations to 0, we end up with the original optimal control flow alignment. Let us next have a quick look into how we can derive the values for data, time, and resource variables to fit the model.

We already know that the $A^\star$-algorithm can help us find the cost-optimal path through the synchronous product of the trace model and the process model (Chapter 7). In addition, we need to make sure that the data constraints in the

model are fulfilled. Intuitively, we would prefer the values that are closer to the original values, when we have an interval to pick from. This fits nicely with known optimization frameworks such as ILP (see Chapter 5) or *mixed integer linear programming* (MILP). Therefore, one solution for this problem is to solve an MILP at every step of the $A^\star$-search to assign values to the variables that best fit the constraints. The proposed solution by Mannhardt et al. [64] does exactly this and furthermore treats time and resources the same way as it treats data variables. In particular, the idea of the techniques is to only look at the written values, and ensure that the constraints imposed by the model up to the current state are met during the search. The solution to each MILP contains variable assignments that meet the constraints.

Furthermore, computing alignments can be expensive, as the number of possible paths through the synchronous product is exponential in the trace length. Intuitively, it should be clear that when adding additional (orthogonal) dimensions to the problem, the complexity increases. In fact, if no modification to the basic $A^\star$ search is done, a solution that is optimal in one dimension could be suboptimal in other dimensions.[1] Nevertheless, we can still apply the $A^\star$-algorithm in the multi-perspective setting because the costs are non-decreasing (the cost function is not allowed to be negative). In this way, the search can simply store the current cost at a certain state and explore the most promising candidates first.

Conceptually, we can treat the time perspective similarly as ordinal data in multi-perspective alignments. Yet, there are some subtle differences and challenges remaining. Imagine a rule stating that a patient is only allowed to continue a treatment after one hour has passed since the administration of a certain drug. In this case, when the records show that some treatments were continued after only 15 min had passed, the simplistic assumption that we can fix one data value (the timestamp) raises another problem. If we fixed the timestamp of the following event and delay it by 45 min to meet the temporal rule, we can accidentally swap the order of this event with the following events.

Iterative side effects such as the one illustrated before make the topic of multi-perspective conformance checking a very challenging one, and therefore interesting research needs to be carried out in the future in this regard.

9.2 Online Conformance Checking

The conformance checking techniques seen so far only allow for *a posteriori* analysis: The amount of (non-)conformant behaviour is computed after the completion of the case. However, this a posteriori analysis may not be acceptable in several

[1] Recently, it has been shown how the technique can be extended so that an optimal balanced alignment can be computed, at the expense of increasing the size of the search space with the notions of *control flow successors* and *augmentation with variable assignments* [64].

contexts, when deviations need to be repaired immediately, before the case finishes. This opens the door to study the conformance checking problem from an online perspective.

Most techniques available nowadays require a complete trace in order to calculate their conformance. From a usability point of view, however, this represents an important limitation: If the respective case is already finished, the counter-measures needed to fix any deviation can be implemented only at a very late stage. Here, we drop such a requirement and present the computation of conformance for *running* cases. Therefore, if a deviation from the modelled behaviour is observed, the problem is noticed immediately, which allows for an immediate response. When these errors are accumulating, the "seriousness" of the case is raised, thus providing stronger alerts for that case.

We assume to have an *event stream* which, basically, is a *data stream* of events. According to [9, 14, 49], a data stream consists of an unbounded sequence of data items which are generated at very high rate. To cope with such data streams, in the literature, often the following assumptions are made: (1) each item is assumed to contain just a small and fixed number of attributes; (2) algorithms processing data streams should be able to process an infinite amount of data, without exceeding memory limits; (3) the amount of memory available to an algorithm is considered finite, and typically much smaller than the data observed in a reasonable span of time; (4) there is a small upper bound on the time allowed to process an item, e.g. algorithms have to scale linearly with the number of processed items: often the algorithms work with one pass of the data; (5) stream sources are assumed to be stationary or evolving. The literature reports several algorithms for the analysis of data streams [9, 44, 45]. However, typically these works cope with different problems, such as classification, frequency counting, time series analysis, and change diagnosis (concept drift detection).

Let C denote the universe of case identifiers, and $\mathcal{A}$ denote the universe of activities. An event stream $\mathcal{S}$ is an infinite sequence over $C \times \mathcal{A}$, i.e. $\mathcal{S} \in (C \times \mathcal{A})^*$. A pair $(c, a) \in C \times \mathcal{A}$ represents an event, i.e. activity a was executed in context of case c. $\mathcal{S}(1)$ denotes the first event that we receive, whereas $\mathcal{S}(i)$ denotes the i-th event.

Consider the Petri net in Figure 9.2, with transitions referring to six activities, namely: "Register request (Rr)", "Check ticket (Ct)", "Examine claim (Ec)", "Pay compensation (Pc)", "Reject claim (Rc)" and "Decide (D)", which appears twice in the model. For this model, we have the stream $\mathcal{S}_1$ shown in Figure 9.3, related to the same activities. Observe that event $(3, D)$ is emitted first ($\mathcal{S}_1(1) = (3, D)$), event $(4, Rr)$ is emitted second, etc. Our knowledge after receiving the third event, i.e. $\mathcal{S}_1(3) = (5, Rr)$, w.r.t. case 5 is different from our knowledge after receiving the fifth event. After the third event, for case 5, we observed $\langle Rr \rangle$, whereas after the fifth event this is $\langle Rr, Ec, Ct \rangle$.

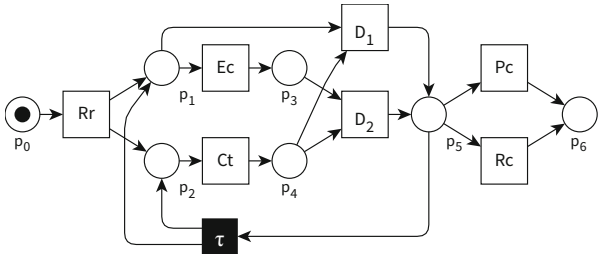


Figure 9.2 Example Petri net (adopted from [114]) with initial marking $[p_0]$ and final marking $[p_6]$

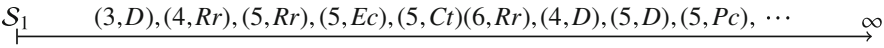


Figure 9.3 Example event stream $\mathcal{S}_1$

Assume that we have only seen the first three events, i.e. $(3, d)$, $(4, a)$ and $(5, a)$, of $\mathcal{S}_1$. The only activity seen for case 5 is activity a. An optimal alignment for case 5 is[2]

event stream	Rr	$\gg$	$\gg$	$\gg$
execution sequence	Rr	Ct	D_1	Rc

After observing the fourth event, i.e. $(5, Ec)$, an optimal alignment for case 5 is

event stream	Rr	Ec	$\gg$	$\gg$	$\gg$
execution sequence	Rr	Ec	Ct	D_2	Pc

In both situations, the costs of the alignments is 3. However, after observing the first nine events on stream $\mathcal{S}_1$ we obtain activity sequence $\langle Rr, Ec, Ct, D, Pc\rangle$ for case 5 with the corresponding optimal alignment:

event stream	Rr	Ec	Ct	D	Pc
execution sequence	Rr	Ec	Ct	D_2	Pc

Thus, since the knowledge we possess about cases changes over time, computing conventional alignments prior to case completion is expected to lead to an overestimation of the true alignment costs. Hence, the techniques presented in Chapter 7 must be adapted for live (i.e., non-finished) cases, in order to avoid underestimating the degree of conformance, and more importantly, to avoid *false*

[2]Remember that the model trace corresponding to an alignment must reach the final marking; see Chapter 7.

event stream	Rr	Ct	D
execution sequence	Rr	Ct	D_1

event stream	Rr	$\gg$	Ct	D
execution sequence	Rr	Ec	Ct	D_2

Figure 9.4 Two prefix alignments for $\langle Rr, Ct, D\rangle$ and the Petri net of Figure 9.2. Note that the end bar on the right is missing to indicate these are prefix alignments and not complete alignments

positives from a deviation perspective. The important idea is to drop the requirement of reaching the final state of the model after executing the execution sequence induced by the alignment. Instead, we require only that the final state is reachable from the state assumed with the partial execution sequence. This way, we are moving from traditional alignments to *prefix alignments*, which are specifically designed to incorporate trace incompleteness.

Figure 9.4 shows two examples of prefix alignments. Notice that the notion of optimality is still valid for prefix alignments: the left prefix alignment is preferred over the right prefix alignment, but also preferred over any other prefix alignment that has log moves, since it only has synchronous moves, i.e., it has minimal cost for replaying the incomplete trace $\langle Rr, Ct, D\rangle$.

To compute prefix alignments based on the event stream, the following steps are taken. If an event related to a certain trace is received, it is checked whether there is a previously computed prefix alignment for that trace. If the event corresponds to a log move, i.e. because the activity simply has no corresponding task in the process model, such a log move is appended to the prefix alignment. If this is not the case, the marking in the process model is fetched, corresponding to the previous prefix alignment.

For example, given prefix alignment $\langle(Rr, Rr)\rangle$ based on the Petri net of Figure 9.2, the corresponding marking is $[p_1, p_2]$. If the event is the first event received for the trace, the marking $[p_0]$ is obtained. If it is possible to directly fire a transition within the obtained marking with the same label as the activity that the event refers to, a corresponding synchronous move is appended to the previously computed prefix alignment. Otherwise a shortest-path algorithm is applied. In order to be applied in the streaming context, the backtracking needed in the search to guarantee optimality is limited.

More in detail, assume we receive the i-th event (c, a) and let m be the marking obtained by executing the model part in the current prefix alignment. If there exist transitions with label a, yet none of these transitions is enabled in m, the algorithm simply tries to find the shortest path to explain a from m. In order to possibly reconsider local decisions that may degrade the quality of the prefix alignment obtained, the search can start from reverting the current alignment up to a maximal revert distance k and start the shortest path search from the corresponding marking.

Consider Figure 9.5, where we depict a prefix alignment for $\langle Rr, Ec, X, Ct, D\rangle$ and the Petri net of Figure 9.2 (X is a new activity here that has not been observed before and is not referenced by any transition of the model).

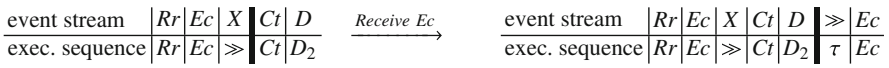


Figure 9.5 Partially reverting ($k = 2$) the prefix alignment of $\langle Rr, Ec, X, Ct, D\rangle$ and the model of Figure 9.2 when of receiving a new event related to activity Ec. The part before the thick rule is not considered when computing the new prefix alignment

Assume a new event is received that states that activity Ec follows $\langle Rr, Ec, X, Ct, D\rangle$ and a revert window size of $k = 2$ is used. Note that the marking reached by the left prefix alignment is $[p_5]$.

The alignment is allowed to be reconsidered for the last two moves, i.e. the moves (D, D_2) and (Ct, Ct) are reverted, reaching the marking $[p_2, p_3]$, from which a search for the shortest path to better explain Ec is computed. The result is of the search is $\langle (Ct, Ct), (D, D_2), (\gg, \tau), (Ec, Ec)\rangle$, depicted on the right-hand side of Figure 9.5. Note that after this step, the window shifts, i.e. two moves have been appended to the current prefix alignment and thus (D, D_2) and (Ct, Ct) are no longer considered upon receiving of new events.

By focusing on prefix alignments instead of alignments, the requirements for an online approach stated at the beginning of this section are addressed. Still, the approach is strongly influenced by the partially reverting window parameter and cannot guarantee optimal prefix alignments unless $k = \infty$.

Excursion 18

Alignments in presence of Velocity: the second V of Big Data

Velocity in Big Data refers to the frequency of incoming data that needs to be processed. In general, one important feature of techniques dealing with a stream of events that arrive at high rates is the ability to store a bounded representation of it, so that memory requirements are under a certain limit. Unavoidably, this causes part of the information to be lost.

In this section, we have seen how the computation of alignments can be adapted so that a stream of events containing the prefixes of several live cases is considered. The aforementioned ability to bound memory requirements is implemented by controlling the window parameter that allows us to revert only part of the previously taken decisions.

9.3 Decomposition-Based Alignments

A well-known concept in computer science to tackle challenging problems is to divide and conquer. The divide and conquer principle can be found in different algorithms for example for sorting (e.g. quicksort [54]).

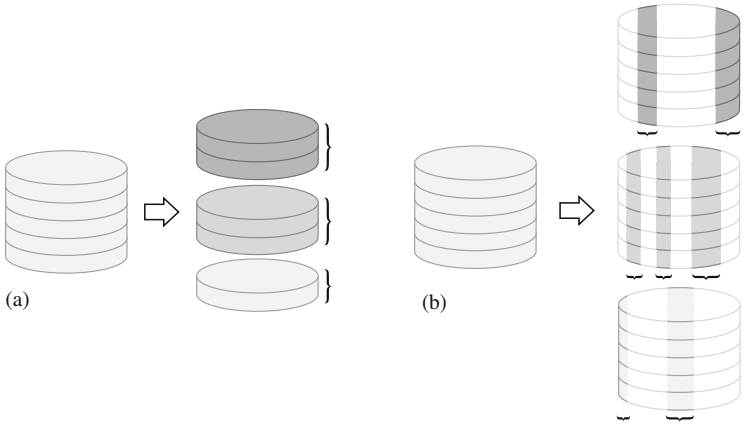


Figure 9.6 Different ways to decompose an event log. For illustration, a log is vertically stacked of traces represented as discs. (**a**) Vertical decomposition of entire traces. (**b**) Horizontal decomposition by slicing the traces by event groups

When we are looking at event logs that contain longer traces, the exponential complexity of the alignment algorithm can make the problem of finding a cost-optimal alignment impractical. Therefore, it is important to find ways of how we can solve conformance questions also for bigger event logs and process models. Obviously, one way to speed up computation is to parallelize the computation of an event log by horizontally splitting the log into multiple pieces. That is, we can look at the set of different trace variants and partition them into multiple groups. Provided that we have a scalable infrastructure at hand, e.g., multiple computing nodes in a cloud, we can distribute the event log and separately align each sub-log to the model to detect conformance errors. This is shown in Figure 9.6a on the left-hand side.

In Figure 9.6b the more interesting scenario of decomposition is depicted that splits the traces in the event log horizontally into partially overlapping pieces that, when put together, cover the original traces. The decomposition is driven by the model that can be decomposed into different *sub-nets*. These sub-nets are constructed by partitioning arcs in the net in such a way that:

- arcs connected to the same place are in the same sub-net,
- arcs connected to the same τ-labelled transition are in the same sub-net, and
- arcs connected to transitions that have the same label, which is assigned to more than one transition in the net, are in the same sub-net.

Each sub-net contains the arcs and their connected nodes.

We can think of sub-nets as puzzle pieces that can be connected at their boundary transitions to reconstruct the original model. Figure 9.7 shows a maximal decomposition of Figure 7.1 into sub-nets. Relying on these sub-nets, decomposing the alignment of an event log to the model is straightforward. For each sub-net, we collect the corresponding events of a trace by projection. For the trace

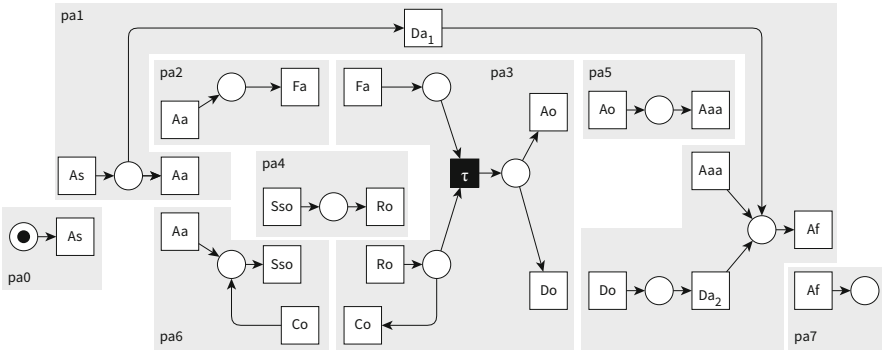


Figure 9.7 A maximally decomposed Petri net for the running example

pa0	⟨ *As*							⟩
pa1	⟨ *As*,	*Aa*,					*Aaa*, *Af*	⟩
pa2	⟨	*Aa*,	*Fa*					⟩
pa3	⟨		*Fa*,		*Ro*,	*Ao*		⟩
pa4	⟨			*Sso*,	*Ro*			⟩
pa5	⟨					*Ao*,	*Aaa*	⟩
pa6	⟨	*Aa*,		*Sso*				⟩
pa7	⟨						*Af*	⟩
trace	⟨ *As*,	*Aa*,	*Fa*,	*Sso*,	*Ro*,	*Ao*,	*Aaa*, *Af*	⟩

Figure 9.8 Projected traces corresponding to the sub-nets shown in Figure 9.7

$\langle As, Aa, Fa, Sso, Ro, Ao, Aaa, Af \rangle$ and the sub-net shown in Figure 9.7, we get the projections as shown in Figure 9.8.

We can now compare each projected sub-trace to the respective sub-net in Figure 9.7. Let us look at sub-net *pa6* and the part of the trace that corresponds to the activities inside *pa6* (i.e., $\{Aa, Co, Sso\}$). The resulting trace is $\langle Aa, Sso \rangle$ as also shown in Figure 9.8. Aligning the projected trace to the sub-net is now an easy exercise, as we decreased the model size and trace size considerably. In this particular example, an alignment with only synchronous moves is possible. We can see that by simply replaying the sub-trace $\langle Aa, Sso \rangle$ on the sub-net *pa6* by using only synchronous moves. After firing *Aa*, the place in *pa6* is assigned a token and the transition *Sso* becomes enabled and can be fired. Decomposed conformance checking is now reduced to computing these piece-wise alignments of all sub-nets with the corresponding projected traces.

Let us reflect here on this decomposition approach. Is it possible to decompose the alignment problem in this way using sub-net and obtaining the same guarantees on the optimality of the alignment result? If this would be true, then the complexity of the problem could be considerably reduced. In the original work that introduces sub-nets for decomposed conformance checking a proof is provided for the fact that

decomposing the problem yields zero alignment cost (i.e., perfect fitness), if and only if, the actual non-decomposed alignment cost of a trace is also zero. In the other settings, when the cost is higher, the decomposition technique yields a lower bound of the actual cost. That is, the cost is underestimated by the decomposition, because the sub-nets do not include all dependencies if treated individually.

More recently, building on this idea of decomposed conformance checking, a solution for getting optimal alignments was proposed by Verbeek et al. [127]. It turns out that the costs of the decomposed alignments can only deviate from the optimal costs for the entire model when the asynchronous moves involve the boundary transitions of the sub-nets. When only transitions within a sub-net are causing non-conformance—that is, the log and model moves are framed by synchronous moves—the sum of the costs on each sub-net equals the cost of the optimal alignment of the entire model. In other words, the decomposition-based results are equivalent to the non-decomposed results in this setting. With this insight, it is possible to devise a method to run the fully decomposed alignment and only repeat alignment computation for the parts of the trace, which have violations on the sub-net boundaries. The affected sub-nets are then merged and the alignment computation is repeated on the merged part. In the worst case, this approach falls back to the non-decomposed method. On average, however, the number of traces affected would be expected to become smaller and smaller, such that the fully-merged step needs to be only repeated for a few traces (if at all).

Excursion 19

Alignments in presence of volume: the first V of Big Data

Volume in Big Data refers to the size of the input data source. Distributed processing is enabled by dividing the problem into smaller subproblems that can be treated individually. This expands the size of problems that can be dealt with considerably.

In principle the decomposition technique described in this section allows, under certain conditions, to divide and conquer the problem of computing alignments. For instance, one can see how unprecedented data sets, i.e., process models containing several hundreds of nodes, can be handled by decomposition-based alignment techniques [74].

9.4 Structural Theory to Compute Alignments

A summary of the techniques for computing alignments of Chapter 7 is that alignment computation can be cast as a reachability problem in Petri nets: find the shortest execution sequence leading to the final marking of the synchronous product.

The reachability problem for Petri nets (is marking m' reachable from m?) is a well-known problem. It has been studied from several angles, but one important branch is the linear characterization of the reachable markings provided by the marking equation, as it has been described in Section 6.2.1. Clearly, the fact that the set of states characterized by the marking equation is a superset of the real set of states ($\mathsf{RS}(N) \subseteq \mathsf{PRS}(N)$), makes any approach that relies only on the marking equation a semi-decision technique. Yet, the marking equation enables useful techniques for conformance checking: We already have seen in Chapter 7 that alignment computation can be accelerated with the help of the marking equation, by computing lower bounds to the distance function at every node of the search space, so that some parts can be discarded in the exploration.

In spite of the aforementioned problem, the marking equation can be adapted to explore the search space of the synchronous product without passing through negative, unrealizable markings. We show that this requires making all the intermediate markings passed by a solution of the marking equation explicit in the ILP problem, i.e. by adding constraints to the ILP that require that all these markings be non-negative. Unfortunately, this adaptation implies that the number of variables of the ILP problem grows significantly (the number of transitions in the alignment times the size of the solution vector), which then incurs a high penalty in memory and time for the ILP solver. Furthermore, the length of the optimal alignment is not known in advance. However, instead of computing the execution sequence from the initial to the final marking in one shot, an incremental computation of the global execution sequence can be applied.

The idea is illustrated in Figure 9.9: Starting in the initial marking, use the marking equation to iteratively compute a prefix $\vec{x}$ of limited size that avoids passing through negative markings. The search is always towards the final marking as the tail of the current prefix (dashed lines starting at $m_1, m_2, \ldots, m_{k-1}$), again using the marking equation but now only as an oracle to reach the final marking. This way, the search for an alignment is done by skipping over states that would be investigated in an $A^\star$ search. With the help of a simple example, we illustrate this sketched iterative algorithm based on the marking equation to compute alignments.

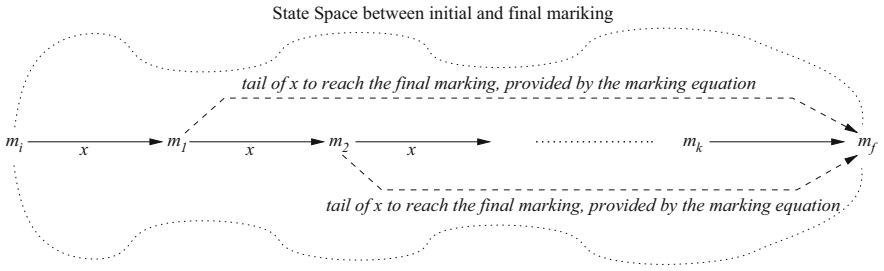


Figure 9.9 Overview of the iterative alignment computation through the marking equation. In the figure, x denotes the solution of the marking equation computed in markings $m_1, m_2, \ldots, m_k$. For each case, assignments to x are guaranteed to not contain more than a small number of non-null entries, and denote an execution sequence that only passes through non-negative markings

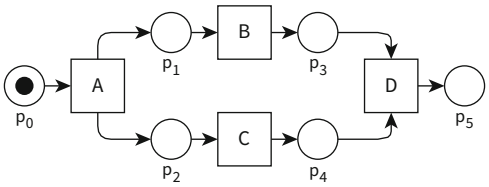


Figure 9.10 Example process model

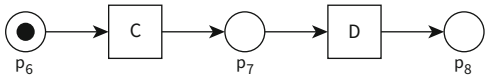


Figure 9.11 Example trace model

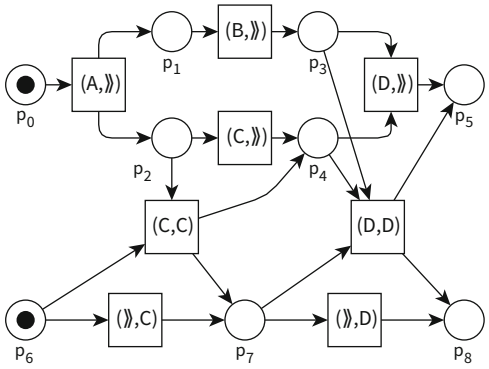


Figure 9.12 Example synchronous product

Consider the example model in Figure 9.10. This model contains a simple parallelism between transitions B and C after A and before D. Now, consider the trace $\langle C, D\rangle$ translated into a trace net as shown in Figure 9.11. Obviously, this trace does not fit the model, as transitions A and B are missing from it. The corresponding synchronous product is shown in Figure 9.12. Here, the transitions (C, C) and (D, D) denote the synchronous moves between model and log.

For this example, an optimal alignment would be:

log trace	A	C	B	D
execution sequence	$\gg$	C	$\gg$	D

For this alignment, the transitions $(A, \gg)$ and $(B, \gg)$ are the model moves, and the transitions (C, C) and (D, D) are the synchronous moves.

The marking equation for the synchronous product is shown below. Here, the columns corresponding to each transition in the incidence matrix are labelled with the corresponding transition label and the rows in the marking are labelled with

place labels:

$$
\begin{array}{c}
\\ p_0 \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \\ p_8
\end{array}
\begin{array}{c}
m_i \\
\left[\begin{array}{c} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{array}\right]
\end{array}
+
\begin{array}{c}
\begin{array}{cccccccc} (A,\gg) & (B,\gg) & (C,\gg) & (D,\gg) & (C,C) & (D,D) & (\gg,C) & (\gg,C) \end{array} \\
\left[\begin{array}{cccccccc}
-1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & -1 & 0 & -1 & 0 & 0 & 0 \\
0 & 1 & 0 & -1 & 0 & -1 & 0 & 0 \\
0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1
\end{array}\right]
\end{array}
\cdot \vec{x} =
\begin{array}{c}
m_f \\
\left[\begin{array}{c} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{array}\right]
\end{array}
$$

By associating a cost function to the firing of each transition (where transitions corresponding to synchronous moves have less cost than the rest, and the cost is minimized), and requiring a solution in $\mathbb{N}$, an ILP (see Chapter 5) can be devised. In this ILP, a solution (an assignment to vector $\vec{x}$) represents a cheapest (in terms of cost) multiset of transitions that take the synchronous product from the initial marking to the final marking.

Yet, when using a symbolic exploration of the state space provided by the marking equation, several problems arise:

1. The vector $\vec{x}$ may be spurious, i.e., does not correspond to a real execution sequence of the synchronous product, passing through markings in $\mathsf{PRS}(N) \setminus \mathsf{RS}(N)$ (see Section 6.2.1).
2. The vector $\vec{x}$ provides only a Parikh representation, but not the real execution sequence. Therefore, one needs to find what execution sequence corresponds to $\vec{x}$.

Excursion 20

The role of the marking equation when computing alignments through $A^\star$

These problems were the reason in Chapter 7 for using the marking equation simply as a means to prune the search space of the $A^\star$, and not as the main exploration aid as we plan to do in this section. In Chapter 7, at each state reached in the exploration, the marking equation is solved to estimate the cost of reaching the final marking. Hence, the risk of dealing with solutions that are spurious is only transferred to underestimating the cost to reach a final marking, i.e., the lower bound used can be simply not achievable, because there is no real execution sequence supporting that cost. In these situations, the exploration cannot stop and further successors of the current state need to be explored (so pruning is not possible).

The marking equation can be refined to address the two problems above: By replicating it to isolate the transition that is fired at each step, the two problems are

solved. The drawbacks of this strategy are twofold: On the one hand, the number of variables of the ILP formulation grows considerably, thus increasing the complexity of the problem. On the other hand, the number of steps to reach the final marking needs to be known in advance. In spite of these two drawbacks, the strategy is used in the technique explained at the end of this section, and hence we formalise the crucial idea: how to fire x transitions by computing the marking reached for the transition fired at each step:

Property 1 (Marking Equation for Executing x Transitions) Let $N = \langle P, T, \mathcal{F} \rangle$ be a Petri net, m_i, m_f two reachable markings of the net and $\sigma = \langle t_1, \ldots, t_x \rangle$ a trace such that $(N, m_i)[\sigma\rangle(N, m_f)$. Furthermore, for $1 \leq j \leq x$, let m_j be such that $(N, m_i)[\langle t_1, \ldots, t_j \rangle\rangle(N, m_j)$. Using the marking equation and general properties of transition firing, the following properties hold:

- The incidence matrix $\mathbf{C}$ can be projected into two matrices: the *production matrix* $\mathbf{C}^+$, which contains the positive elements in $\mathbf{C}$, and the *consumption matrix* $\mathbf{C}^-$, which contains the negative elements in $\mathbf{C}$. Clearly, $\mathbf{C} = \mathbf{C}^+ + \mathbf{C}^-$.
- $\overrightarrow{m_f} = \overrightarrow{m_i} + \mathbf{C} \cdot \vec{\sigma}$ as the trace σ is executable,
- for $1 < j \leq x$ it holds that $\overrightarrow{m_j} = \overrightarrow{m_{j-1}} + \mathbf{C} \cdot \overrightarrow{1_{t_j}}$, i.e. the marking equation holds for each individual transition in the execution sequence,
- for $1 < j \leq x$ it holds that $\overrightarrow{m_i} + \mathbf{C}^+ \cdot \overrightarrow{\sigma_{1..j-1}} + \mathbf{C}^- \cdot \overrightarrow{\sigma_{1..j}} \geq \vec{0}$, i.e. before firing of each transition there are sufficient tokens to fire that transition.

Using the aforementioned property, an algorithm for incrementally computing alignments using the marking equation can be devised. The core idea of this algorithm is to use an ILP that constructs an exact prefix of an alignment of relatively short length, and estimates the remainder of the alignment in the same way the $A^\star$ techniques do. Then, the exact prefix of relatively small length x is executed, computing the resulting marking and repeating the computation until the final marking is reached.

Let us illustrate the idea over the example described at the beginning of this section. We can use the marking equation for firing $x = 2$ transitions in the synchronous product of the example, and then reach a solution in two steps. The solution corresponding to the first step would be:

1. Vectors of variables $\overrightarrow{x_1}, \overrightarrow{x_2}, \overrightarrow{x_3} \in \mathbb{N}^{|T|}$ are used, with the constraints $\sum_{0<i\leq|T|} \overrightarrow{x_1}(t_i) = 1$, $\sum_{0<i\leq|T|} \overrightarrow{x_2}(t_i) = 1$ (so both vectors $\overrightarrow{x_1}, \overrightarrow{x_2}$ are selectors for exactly one transition to fire). The marking equation is required for the first two transitions that are fired (corresponding to the first two vectors of variables), i.e., $\overrightarrow{m_1} = \overrightarrow{m_0} + \mathbf{C} \cdot \overrightarrow{x_1}$, and $\overrightarrow{m_2} = \overrightarrow{m_1} + \mathbf{C} \cdot \overrightarrow{x_2}$. To guarantee non-negativity of markings $\overrightarrow{m_1}$ and $\overrightarrow{m_2}$, the constraints $\overrightarrow{m_0} + \mathbf{C}^- \cdot \overrightarrow{x_1} \geq \vec{0}$ and $\overrightarrow{m_1} + \mathbf{C}^- \cdot \overrightarrow{x_2} \geq \vec{0}$ are required, respectively. The last vector of variables, which is not restricted to have only one transition, is used to estimate the rest of the alignment, i.e., $\overrightarrow{m_f} = \overrightarrow{m_2} + \mathbf{C} \cdot \overrightarrow{x_3}$.

2. The following solution is provided by the ILP solver:

$$\begin{array}{c} \overrightarrow{m_i} \\ \begin{matrix} p_0 \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \\ p_8 \end{matrix}\!\!\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \end{array} + \mathbf{C} \cdot \begin{array}{c} \overrightarrow{x_1} \\ \begin{matrix} A_m \\ B_m \\ C_m \\ D_m \\ C_s \\ D_s \\ C_l \\ D_l \end{matrix}\!\!\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \end{array} = \begin{array}{c} \overrightarrow{m_1} \\ \begin{matrix} p_0 \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \\ p_8 \end{matrix}\!\!\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \end{array} + \mathbf{C} \cdot \begin{array}{c} \overrightarrow{x_2} \\ \begin{matrix} A_m \\ B_m \\ C_m \\ D_m \\ C_s \\ D_s \\ C_l \\ D_l \end{matrix}\!\!\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \end{array} = \begin{array}{c} \overrightarrow{m_2} \\ \begin{matrix} p_0 \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \\ p_8 \end{matrix}\!\!\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \end{array}$$

and $\overrightarrow{x_3}$ reaches the final marking:

$$\begin{array}{c} m_2 \\ \begin{matrix} p_0 \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \\ p_8 \end{matrix}\!\!\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \end{array} + \mathbf{C} \cdot \begin{array}{c} \overrightarrow{x_3} \\ \begin{matrix} A_m \\ B_m \\ C_m \\ D_m \\ C_s \\ D_s \\ C_l \\ D_l \end{matrix}\!\!\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \end{array} = \begin{array}{c} \overrightarrow{m_f} \\ \begin{matrix} p_0 \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \\ p_8 \end{matrix}\!\!\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} \end{array}$$

Now at this point it is known that the prefix $\langle A, C \rangle$ corresponds to an execution sequence in the model that reaches a marking m_2, which corresponds to a model move on A (transition A_m) and a synchronous move on C (transition C_s). From m_2, the marking equation, through vector $\overrightarrow{x_3}$, suggests that there exists an execution sequence that reaches the final marking.

Now we initiate the marking equation using m_2 as initial marking and we repeat the aforementioned step, with the corresponding set of constraints that adopt the same philosophy as the ones used before. The following solution for vectors of variables $\overrightarrow{x_1}, \overrightarrow{x_2}, \overrightarrow{x_3} \in \mathbb{N}^{|T|}$ is found by the ILP solver ($\overrightarrow{x_3} = \overrightarrow{0}$):

$$\begin{array}{c} \overrightarrow{m_2} \\ \begin{matrix} p_0 \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \\ p_8 \end{matrix}\!\!\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \end{array} + \mathbf{C} \cdot \begin{array}{c} \overrightarrow{x_1} \\ \begin{matrix} A_m \\ B_m \\ C_m \\ D_m \\ C_s \\ D_s \\ C_l \\ D_l \end{matrix}\!\!\begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \end{array} = \begin{array}{c} \overrightarrow{m_3} \\ \begin{matrix} p_0 \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \\ p_8 \end{matrix}\!\!\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \end{array} + \mathbf{C} \cdot \begin{array}{c} \overrightarrow{x_2} \\ \begin{matrix} A_m \\ B_m \\ C_m \\ D_m \\ C_s \\ D_s \\ C_l \\ D_l \end{matrix}\!\!\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \end{array} = \begin{array}{c} \overrightarrow{m_f} \\ \begin{matrix} p_0 \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \\ p_7 \\ p_8 \end{matrix}\!\!\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} \end{array}$$

which completes the prefix previously computed up to the final marking, providing the full execution sequence $\langle A, C, B, D \rangle$, corresponding to the alignment discussed above.

In general, for large problem instances and because the length of the execution sequence is not known in advance, the first x fireable transitions should be a low number, and some type of progress should be required for these x transitions, so that at least some of them explain some event, by performing a synchronous or log move. This way, the state space of the synchronous product is explored by jumps of x transitions in a depth-first manner until the final marking is reached (see Figure 9.9). These x firings are locally optimal with respect to the cost function that minimizes the number of asynchronous moves, but in general the method cannot guarantee that the global solution found is optimal: for instance, there may be a path from m_i to m_f in Figure 9.9 that globally has less cost than the concatenation of locally optimal paths x shown in the figure.

In conclusion, the method described in this section can serve as an alternative to the alignment technique provided in Chapter 7, especially when the state space of the synchronous product is very large, and thus the complexity of exploring it explicitly is high. In an extreme setting, when it is not possible to fully explore the state space, methods like the one described in this section can at least provide an alignment (although not guaranteed to be optimal).

9.5 Alignments Beyond Petri Nets

In this last section, we provide two alternative representations that can be used to compute alignments. The first one is event structures, a well-known model for concurrency which facilitates the uniform representation of event logs and process models. The second is mixed-paradigm models, which can be fully declarative, or instead a combination of declarative and procedural constraints.

9.5.1 Event Structures

Process models and event logs are different representations of a process. The computation of an alignment between these two representations may reveal inconsistencies, that sometimes originate from the different perspectives these two representations have: For instance, while process modelling languages like Petri nets or BPMN show the concurrency between activities explicitly, this does not happen for event logs.

A possible alternative is to map both representations into a unified representation, so that the comparison is facilitated. *Prime Event Structures* (PES) is a well-known model for concurrency [77]. In particular, PES can be a unifying language for process models and logs, embedding the particularities of each representation. In

this short summary, we do not provide the details on how to transform a process model or a log into a PES, and refer the interested reader to [46].

A PES is a graph of events, where an event represents an activity execution in the process. Using a PES to define a process model, for a task that occurs multiple times in an execution sequence, each occurrence is represented by a different event. The order of events is defined via three binary relations: (1) Causality ($a < b$) indicates that event a is a prerequisite for b ; (2) Conflict ($a\#b$) implies that a and b cannot occur in the same execution sequence; (3) Concurrency ($a||b$) indicates that no order can be established between a and b.

Let us consider a different, simple BPMN process, depicted in Figure 9.13. We show the corresponding PES in Figure 9.14. The directed arcs represent direct causality, while dashed (undirected) arcs denote conflict. Concurrency can be derived from causality and conflict relations [77]. A possible execution sequence of the PES shown in Figure 9.14 starts with event e_0 (A), then e_1 (B), then e_3 (D), and finally e_5 (E) and e_9 (H). Notice that by including event e_3, which is in conflict with event e_2, the latter (and the events caused by e_2) cannot be included in this execution sequence.

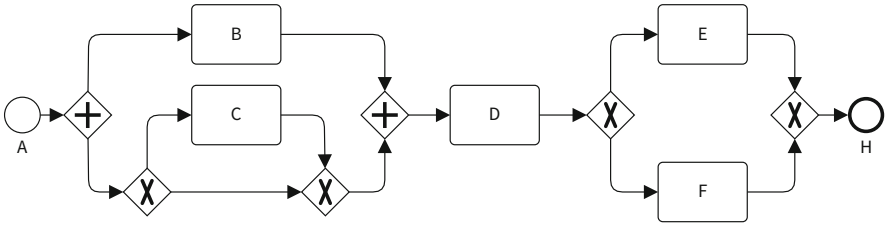


Figure 9.13 Simple process, taken from [46]

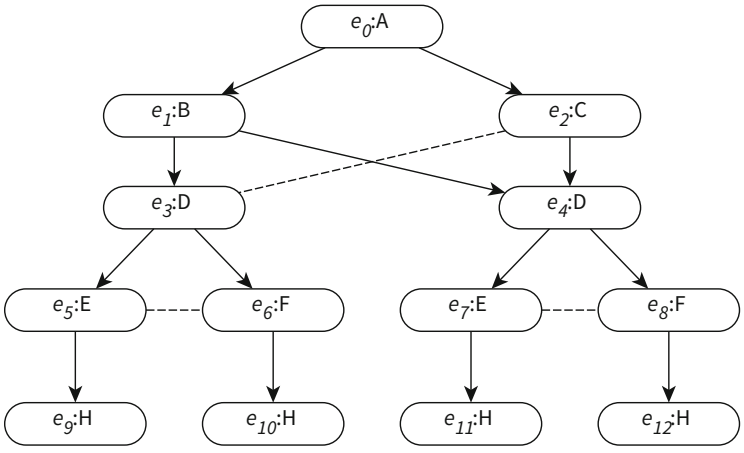


Figure 9.14 PES for the process model of Figure 9.13, taken from [46]

Table 9.3 Event log from which the PES shown in Figure 9.14 can be generated

Trace	Frequency
$\langle A, B, C, D, E, H \rangle$	1
$\langle A, B, C, D, F, H \rangle$	1
$\langle A, C, B, D, E, H \rangle$	1
$\langle A, C, B, D, F, H \rangle$	1
$\langle A, B, D, E, H \rangle$	1
$\langle A, B, D, F, H \rangle$	1

Likewise, traces in a log can also be converted into a PES. In Table 9.3 we show an event log from which the PES of Figure 9.14 can be obtained.

Once both process model and event log are mapped into a PES, their conformance can be assessed. The intuition behind this approach is to compute a *partially synchronised product* of the two PESs. This is a state machine in which the states denote pairs of configurations (each being a set of events) visited during an error-correcting synchronised traversal of the two input event structures (see Figure 9.15). The arcs are labelled with the possible operations between the two event structures: matching an activity provided that both PESs can execute the corresponding events labelled with it, or hiding an event on the left (lhide) or right (rhide) PES.

The process starts from the empty configurations of each PES, and ends with all pairs of maximal configurations of the two input PESs. Figure 9.15b shows a simple example of the computation of the partially synchronised product from two PESs, shown in Figure 9.15a. Every state is annotated with the configuration of each one of the PESs (collection of causality-closed events executed until the state), and the sequence of matched activities so far. The goal is to find the set of optimal event matchings, i.e., the set of states corresponding to maximal configurations for which the number of matched activities is maximal. For that, similarly to the procedure in Chapter 7 for optimal alignments, the $A^\star$ heuristic search technique can be used. Notice that, once these optimal event matchings are computed, alignments in the form reported in this book can be generated. For instance, the alignment corresponding to the maximal configuration on the left branch of Figure 9.15b is:

$$\begin{array}{|c|c|c|c|c|} \hline \mathcal{E}^l & A & B & \gg & C \\ \hline \mathcal{E}^r & A & \gg & B & C \\ \hline \end{array}$$

Interestingly, the approach based on PESs allows for assessing *behavioural conformance*, so that not only the fine-grained deviations (as the ones arising from alignments) can be extracted, but also new mismatch patterns can be obtained from a tailored partially synchronised product. For instance, the existence of additional behaviour like unmatched repetitions, either in the event log or in the process model, can be computed using this approach. The reader is referred to [46] for an in-depth presentation of this approach.

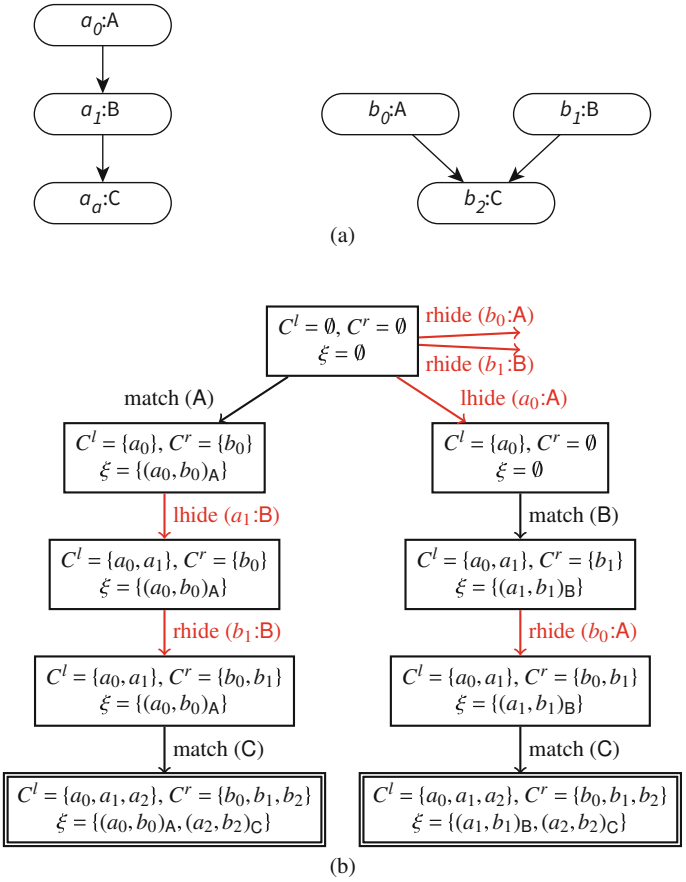


Figure 9.15 (**a**) Two example PESs $\mathcal{E}^l$ and $\mathcal{E}^r$, respectively. (**b**) Fragment of the partially synchronised product of $\mathcal{E}^l$ and $\mathcal{E}^r$. Example taken from [46]

9.5.2 *Mixed-Paradigm Model Alignments*

So far, in this book, we considered so-called procedural process models, i.e. process models that describe which activities should be performed in which order. An alternative modelling paradigm is offered by *declarative* models, see also Excursion 10. Such models capture activities and rules which each case of the process should obey. However, as long as all rules are obeyed, anything is possible.

An example of such a rule is a *response constraint*, as discussed already in Section 4.1 in the context of rule-based approaches to conformance checking. The response constraint between activity A and activity B essentially states that "once activity A has been executed, activity B must be executed afterwards, at least once, before completing the case". However, it does not specify how often B should occur, or when, only that it should be afterwards. The trace $\langle A, A, A, B \rangle$ satisfies this constraint.

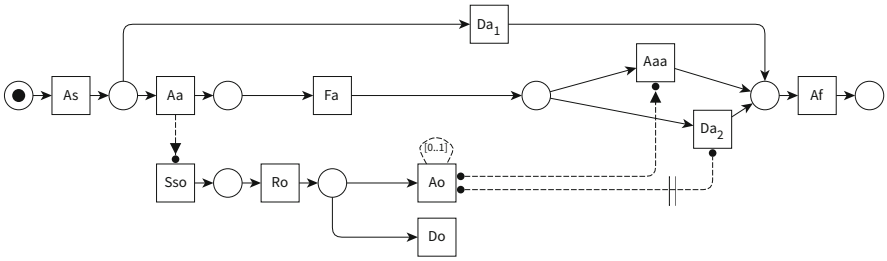


Figure 9.16 Mixed paradigm representation of the running example

Declarative models have been explained in detail in [79] and conformance checking of declarative models has also been discussed in [20]. Furthermore, the same technique can be used as outlined in Chapter 7; however, no heuristic function is available to efficiently search the entire state space of the model.

To overcome this issue, mixed-paradigm models can be considered. Mixed paradigm models combine Petri nets on the one hand and declarative models on the other hand. Figure 9.16 shows a mixed paradigm model for our running example. The top part is the same as before, i.e. a Petri net explaining the control flow for the application. However, the transitions referring to the offer have been separated to allow for multiple offers to be selected and sent to the customer in parallel to each other.

There are four declarative constraints in the model expressing the following:

- "Select and send offer (*Sso*)" cannot be executed unless "Accept application (*Aa*)" was executed at least once before. This is a so-called *precedence constraint*. Note that once *Aa* has fired, *Sso* can occur an arbitrary number of times, or not at all.
- "Accept offer (*Ao*)" can be executed at most once. This is a *unary constraint* on that transition.
- The rightmost "Decline application (*Da*)" and "Accept offer (*Ao*)" cannot *co-exist* in the same case, i.e. if the customer accepts an offer, the application will not be declined and if the application is declined, the customer cannot accept an offer.
- "Accept offer (*Ao*)" is succeeded by "Approve and activate application (*Aaa*)" which is expressed by the *succession constraint*. In other words *Aaa* cannot occur before *Ao* occurred, and after *Ao* occurs, *Aaa* must occur.

Clearly, the mixed paradigm model allows for more behaviour than the original procedural model. For example, multiple offers can be selected and sent to the customer in parallel, without the need to wait for the customer. Furthermore, the "Cancel offer" activity is no longer needed, since every offer needs to be either accepted or declined. Due to the nature of the model, the company can decline the application before a response was given to all outstanding offers. The "not co-existence" constraint then ensures that all remaining offers will eventually be registered as declined.

Consider the trace $\langle As, Aa, Fa, Sso, Sso, Ro, Ro, Ao, Aaa, Do, Af \rangle$. In the Petri net model of Figure 7.1, this trace would lead to the following alignment:

log trace	As	Aa	Fa	Sso	Sso	Ro	Ro	$\gg$	Ao	Aaa	Do	Af
execution sequence	As	Aa	Fa	Sso	$\gg$	Ro	$\gg$	τ	Ao	Aaa	$\gg$	Af

In the model of Figure 9.16 however, this would be a fitting trace in which, after accepting the application, two offers were selected and sent to the customer, of which one was accepted and the other one declined due to a timeout.

Computing alignments on mixed paradigm models is essentially not different than computing alignments for regular Petri nets. The additional constraints only limit the possible firing of transitions, i.e. the state space of the mixed paradigm model is strictly smaller than the state space of the Petri net without the constraints. Hence, in conformance checking, the heuristic based on the marking equation can still be used. During the computation, the decision about whether a transition is enabled or not needs to be made based on a combination of the marking in the Petri net and the state of each constraint.

Aligning event logs and mixed paradigm models becomes more interesting once costs are assigned to violating a constraint, i.e. rather than guaranteeing that at the end of the alignment all constraints are satisfied, we allow for constraints to be violated at a certain cost. For example, we could say that accepting more than one offer is allowed at a certain cost.

Consider the trace $\langle As, Aa, Fa, Sso, Sso, Ro, Ro, Ao, Aaa, Ao, Af \rangle$. In the Petri net model of Figure 7.1, this trace would lead to the following alignment:

log trace	As	Aa	Fa	Sso	Sso	Ro	Ro	$\gg$	Ao	Aaa	Ao	Af
execution sequence	As	Aa	Fa	Sso	$\gg$	Ro	$\gg$	τ	Ao	Aaa	$\gg$	Af

When doing conformance checking on the mixed paradigm model of Figure 9.16 while enforcing all constraints, the alignment would be:

log trace	As	Aa	Fa	Sso	Sso	Ro	Ro	Ao	Aaa	Ao	$\gg$	Af
execution sequence	As	Aa	Fa	Sso	Sso	Ro	Ro	Ao	Aaa	$\gg$	Do	Af

If the default cost function is used, then this alignment has cost 2 due to the mismatch between the "Accept offer" in the trace and the "Decline offer" which is needed to complete the model and satisfy all constraints. However, if we allow the unary constraint to be violated with cost 1, then there is a different alignment, namely:

log trace	As	Aa	Fa	Sso	Sso	Ro	Ro	Ao	Aaa	Ao	Af
execution sequence	As	Aa	Fa	Sso	Sso	Ro	Ro	Ao	Aaa	$Ao!$	Af

In this alignment, the exclamation mark indicates the point where a constraint becomes violated. However, since this only has cost 1, this alignment is better than the previous one.

Bibliographic Notes

The computation of alignments has received quite significant attention in the last few years. Methods for the efficient computation of alignments based on the marking equation of Petri net are described in [109]. The method in [122] is an evolution of the previous work, which has been summarized in Section 9.4. In [15] alignments are computed by maximizing the number of synchronous moves. Another approach, also based on the structural theory of Petri net but with a different perspective, has been presented in [108]: the idea is to perform model-based reductions that help in reducing parts of the model (and correspondingly, the log), so that the computation of alignments can be done more efficiently. An evolutionary technique that has a low memory footprint and can compute multiple alignments has been proposed in [110].

Alignments beyond control flow, so that data attributes are taken into consideration have also been explored. The seminal work on this (discussed in Section 9.1) was presented in [29, 32], and followed up in [64].

An interesting research direction to overcome the challenge of computing alignments is to decompose the problem into pieces, so that the problem is alleviated. The first works on proposing this direction are [112, 113]. This idea has been later extended for a particular decomposition based on Single-Entry Single-Exit (SESE) components in [74]. Also, SESE-based decomposition for data-aware alignment computation has been explored [28].

Very few techniques have appeared for applying conformance checking in an online scenario. One of the first approaches was presented in [125]; it describes how the SESE-based decomposition can be combined with a tailored token replay technique, so that deviations are detected by replaying individual fragments of the process model. The first approach that uses alignments in an online setting was presented in [123], which has been shortly described in Section 9.2. It introduces the notion of prefix alignments and shows how to compute them online. An alternative framework to compute alignments that is based on pre-computing deviations was recently presented in [19]. Another remarkable approach that presents a RESTful service for conformance checking for the case of BPMN process models can be found in [129]. In [21], online conformance checking is done using behavioural patterns obtained through pre-processing the model.

The use of event structures as a unifying notation for event logs and process models represents a new way to align recorded and modelled behaviour [47]. Interestingly, each behavioural difference detected by this approach can be verbalised as a natural language statement.

Alignments can be computed on different representations, like declarative processes [27], optionally including multiple process perspectives [20].

Also, recent approaches transform the internal representation of conformance checking methods into an automaton, so that efficient automata-based techniques can be applied [58, 87].

9.6 Exercises

9.A) Alignments for other perspectives

Given the loan application process shown in Figure 9.1, compute an alignment that also takes into account the data perspective for the following case.

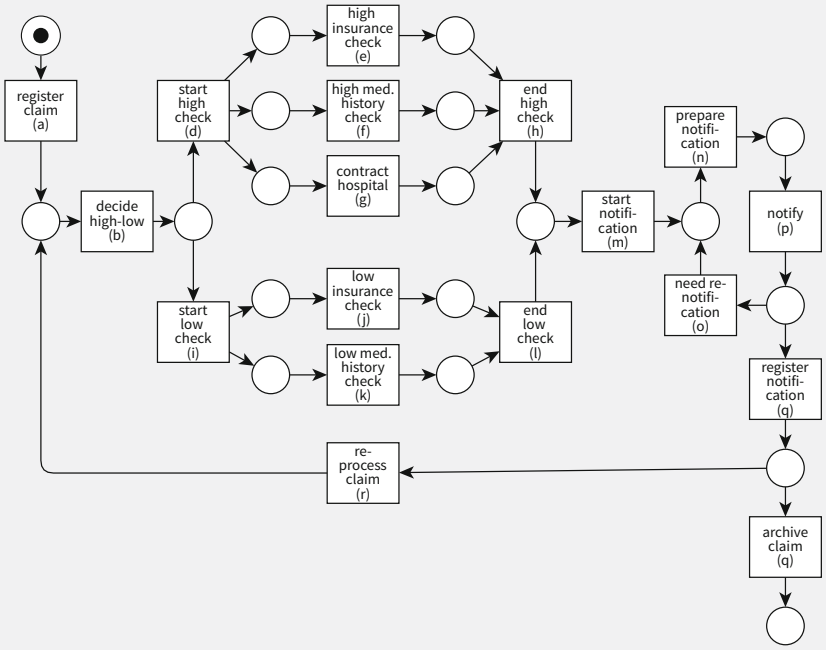
Event	Application	Offer	Activity	Amount	Signed	Resource	Timestamp
e_{17}	A006		Application submitted	€7000		<user>	Feb. 1, 12:31
e_{34}	A006		Accept application			John	Feb. 1, 12:32
e_{43}	A006		Finalise application			John	Feb. 3, 09:00
e_{47}	A006	5407	Select and send offer	€11,000		John	Feb. 5, 12:32
e_{62}	A006	5407	Receive offer		YES	John	Feb. 10, 10:00
e_{68}	A006		Cross-check application			John	Feb. 11, 10:00
e_{74}	A006	5407	Accept offer			John	Feb. 11, 10:04
e_{84}	A006		Approve and act. application			John	Feb. 14, 09:05
e_{85}	A006		Application finished				Feb. 15, 09:06

9.B) Online computation of prefix alignments

Given the Petri net of Figure 9.2 and the partial trace $\langle a, b, b, d \rangle$, compute a prefix alignment.

9.C) Decompositional computation of alignments

Given the following process model:



- Compute a valid decomposition to allow for decompositional computation of alignments.
- Project the trace $T_1 = \langle a, b, d, g, j, k, f, e, h, l, m, n, p, o, n, p, q, s \rangle$ according to the decomposition obtained.
- Decide the fitness of T_1 on the basis of the replay of its projections' replay on the valid decomposition.
- *Do* the same for the trace $T_2 = \langle b, d, j, k, l, f, e, h, l, m, o, n, p, s \rangle$.

9) In the lab: tool support for computing variants of alignments

Check out the lab session to practise with different tools to compute variants of alignments introduced in this chapter.

http://www.conformancechecking.com/CC_book_Chapter_9

Part III
Conformance Checking Applications

Preface to the Third Part

The capability of automatically aligning recorded and modelled behaviour is crucial to detect deviations between both process representations. Although important, in some situations it is not enough to simply report the (dis)agreement. The third part of this book is meant to provide further techniques to enhance, repair and reflect on the process models and event logs used in conformance checking.

Organizations often demand incorporating more information when aligning a log and a model, so that more information about the real process can be projected onto the model. In Chapter 10 some model enhancement techniques are provided which allow for important aspects like performance or decision points to be incorporated in the model. This information is extracted from the additional attributes that often exist in event logs. Additionally, further metrics that complement fitness and precision are provided in Chapter 10 to better understand a process model: generalization and simplicity.

When deviations are detected that pinpoint undesired situations, amending the source of these deviations (either the process model, or the event log) is a reasonable next step. Chapter 11 provides two complementary forms of overcoming such deviations: repairing the process model or repairing the event log. Moreover, once these two operations are defined, a general framework for conformance checking is presented.

Practical tools for conformance checking are considered in Chapter 12. Specifically, we review the conformance checking functionality implemented in ProM, a widely established open-source framework. We also discuss functional and non-functional aspects that help in the assessment of commercial offerings for conformance checking.

Chapter 10
Understanding Processes

After aligning recorded and modelled behaviour, a general perspective of the process can be attained if further dimensions are explored. For instance, the additional information available in an event log on the time elapsed between the events can be crucial to understand the process bottlenecks. Likewise, further event attributes can explain the rules governing the decisions made during the executions of the process, so that hidden strategies with respect to the data are elicited.

Also, it is very important to consider additional quality metrics that complement fitness and precision, which may help in assessing the ability of a model to generalizing the patterns recorded in the event log, or analysing its simplicity.

This chapter is organized as follows: in Section 10.1 a gentle introduction to performance analysis is presented. Then, we provide techniques for decision point analysis in Section 10.2. Finally, Section 10.3 completes the spectrum of quality dimensions, introducing the notion of generalization and simplicity.

10.1 Performance Analysis

The performance of its processes is critical to the success of an organization. Customers waiting for their services and products longer than necessary can become dissatisfied, for example if the delivery date is postponed, or not communicated clearly. Even more so, if the competitors are faster and do manage their processes better, customers are likely to switch to them.

By measuring the actual performance of a process for a specified period, it is possible to compare one period to another. If an organization tries to improve its processes and changes some aspects of a process (e.g., removes an unnecessary activity), they need to have an understanding of the effects of the changes. Ideally, the performance can then be projected onto the new process model, which is the central artefact in process management to communicate about the process.

J. Carmona et al., *Conformance Checking*,
https://doi.org/10.1007/978-3-319-99414-7_10

10.1.1 Basics of Performance Analysis

In this section, we explore how to use alignment techniques that we covered in Part II of this book for performance analysis. To be able to analyse durations of activities and waiting times in between, we need temporal information in the event logs (e.g. in the form of timestamps). If we have an alignment of an event log with respect to a process model, we already know the synchronous moves and we are able to enrich the model using the timing information from the event log. In essence, for every path traversed in a process model, we record how long each activity took (i.e., we record the activity durations), and how much time has passed on the arcs that connect the nodes in the model (i.e., the waiting times between activities).

After collecting the information in the model, a process model with temporal statistics is obtained. In process mining, this procedure is called *model enrichment*. The temporally enriched process model collects the encountered durations per element and aggregates them for quick analysis. Typical performance measures are statistics about the duration of an activity. That is, we are interested in how long activities take on average, or how long they took in the worst case (maximum), or the best case (minimum). One very important measure is the variability of the duration. We can measure for example the standard deviation of a duration to get a feeling for the variance. Besides the summary statistics, we can also display the estimated distribution of the durations.

Figure 10.1 shows exemplary enrichment information. To avoid cluttering the image, we only display some example durations and branching probabilities. Performance analysis can be performed on individual activities and arcs between them, but it can also be performed on milestones in the process that reflect *key performance indicators* (KPIs). For example, the time passed between the receipt of an order and the delivery of the goods is more important for the customer than the time from the receipt of an order to issuing an invoice. Modelling and extracting KPIs in a process model is only possible through accurately mapping actual event data to the process model.

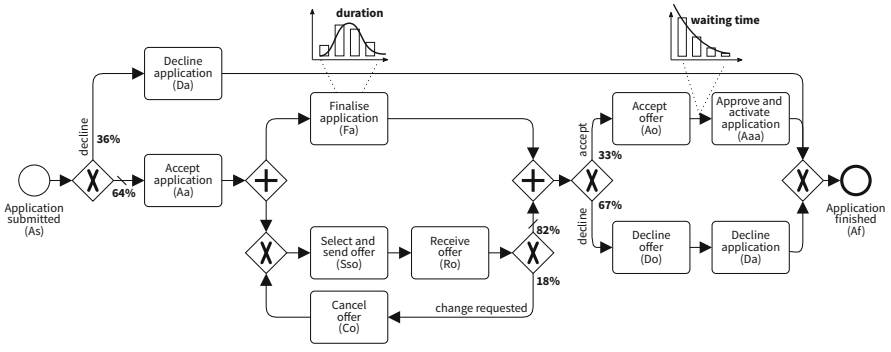


Figure 10.1 Running example process with enriched durations, waiting times and branching probabilities

Excursion 21

What about a synchronous moves?

The simple enrichment procedure, as sketched in Section 10.1.1, does not work well if there are only a few synchronous moves, and we did not discuss what to do with the elements in the log that do not have counterparts in the model to project onto.

One simple solution is to discard asynchronous moves entirely from the analysis and only focus on the synchronous moves. We might risk to discarding valuable data and the aggregate statistics can be biased if the unaligned data behaves differently than the aligned data.

A typical example is a missing record of an activity (say b) in the log, which is required according to the model. This typically results in a model move in the alignment (the activity b is skipped in the log). If there is an activity a before, and an activity c after a, we can think about distributing the time between a and c to the two arcs connecting the activities in sequence. However, things get more complicated when in reality we know that b actually must have occurred before c. Then, we have the problem of *missing data* as it is known in statistics, and we can rely on different *missing data imputation* methods to correct the durations.

If the deviations between the model and the log are frequent, and there is a recurring pattern in the asynchronous moves, we can think about repairing the process model to reflect more what is in the data. Section 11.1 shows how models can be repaired in this way.

10.1.2 Performance Beyond the Temporal Perspective

The time taken to perform a process and the activities therein is important. But there are more perspectives an analyst can look at. The same enrichment procedure as for the temporal aspect can be applied to monetary costs. By projecting the costs of certain activities into the process model, analysts can quickly see which tasks incur the most costs and which ones are cheapest. For a hospital, for example, each treatment is associated to refundable costs, which can also be projected into the process model.

Tools that help in understanding the performance of a process according to different perspectives, provide different views on the data. Through these views, only those perspectives are shown which are relevant to the question at hand, thereby greatly improving understandability of an enriched model.

Another example of a perspective that is commonly considered is the organizational perspective. Using information from the log about the resources involved in and the participants interacting in the process, insights can be obtained into which

activities were performed by whom. An interesting metric in this regard is the count of hand-overs of work, i.e. when was a case handed over from one person or department to the next person or department? By annotating the resources that really performed the activities, we can check whether a new process version indeed requires fewer hand-overs than the previous one.

10.2 Decision Point Analysis

As the previous section has already pinpointed, once the process model and the event log are aligned, further information can be derived from the event log to enrich the model. Where performance analysis focuses on the projection of information onto the model, decision point analysis goes one step further. In decision point analysis, the event log is used to analyse decisions from a data perspective.

Typical data available in the event log is manifested as additional attributes for the events. For instance, in the event log shown in Table 2.2, there are attributes like Amount and Signed, which complement the information about the activity executed. Decision point analysis uses these and other attributes to learn rules that can explain decisions in the process model. For instance, Figure 10.2 shows simple rules that can be incorporated in the process model to explain the decision made in the loan application process, from the perspective of the traces in the event log. The following decisions are elicited on top of the model:

1. Applications are declined if their amount is very small (less than 100 Euros), or when too many loans have been already accepted in the current month (more than 250K Euros).
2. The customer tends to ask for a change when the offer amount is less than 80% of the initial amount. If the customer forgets to sign it, it is also sent again.
3. Finally, loan applications that cannot be resolved in less than 2 months are always rejected.

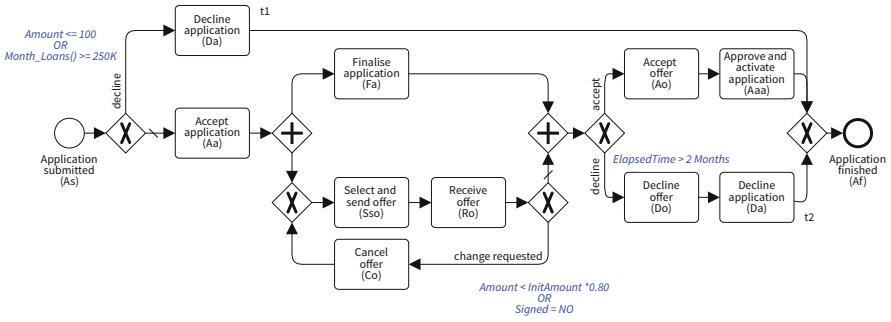


Figure 10.2 Example of decision point analysis for the process model of a loan application

Excursion 22

Data mining and conformance checking: Enhancing process models with data

The digital explosion of data makes it easy to find it almost everywhere, enabling the application of techniques that extract meaning from and assign value to this data. Data mining is a mature area providing a family of techniques to elicit meaning (e.g., in the form of rules, patterns or other structures) from raw data [17].

Input for data mining techniques is data in tabular form, like for instance:

Age	Gender	Weight	Sport	Family Issues	Married	Decision
23	Male	67	Yes	Cancer	No	No
38	Female	55	No	None	Yes	Yes
59	Female	51	No	None	Yes	Yes
50	Male	80	No	Obesity	Yes	No
...	...	...	...	...	...	...

Data mining techniques can be partitioned into two families:

- *Supervised Learning*: In the data there is a special attribute, denoted as *response variable*, that corresponds to the label of the instance considered. In contrast, the rest of variables are denoted *predictor variables*, which are used to explain the response variable. Two different families of techniques are available under supervised learning: *classification* and *regression*. The former assumes the predictor variable to be a category. It is applied when the goal is to classify the instances with the help of the predictor variables. In this section, we show an example of classification techniques, *decision tree learning*. Instead, regression assumes the predictor variable to be numerical. It is applied to find a function that fits the data with the least error.
- *Unsupervised Learning*: When no response variable is available, unsupervised learning is applied. Two families of techniques are available in unsupervised learning: *clustering* and *pattern discovery*. The former is applied to classify the input data into groups, so that the instances of each group are similar, given a notion of similarity. Pattern discovery is used to extract patterns from data, e.g., rules that relate the variables. One of the most popular techniques in pattern discovery is *association rule learning*.

As we have said at the beginning of this section, event logs contain many data attributes that can be used as input for data mining techniques.

In the previous example, apart from the event attributes (Amount, Signed), some other types of attributes can be considered for decision point analysis:

- Case attributes: Attributes that refer to the case, like accumulated elapsed time from the beginning of the case at every event, or the number of times a certain activity has been executed. In our example, the former is used in the decision to decline an application if it cannot be resolved in 2 months.
- Contextual attributes: This information is not usually encountered in the event log, but obtained from other sources of information. For instance, the information on the total amount of Euros corresponding to loans granted in the current month is an example.

We use the example of Figure 10.2 to illustrate how decision point analysis can be performed. Let us first have a look at Table 10.1, which shows an event log in such a way it can be used for decision point analysis. We focus on the first decision found in the process model of Figure 10.2.

In order to learn the rules governing the decision once the application is submitted, a supervised learning problem is generated (see Excursion 22), where the response variable is the next activity to be executed in the case, and the predictor variables correspond to all the attributes available (including contextual attributes, like the amount of Euros in the current month for approved loans). If the data from Table 10.1 is given to a classification technique like *C4.5 algorithm*, a well-known decision tree learning algorithm [85], the tree shown in Figure 10.3 is extracted for predicting the next activity.

Table 10.1 Part of an event log extended for decision point analysis

Case ID	Activity	Timestamp	Amount	Month_Loans	Signed	Next Activity
A5345	*As*	10-02-2017 9:08am	1000	130K	Yes	*Aa*
A5342	*As*	12-04-2017 10:03am	1500	30K	No	*Aa*
A5343	*As*	20-04-2017 11:32am	2000	235K	Yes	*Aa*
A5324	*As*	27-04-2017 2:01pm	1500	250K	Yes	*Da*
A5352	*As*	3-05-2017 7:06pm	3500	15K	No	*Aa*
A5351	*As*	18-05-2017 9:08pm	75	120K	No	*Da*
A5357	*As*	6-06-2017 10:28pm	550	75K	No	*Aa*
A5359	*As*	29-06-2017 10:40pm	7500	255K	Yes	*Da*
A5513	*As*	12-07-2017 9:08am	8000	136K	No	*Aa*
A5434	*As*	19-07-2017 10:03am	100	207K	No	*Da*
A5222	*As*	23-07-2017 11:32am	2500	249K	Yes	*Aa*
A5555	*As*	1-08-2017 2:01pm	5000	0K	No	*Aa*
A5232	*As*	12-08-2017 7:06pm	9000	102K	Yes	*Aa*
A5444	*As*	20-08-2017 9:08pm	6000	160K	Yes	*Aa*
A5213	*As*	28-08-2017 10:28pm	5000	257K	No	*Da*
A5115	*As*	30-08-2017 10:40pm	30	257K	No	*Da*

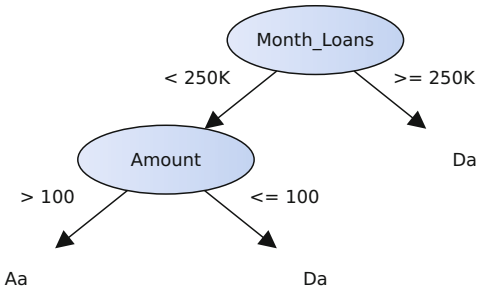


Figure 10.3 Decision tree learned from the data in Table 10.1

From the tree in Figure 10.3, the rule

$$\text{Month\_Loans} \geq 250K \vee \text{Amount} \leq 100$$

can be extracted for predicting the execution of the activity "Decline application". However, notice that in the process model in Figure 10.2, there exist two tasks with the label "Decline application": one at the beginning (t_1, right after the start of the process), and one near the end (t_2, right before the end of the proces). In order to decide at which point the rule should be inserted into the model, we again resort to alignments, which tell us that after activity "Application submitted", the first activity "Decline application" is the most likely candidate and the rule should be added to the first decision point.

An interesting feature of decision tree learning is the capability to include only a subset of the attributes in the final rules. For instance, in the rule learned before, the attribute Signed is not used (see Figure 10.3). This is beneficial to decision point analysis, as in principle only the necessary attributes appear in the rules learned to annotate the process model, which helps readability and comprehension.

10.3 Further Metrics for Comparing Models

In this book, the importance of having metrics that enable evaluating the relation between a process model and an event log has already been emphasized. The two main metrics considered, fitness and precision, are tailored to this. Fitness, on one hand, assesses whether an event log is well represented as a process model and precision, on the other hand, assesses whether the process model represents the behaviour of the event log in a concise manner. In this section, we reflect on two additional metrics, which complement fitness and precision by focusing on a different perspective, namely generalization and simplicity.

10.3.1 Generalization

In Section 3.4.2, a discussion on the behaviour that is modelled but not recorded can be found. With regard to this behaviour, elements in $(\mathcal{L}_M \setminus L) \cap S$ denote behaviour that, in spite of not being recorded, correctly represents the underlying process. Generally speaking, these elements represent an interesting feature of a process model: correctly capturing parts of the system that have not been recorded.

A typical example of parts of the system that were not recorded are loops: When the real process allows for the iterative execution of a set of activities, one cannot assume that all possible executions (in principle, an infinite number) were recorded in reality. Accordingly, in spite of not observing all the possible instances of the loop, a process model for the underlying process is expected to contain a loop over these activities, generalizing the finite behaviour recorded. For instance, let us consider the loan application process that is used through this book. Clearly, an important feature of this process is the iteration over the activities "Select and send offer", "Receive offer" and "Cancel offer", so that they can be done more than once within a case. When confronted with an event log for this process, it is clear that there exists a maximum number of times where these activities have been performed in any case, even though in principle, no upper bound has been explicitly defined. In this context, the process model shown in Figure 1.3, which explicitly contains a loop over these activities, generalizes the recorded behaviour.

Now let us assume that domain knowledge is available with respect to the maximal number of iterations among the activities "Select and send offer", "Receive offer" and "Cancel offer". If at most three iterations are possible, is the process model of Figure 1.3 then a proper generalization of the process recorded in the event log? Clearly, the process model of Figure 1.3 allows also for traces with more than three repetitions over these activities, and therefore, the underlying process is not precisely described. And what if the domain knowledge states that offers can be selected for the first month after the application was accepted, but not longer? In that case, no explicit upper limit is specified, but the number of iterations will be finite.

In summary, process models are expected to generalize the behaviour recorded in event logs. However, the notion of good or bad generalization strongly depends on the system and any comparison between modelled and recorded behaviour should be considered in the context of this system.

Definition 22 (Generalization) A process model M generalizes a log L with respect to system S if some behaviour in $S \setminus L$ exists in $\mathcal{L}_M$.

Although the existence of loops influences the generalization of recorded behaviour, they are not the only factors to consider. Concurrency is another construct related to generalization. For instance, assume that a process contains ten parallel activities. It is very unlikely to observe the $10! = 3,628,800$ possible permutations of executions of these activities in the event log. In spite of observing a considerably

smaller set of permutations, a process model that elicits this concurrency denotes a good generalization of the behaviour considered.

A careful look at Definition 22 reveals a complication when trying to evaluate the generalization dimension: the real process, S, is in general not available. Therefore, the few available metrics for generalization dimension only consider the event log and the process model [18, 121, 126] as input.

Generalization Based on Frequency of Replay

A simple metric to evaluate generalization was presented in [18]. This metric is based on the frequency of each task execution when replaying the traces of the event log. The intuitive idea is that if parts of the process model are infrequently visited during the replay of the log, generalization is bad. For instance, the process model in Figure 10.4 shows frequent tasks with thick lines. Clearly, in this model some parts are not frequent (tasks "Decline application", "Cancel offer", and "Decline offer"). According to the metric from [18] this represents a low generalization capability of the process model.

Generalization Based on Negative Artificial Events

An alternative metric to evaluate generalization is based on the notion of *weighted artificial negative events* [126]. This metric first computes, for every possible prefix of a trace in the event log, the activities that *cannot* occur, denoted as negative events. For instance, in the loan application example used throughout this book, if we consider the prefix of a log trace to be $\langle As, Aa, Sso, Ro, Fa \rangle$, then examples of negative events at the end of the prefix are As, Aa, Da, Fa, Da, Sso, Ro, Aaa and Af, i.e., according to the traces in the log, events relating to these activities are not expected to occur after observing the prefix above. Notice the difference between the concept of negative events with respect to the escaping arcs used to estimate precision: Negative events do not consider the process model, and instead are only computed on the information available in the event log.

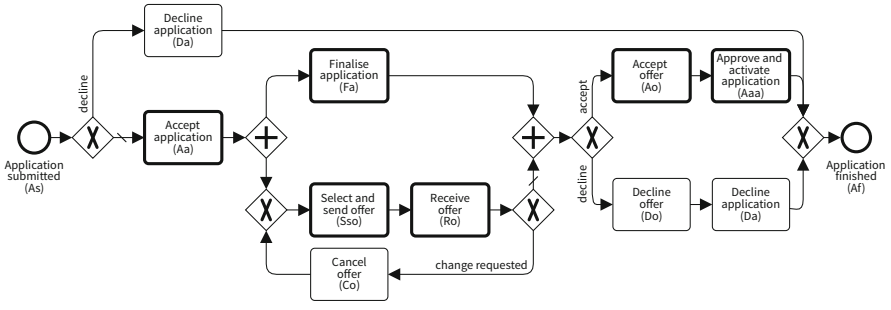


Figure 10.4 Loan application process model with frequent tasks highlighted with thick line

Generalization can be computed from the complement of the set of negative events (i.e. the so-called positive events) at each position using the model. At every prefix of a trace, the allowed generalizations (AG, activities that the model can replay) and the disallowed generalizations (DG, activities that the model cannot replay) are considered. The formula $\frac{|AG|}{|AG|+|DG|}$ is used to estimate generalization.

Generalization Based on Anti-Alignments

In Section 8.5.2, it is shown how precision can be measured using the notion of anti-alignments. Remember that an anti-alignment of a model with respect to a log is an execution sequence of a model, which is as different as possible from the traces in the log. In order to quantify generalization, the approach in [121] considers not only the sequential behaviour that is allowed by the model, but also quantifies how different this behaviour is when considering the state space of the model: Models that introduce new traces without introducing new states are considered to be generalizing.

Structured loops and parallel structures, which are most commonly used to achieve generalization when modelling a system, have the tendency to allow for many different sequential traces while introducing fewer states. In particular, for structured loops, the number of states does not increase with the number of executions of the loop. Also for parallel transitions, the number of states 2^n grows slower than the number of sequences ($n!$).

In the generalization metric in [121], the notion of a *recovery distance* for an anti-alignment is used, which establishes the maximum distance between any of the states reached in the anti-alignment and the states visited by the log.

Recall that an anti-alignment is an execution sequence of the model which is as different as possible from the recorded traces. For precision, we would remove one trace from the log, then compute an anti-alignment for the remaining log and compare the anti-alignment to the removed trace. The more different the anti-alignment is to the removed trace, the less precise the model (see Section 8.5.2).

For generalization, we compare the anti-alignment identified this way with the log and we investigate, at each point in the execution sequence, what the *recovery distance* is. The recovery distance in this context is the maximum number of task executions necessary to get from a state reached by the anti-alignment to any of the states reached by the (aligned) event log.

The combination of the anti-alignment distance and the recovery distance then determine the generalization. If a model is properly generalizing, it is likely that the behaviour recorded in the log covers a significant part of the state space introduced by the generalizing structure. Hence, a previously unobserved trace (the anti-alignment) does not introduce many new states, but rather new paths between existing states, even if the introduced trace is completely different from anything recorded in the log, resulting in a high anti-alignment distance with low recovery distance.

Excursion 23

Evaluating generalization for discovery algorithms

The scope of the previous techniques is a general one: Given an event log and a process model, to estimate how good the model is in generalizing the underlying process. In some situations, process models are not created manually and instead are the result of a discovery algorithm [114]. In these contexts, one can shift the focus, evaluating generalization not on process models, but instead on the discovery algorithms that were used to create these process models.

In machine learning, a learning algorithm generalizes when it is able to predict previously unseen data. In our context, generalization for a process discovery algorithm tests whether discovered process models generate traces that are not present in the log, but that can be produced by the underlying process. One can use *k-fold cross validation* to assess the generalization of a process discovery technique, by partitioning the event log into k parts, and using $k - 1$ parts to discover a process models that would be tested on the part left out. The test simply measures fitness, so that the ability of the learned process model to reproduce the part left out is considered. One can then measure generalization by repeating the experiment for each one of the possible k parts, and averaging the fitness values accordingly. If the algorithm is able to generalize, then the average fitness computed should be high. If in contrast, the algorithm has problems in generalizing, then it should obtain a low average value.

10.3.2 Simplicity

Another metric to evaluate a process model is simplicity: Is the derived process model the most simple explanation of the underlying process? This metric refers to the *Occam's Razor* principle. Simplicity is generally considered to be a property of the model itself and not a relation between the model and an event log. We mention some available metrics for completeness, but we refer to the relevant literature for an in-depth discussion.

One method for measuring simplicity is by analysing the complexity of the underlying graph. In [70], the following metrics have been proposed to measure this complexity:

- *Size*: number of nodes and arcs,
- *Diameter*: length of the longest path from a start node to an end node,
- *Density*: relation between the number of arcs and the maximum number of arcs between all nodes,
- *Connectivity*: ratio between number of arcs and number of nodes,

- *Node Degree*: average or maximum number of nodes each node is connected to,
- *Separability*: ratio between the number of cut-vertices (i.e., the minimal number of vertices needed to disconnect the graph) to the number of nodes.

When taking into account that the graph is in fact describing a process model, some other tailored metrics are considered in [70]:

- *Sequentiality*: number of arcs to/from gateways (i.e., nodes that encode control flow semantics) divided by the overall number of arcs,
- *Structuredness*: the proportion or well-structured parts with respect to the rest (non-structured) in the process model. Well-structured parts can be reduced by applying simple reduction rules,
- *Depth*: average or maximum depth for split/join constructs,
- *Gateway Mismatch*: absolute sum of all input arcs minus output arcs over all pairs of split/join connected gateways,
- *Gateway Heterogeneity*: entropy of the gateways used,
- *Control Flow Complexity*: sums of all choices of a process based on the number of splits of each type and its number of outgoing arcs,
- *Cyclicity*: number of nodes within cycles with respect to the total number of nodes,
- *Token Splits*: number of concurrent threads that can be activated by AND- and OR-splits in the process.

Clearly, any of the aforementioned metrics can be used to compare the simplicity of two process models represented in the same notation, e.g., two process models in BPMN. However, these metrics may be less reliable when the comparison is done across different notations. For instance, a Petri net does not explicitly represent gateways, and therefore some of the metrics in the second list are not applicable.

Bibliographic Notes

Enriching process models for simulation purposes was introduced based on token replay [94], and later based on alignments [115]. Temporal aspects in business processes play an important role that was addressed in different works, e.g. [42, 92, 100, 118]. A state-transition system is enriched with temporal information to predict remaining case duration based on a partial trace in [118]. Based on the latter, [42] shows that the system load, an additional variable counting the number of running cases, can explain much of the variance in the remaining durations. Taking the load into account significantly improves predictions. Enriching Petri nets with stochastic information through alignments is proposed in [88]. The resulting models are called Stochastic Petri Nets, of which different variants have been discussed [69]. Also machine learning models, like neural networks, are trained with event log data to achieve highest accuracy [107].

Decision point analysis was first considered in [50, 96], and since then, several approaches have been presented [12, 23, 26, 30, 39, 48, 56]. A more general

framework which covers decision point analysis and other analysis has been recently proposed [31]. Another interesting direction is the discovery of overlapping rules, which may be useful when insufficient information is available, or in case of non-deterministic data, as the approach in [65] describes.

10.4 Exercises

10.A) Performance

Given the part of a bigger model, as shown in Figure 10.5 and the following log.

Case ID	Event	Time	Date	User
X123	p_{start}	9:45	10.03	Freddy
X123	p_{compl}	10:05	10.03	Freddy
X144	n_{start}	10:00	10.03	Peter
X144	n_{compl}	10:10	10.03	Peter
X188	n_{start}	18:00	10.03	Carla
X188	n_{compl}	09:30	11.03	Carla

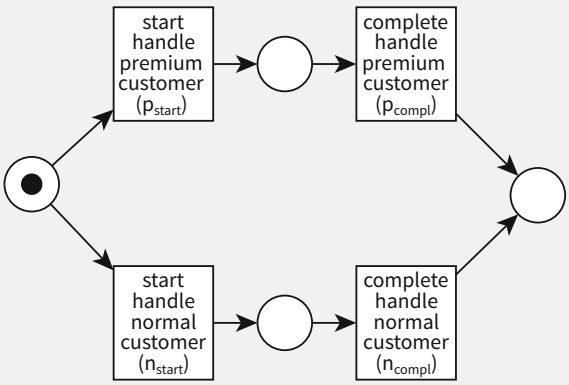


Figure 10.5 Part of a process with customer interaction

1. Calculate the average duration of activity "handle normal customer" based on the given observations.
2. What is the ratio between the two customer types premium and normal?

(continued)

10.A) (continued)

3. Activity "handle normal customer" for case X188 took 15 hours and 30 minutes to finish. What is the most plausible reason for this long activity duration? And what does that mean for handling temporal information derived from raw log data?

10.B) Decision point analysis

Using the process model of the loan application in Figure 1.3, find the rules that explain the decisions to accept or decline an offer according to the data shown in the table below.

Case ID	Activity	Elapsed Days	Amount	Offer	Responsible	Next Activity
A1345	*Fa*	18	1000	900	Darla	*Ao*
A2342	*Fa*	100	1500	1420	Aaron	*Do*
A3343	*Fa*	25	2000	2000	Jane	*Do*
A5324	*Fa*	20	1500	1450	Aaron	*Ao*
A8352	*Fa*	230	3500	3300	Darla	*Do*
A1351	*Fa*	26	7550	7000	Darla	*Ao*
A3357	*Fa*	12	5500	5000	Jane	*Do*
A4359	*Fa*	30	7500	3400	Aaron	*Do*
A5513	*Fa*	34	8000	7900	Aaron	*Ao*
A8434	*Fa*	40	1000	150	Darla	*Do*
A0222	*Fa*	120	2500	2400	Darla	*Do*
A7555	*Fa*	23	5000	4200	Aaron	*Ao*
A1232	*Fa*	15	6000	5800	Jane	*Do*
A1444	*Fa*	140	6000	5900	Darla	*Do*
A2213	*Fa*	17	5000	4600	Aaron	*Ao*
A3115	*Fa*	23	3000	1200	Aaron	*Do*

The columns of the table represent:

- Case ID, Activity and Next Activity: the case of an event, and the activity performed before and after the decision, respectively.
- Elapsed Days: the number of days since the application entered the system.
- Amount: amount of the loan.
- Offer: amount of the last offer made.
- Responsible: person in charge of the loan application.

10.C) Generalization

Given the following event log:

Trace	Frequency
$\langle As, Aa, Sso, Fa, Ro, Do, Da, Af \rangle$	700
$\langle As, Aa, Fa, Sso, Ro, Ao, Aaa, Af \rangle$	545
$\langle As, Aa, Sso, Fa, Ro, Co, Sso, Ro, Do, Da, Af \rangle$	50
$\langle As, Da, Af \rangle$	28

Decide whether the loan application process model in Figure 1.3 generalizes well or not according to the simple precision metric based on frequency of replay from [18], explained in Section 10.3.1.

10.D) Simplicity

Given the following three process models, namely M_1, M_2 and M_3 (Figures 10.6, 10.7, and 10.8).

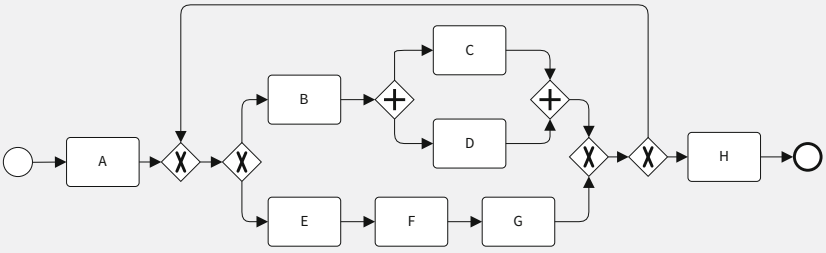


Figure 10.6 M_1

(continued)

10.D) (continued)

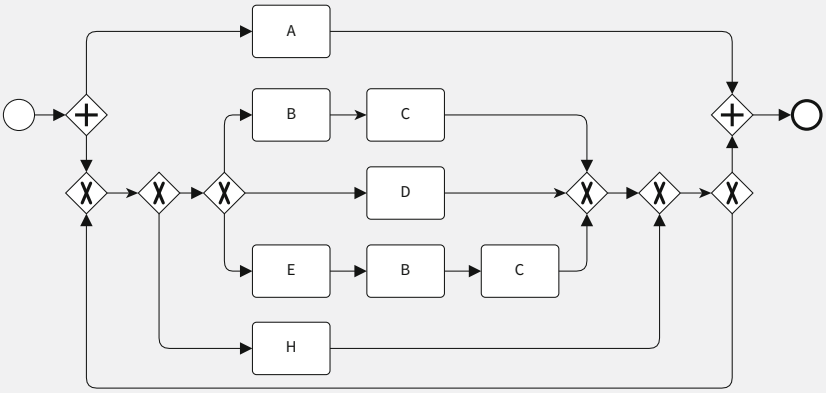


Figure 10.7 M_2

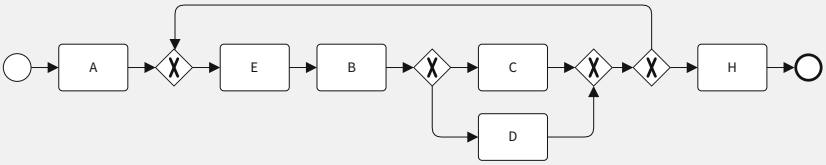


Figure 10.8 M_3

Rank them according to the *size*, *diameter* and *density* simplicity metrics reported in this chapter.

10) In the lab: tool support for understanding processes

Check out the lab session to practise with different tools for understanding processes and implementing techniques introduced in this chapter.

http://www.conformancechecking.com/CC_book_Chapter_10

Chapter 11
Improving Processes Using Conformance Checking

Processes evolve continuously, so it is desirable to guarantee today's process models to be optimally aligned with respect to today's processes as they are recorded. Detecting deviations between the process, as manifested in event logs, and the process model is necessary functionality in an operational setting. So far, conformance checking has been positioned as a diagnostic technique, able to identify deviations, but not able to follow up on these deviations and explain them in the context of the underlying process.

In this chapter, we provide an overview of automated techniques that enable addressing the deviations encountered, so that a better relation can be established between the process model and the process as recorded. This clearly widens the usefulness of conformance checking. In this chapter, we present a symmetrical view to this problem, that builds up on top of the conformance artefacts, and enables the automatic repair of a model (Section 11.1) or a log (Section 11.2), or both. In Section 11.3, the general problem of conformance checking, which also incorporates model or log repair in a continuous setting, is provided.

11.1 Model Repair

When deviations are encountered between a process model and an event log, they may be attributed to the current process model failing to represent the real process. Consequently, the process model should be modified (repaired) to better describe the reality, as recorded in the event log. These deviations can then be used to reconstruct the process model, so that the updated process model is free of these problems. Conceptually, model repair tries to bring the model M closer to the event log L, as sketched in Figure 11.1. Hopefully, after the repair, the resulting model M^* is better fitting, or more precise with respect to L than the original model M.

J. Carmona et al., *Conformance Checking*,
https://doi.org/10.1007/978-3-319-99414-7_11

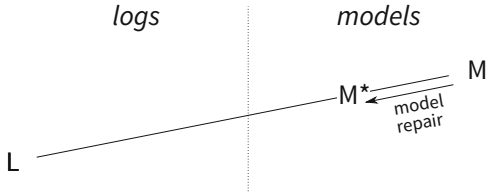


Figure 11.1 Model repair brings the model M closer to the event log L. The repaired model is sketched as M^*

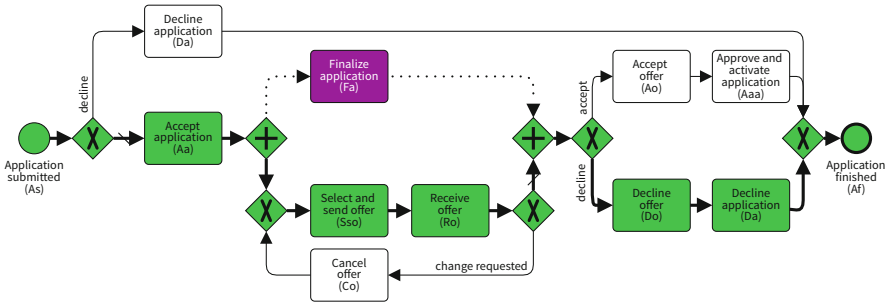


Figure 11.2 Process highlighting one trace with a missing event. More precisely, a deviation corresponding to a trace that does not include an event to signal that the application has been finalised (*Fa*), illustrated already in Figure 3.7 on page 48

Let us illustrate how a repair technique handles a deviation. From the loan application process (shown in Figure 1.3), assume that the activity *Finalise application* (*Fa*) becomes optional. A new possible trace is for example $\langle As, Aa, Sso, Ro, Do, Da, Af \rangle$. Let us call this trace σ_{new}. The deviation of trace σ_{new} is illustrated in Figure 11.2, which pinpoints an unfitting behaviour seen in the event log. We want to repair the running example process model in Figure 1.3 to also reflect cases like σ_{new}. Figure 11.3 shows a possible way of repairing the deviation reported in Figure 11.2 by changing the type of the two parallel gateways to a different gateway type: the *inclusive gateway*.

Excursion 24

The inclusive gateway

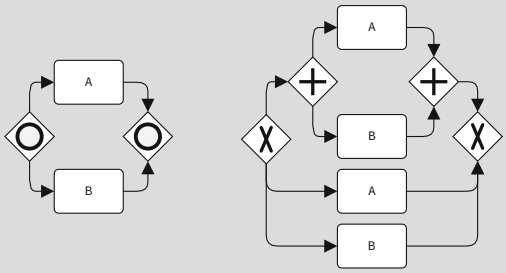
In this book, so far, we have only considered two main gateway types: the exclusive and the parallel gateway. The inclusive gateway, denoted by a circle symbol, can be seen as a combination of an exclusive and a parallel gateway. Like an exclusive gateway, it can be used to define conditions on outgoing sequence flows and the inclusive gateway then evaluates them. However, the

(continued)

main difference is that the inclusive gateway can produce more than one token. The enabled branches are executed in parallel. The functionality of the inclusive gateway is based on the incoming and outgoing arcs:

- *split*: All outgoing arc conditions are evaluated and for the conditions that evaluate to "true", the arcs are assigned an additional token, creating one concurrent branch for each arc.
- *join*: All tokens arriving at the inclusive gateway wait at the gateway until a token has arrived for each of the incoming arcs, on which a token may arrive eventually. This is an important difference from the parallel gateway. In other words, the inclusive gateway only waits for tokens on arcs of branches that are executed. After the join, the process continues past the joining inclusive gateway.

Inclusive gateways can be modelled with exclusive and parallel gateways, duplicating the tasks accordingly. For instance, the following two BPMN fragments are behaviourally equivalent:



That is the main reason why, in this book, we stick to the exclusive and parallel gateways.

In terms of the simplicity metrics reported in Section 10.3.2, repairing the model by changing gateway types has zero cost, i.e., the model shown in Figure 11.3 is as easy to understand as the original. So, this modification is ideal in this regard, i.e., the model obtained is structurally similar to the original model. Moreover, the repaired model accepts the trace σ_{new}, as it allows us to skip the branch with the *Fa* activity. Swapping the parallel gateway for an inclusive one, only added possible behaviour. Therefore, the new model is more fitting to the log than the original model. However, the repaired model in Figure 11.3 is less precise. Some of the new behaviour is not backed up by the event log. For example, the trace $\langle As, Aa, Fa, Ao, Aaa, Af \rangle$, i.e., an application that is approved without an offer, is now an execution sequence of the model.

In contrast, a different strategy to repair the aforementioned deviation is described in Figure 11.4. This repair modification inserts two exclusive gateways and connecting arcs around activity *Finalise application (Fa)*. The repaired model

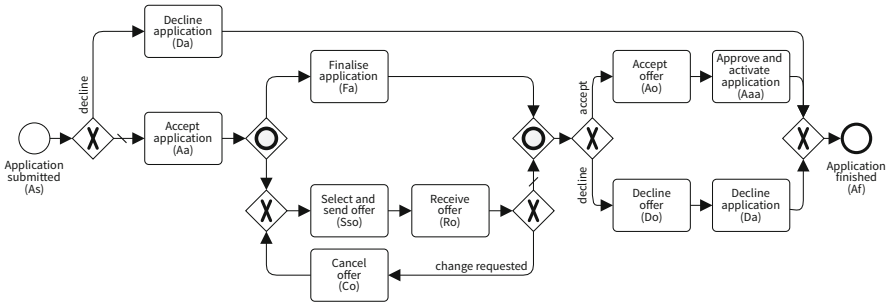


Figure 11.3 Possible repair of the process model for the deviation reported in Figure 3.7

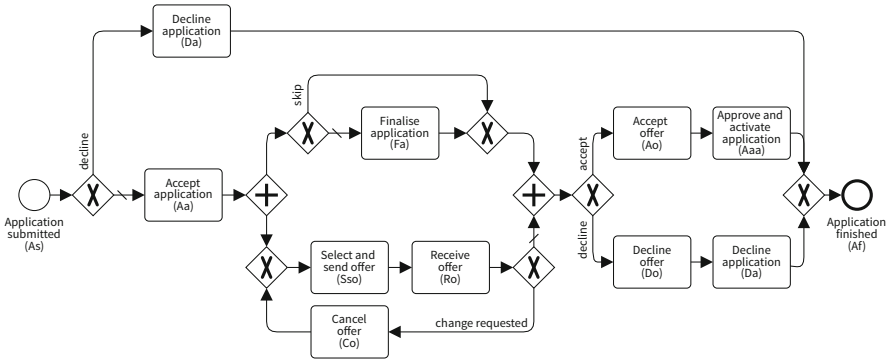


Figure 11.4 Possible repair of the process model for the deviation reported in Figure 3.7

has more elements than the original. Hence, its simplicity is reduced. It is better fitting, by accepting also the trace that underlies Figure 3.7. Finally, the obtained model retains the precision of the original model, since the only new traces accepted are precisely the ones corresponding to the unfitting behaviour diagnosed.

In the previous example, we saw one of the main challenges in model repair: the trade-off between staying close to the initial model on the one hand, and improving conformance metrics like fitness and precision on the other hand. This means the transformation needs to retain the initial process model structure, while amending deviations as much as possible.

We now shortly describe a simple technique for model repair, so that the main idea of using alignments to repair process models is illustrated (see the references at the end of this section for more elaborated approaches). We explain the technique on an example. Let us have a look to the following trace, where the activities *Senior check 1* (*Sc1*) and *Senior check 2* (*Sc2*) are two new activities incorporated in the loan application, that force a senior member to perform two different checks, possibly more than once: $\langle As, Aa, Sso, Ro, Ao, Aaa, Sc1, Sc2, Sc1, Sc2, Af \rangle$.

If we align this trace with the process model of Figure 1.3, the following alignment is obtained:

log trace	As	Aa	$\gg$	Sso	Ro	Ao	$Sc1$	$Sc2$	$Sc1$	$Sc2$	Aaa	Af
execution sequence	As	Aa	Fa	Sso	Ro	Ao	$\gg$	$\gg$	$\gg$	$\gg$	Aaa	Af

There are two ways of repairing deviations: repairs associated to moves in the model (such as $(\gg, Fa)$ in the alignment above), and repairs associated to moves in the log (such as the consecutive moves in log $(Sc1, \gg)$, $(Sc2, \gg)$, $(Sc1, \gg)$, $(Sc2, \gg)$ in the alignment above).

A simple way of repairing a move in the model is by substituting the corresponding task in the process model by an optional execution of it, i.e., a simple exclusive-choice-based structure that allows us to optionally execute the task. The repair was already shown in Figure 11.4.

In general, each model move encountered in the alignment can be repaired as described above, extending the model capability to decide on the execution of those tasks. If we align the trace to the model of Figure 11.4, the following alignment is obtained where the model move $(\gg, Fa)$ has been repaired, thus improving the fitness for the process model:

log trace	As	Aa	Sso	Ro	Ao	$Sc1$	$Sc2$	$Sc1$	$Sc2$	Aaa	Af
execution sequence	As	Aa	Sso	Ro	Ao	$\gg$	$\gg$	$\gg$	$\gg$	Aaa	Af

The strategy to repair log moves is different. In particular, the repair of segments of consecutive log moves, such as $(Sc1, \gg)$, $(Sc2, \gg)$, $(Sc1, \gg)$, $(Sc2, \gg)$ in the example above. First, the state of the process model at the start of this deviation is computed. The state of the model in the previous alignment, at the beginning of these log moves, is reported in Figure 11.5.

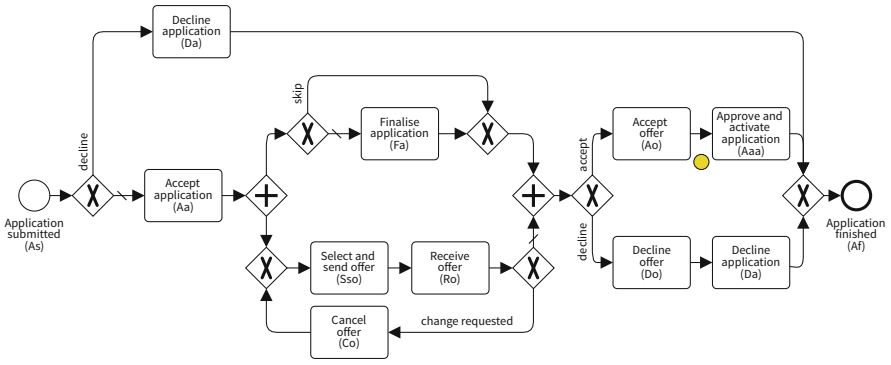


Figure 11.5 State of the process model in Figure 11.4 before the model moves $(Sc1, \gg)$, $(Sc2, \gg)$, $(Sc1, \gg)$, $(Sc2, \gg)$

The behaviour in the state reported in Figure 11.5 is extended to allow for the new behaviour observed, i.e., $\sigma = \langle Sc1, Sc2, Sc1, Sc2 \rangle$, so that the model can either execute it and go back to the same state, or simply move forward in the process. But how we can generate a BPMN fragment that accepts the behaviour in σ? *Process discovery* comes to the rescue!

Excursion 25

Process discovery: Inducing process models from traces

A process discovery technique produces a process model from an event log. Given the traces in an event log, process discovery techniques analyse them to infer the necessary structures which are composed to produce a process model that is capable of reproducing the input traces. There exist a plethora of process discovery techniques, not always producing the same process model. Please refer to [114] for a detailed survey on available techniques for process discovery.

Therefore, through some of the state-of-the-art process discovery techniques, the fragment in Figure 11.6 to represent the subtrace σ is discovered.

Now, this fragment should be inserted in the state highlighted in the process model of Figure 11.5, so that the two checks can be executed several times, or not at all. For that, the process start and end in Figure 11.6 are removed and the fragment is inserted between activities *Ao* and *Aaa*. The final process model is depicted in Figure 11.7.

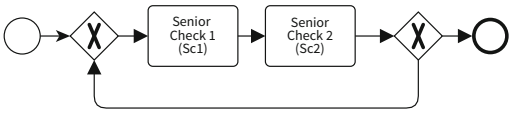


Figure 11.6 Process model discovered for the trace $\sigma = \langle Sc1, Sc2, Sc1, Sc2 \rangle$

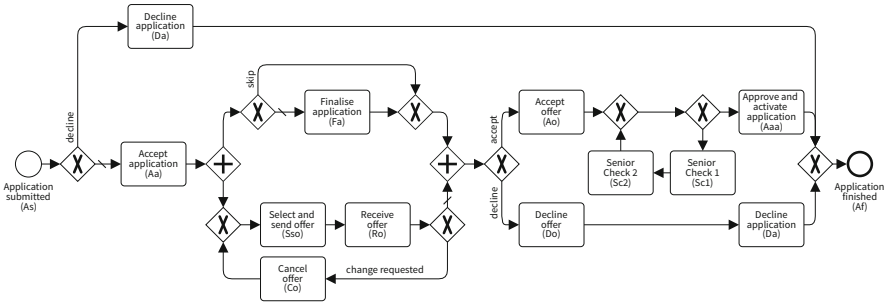


Figure 11.7 Final repair of the process model of Figure 1.3 for the trace $\langle As, Aa, Sso, Ro, Ao, Sc1, Sc2, Sc1, Sc2, Aaa, Af \rangle$

In the final model, the trace $\langle As, Aa, Sso, Ro, Ao, Sc1, Sc2, Sc1, Sc2, Aaa, Af \rangle$ is fitting, as are all previously fitting traces. Hence, the fitness of the model has improved. Moreover, since no new escaping arc has been introduced (see Section 8.5.1), the precision of the model according to the escaping arcs metrics has not changed. However, the complexity of the model increased due to the repairs.

11.2 Log Repair

In the previous section, we assumed that the event log contained the truth and the model was incorrect and hence needed to be repaired to better reflect the event log's contents. However, this is not the only way to look at non-aligned parts between an event log and a process model. Errors in the recording of the log can lead to noisy or incorrect events in an event log. For example, human recording of events is prone to errors. If the time of one event is noted with a mistake, the chronological order of events can be altered in the event log. Also, especially in manually logged processes, the documentation of occurred activities can be forgotten, leading to missing events in a trace.

Conceptually, log repair uses the information and rules stored in the model to correct deviations and violations in the event log. Figure 11.8 shows the conceptual picture of log repair. Errors in the log L are fixed in L^* to make it better fit to the model M.

Looking at the event log from this angle, *log repair* can be used to fix misalignments in the event log. Using alignments as an information source for correcting event logs is straightforward. We can simply insert corresponding events for each model move, and remove events that cause a log move in the alignment (i.e., that do not have a counterpart in the model). For instance, consider again the log trace used in the previous section:

$$\langle As, Aa, Sso, Ro, Ao, Sc1, Sc2, Sc1, Sc2, Aaa, Af \rangle$$

After aligning this trace with the process model of Figure 1.3, the alignment is:
Repairing the trace based on this alignment would result in the following trace:

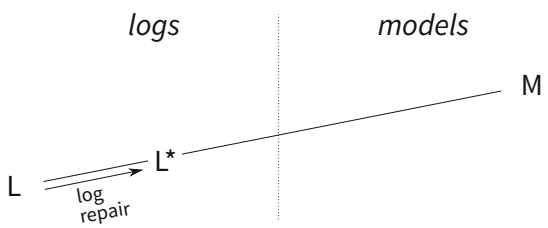


Figure 11.8 Log repair brings the log L closer to the model M. The repaired log is sketched as L^*

log trace	*As*	*Aa*	≫	*Sso*	*Ro*	*Ao*	*Sc1*	*Sc2*	*Sc1*	*Sc2*	*Aaa*	*Af*
execution sequence	*As*	*Aa*	*Fa*	*Sso*	*Ro*	*Ao*	≫	≫	≫	≫	*Aaa*	*Af*

$$\langle As, Aa, Fa, Sso, Ro, Ao, Aaa, Af \rangle$$

Log repair is to be used with caution and, as any other method, needs to be well documented. Ideally, artificially inserted events or removed events can be distinguished in later analysis of the event logs, to compare the repaired log with the original. We need to distinguish between the artificial events (and likewise the removed ones) that we are certain to have happened, from those where we cannot know this. Sometimes, we have domain knowledge that allows us to reason about missing events. One example is a medical case, where events are manually recorded. In this case a process model describes that before the surgery, the patient must be anaesthetised. In such a setting, assume that we see a trace that only contains events recording the surgery and no events pertaining to the patient being anaesthetised.

We know that is it very unlikely that surgery was performed on the patient while they were awake. Therefore, we can deduce that the missing event is not a modelling error, but a recording error, and *induce* the missing "anaesthetize patient" event. These logical dependencies can be encoded in the model as causal dependencies between respective tasks. Furthermore, data is created, read, and written throughout the process. With the same argumentation as with the hospital case, we can deduce that a "fill form" activity must have happened, if a filled form exists in the data of the process at some point. To make use of these logical dependencies, logic programming is one way to fill in the blanks in an event log [24].

11.3 General View on Conformance Checking

In the previous two sections, we discussed model repair and log repair. Each of these problems addresses one particular side of conformance checking. Now, we would like to introduce a more general view that allows us to balance (non-)conformance between models and logs.

When considering whether we need to repair the model or the log, we need to have a mechanism that helps us to decide which part needs to be repaired by how much. The notion of *trust* can be used to specify the degree of assumed correctness at each side. For example, if the model is the result of a quick, first brainstorming session, we would have less trust in its correctness and completeness, as opposed to when the model is the result of a rigorous process analysis project.

Analogously, event logs that are generated by an erroneous process (for example manual documentation of activities, followed by optical character recognition) would be less trustworthy than the event log generated by an model-driven information system orchestrating the execution of the process.

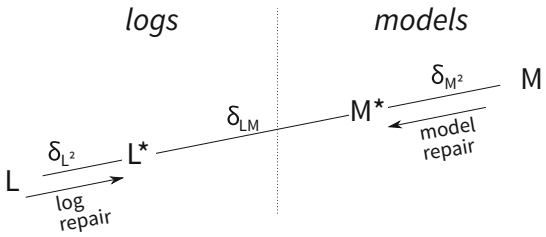


Figure 11.9 A general view on conformance checking needs to balance the repair on both the model M and the log L

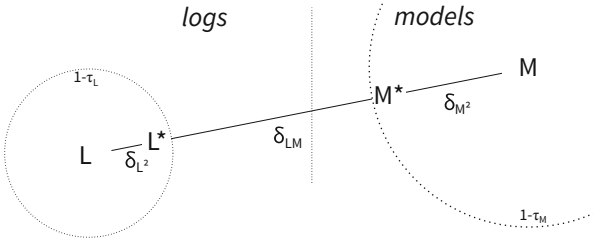


Figure 11.10 General view on conformance checking with trust-based radii to balance the repair on both the model M and the log L

Figure 11.9 shows that in the general view, we repair parts of the model and parts of the log. In the problem formulation, we show one proposal on how it is possible to balance the repairs based on the respective trust levels.

11.3.1 General Problem Formulation

With the introduction of trust level in the model, we can specify how much we would like to repair the model. The more we trust the model, the less we want to repair it. If we fully trust the model, it should not be repaired at all. The same argumentation holds for the trust that we have in the event log.

Therefore, we define the trust levels to be normalised between 0 (no trust whatsoever, can be swapped entirely) and 1 (complete trust, must not be changed). Let us illustrate the effect of trust levels on the allowed changes in Figure 11.10. We see that the trust levels in log and model are not equal. In this sketch, we have a higher trust in the log than in the model. Therefore, the allowed repair radius around log L is smaller than the one around the model M.

The trust radius specifies how much we are allowed to *move away* from the input. To be able to quantify the *repairing*, we need distance notions. We need to compute distance measures among event logs, and among models. Additionally, we

we can define the distance between log and model as some combination of fitness and precision.

Formally, we distinguish between the following distance functions:

- The function $\delta_{L^2} : \mathcal{U}^L \times \mathcal{U}^L \to [0, 1]$ measures the distance between two event logs.
- The function $\delta_{M^2} : \mathcal{U}^M \times \mathcal{U}^M \to [0, 1]$ measures the distance between two models.
- The function $\delta_{L,M} : \mathcal{U}^L \times \mathcal{U}^M \to [0, 1]$ measures the distance between an event log and a process model, for example alignment-based fitness, or the average fitness and precision.

Now we are ready to formulate the generalized conformance checking problem [91].

Problem 1 (Generalized Conformance Checking Problem) Given the input tuple $\langle L, M, \delta_{L^2}, \delta_{M^2}, \delta_{L,M}, \tau_L, \tau_M \rangle$ of the initial log, the initial model, the distance functions and the two trust levels, find a pair $(L^*, M^*) \in \mathcal{U}^L \times \mathcal{U}^M$ such that

$$\begin{aligned}(L^*, M^*) = \quad & \underset{\substack{L' \in \{L'' \in \mathcal{U}^L \mid \delta_{L^2}(L, L'') \leq 1 - \tau_L\} \\ M' \in \{M'' \in \mathcal{U}^M \mid \delta_{M^2}(M, M'') \leq 1 - \tau_M\}}}{\arg\min} f(\delta_{L,M}(L', M'), \delta_{L^2}(L, L'), \delta_{M^2}(M, M'))\end{aligned} \tag{11.1}$$

where f is an abstract function weighing the three distance metrics, for example by taking the weighted average.

In generalized conformance checking, we aim at a log model pair (L^*, M^*) that is in a trust-based proximity to the original log L and model M, as well as at minimal distance from each other. The former is reflected by bounding L' and M' to logs and models that are inside the trust-based radius and the latter is reflected by minimizing the mutual distance (arg min returns the argument that minimizes an expression).

11.3.2 Different Process Mining Techniques as Specific Instances

With this problem formulation, conformance checking can be seen as a search problem, where we try to find the optimal solution to minimize the distances. However, unlike the alignment problem, the generalized conformance checking problem cannot be solved with the $A^\star$ algorithm. The reason is that the goal state is not known. We have a more general optimization problem at hand, which in its generality also includes process mining.

Table 11.1 shows the various disciplines known in the process mining field that can be seen as instances of the generalized conformance checking problem, Problem 1. This table emphasizes that the problem itself is interesting to explore,

Table 11.1 Some process mining tasks can be seen as instances of the generalized conformance checking problem. The respective trust levels in log and model are given on the right; cf. [91]

Process mining task	Log Trust	Model Trust
Classical Process Discovery finds a model that best fits to the entire event log, e.g., the alpha algorithm [119].	$\tau_L = 1$	$\tau_M = 0$
Heuristic Process Discovery algorithms apply preprocessing to the event log by discarding infrequent patterns [52, 134].	$0 < \tau_L < 1$	$\tau_M = 0$
Model Repair fixes deficient models due to, e.g., a change in the system that is reflected in the log. For example [41].	$\tau_L = 1$	$0 < \tau_M < 1$
Conformance Checking tries to find misalignments between event log and model. Example works include [97, 101, 115].	$\tau_L = 1$	$\tau_M = 1$
Log Repair modifies the log such that it better conforms to the given trusted model [90, 128].	$0 < \tau_L < 1$	$\tau_M = 1$
"Happy Path" Simulation is complementary to heuristic process discovery. It is a theoretical use case where we do not trust infrequent parts of the model [68].	$\tau_L = 0$	$0 < \tau_M < 1$
Process Simulation is complementary to process discovery, where we are given an untrustworthy empty log and a fully trustworthy model.	$\tau_L = 0$	$\tau_M = 1$
Garbage In, Garbage Out. When both the model and the log are untrustworthy, the best log and model tuple that fits them is any pair of model and log that fits each other, including an empty log and an empty model.	$\tau_L = 0$	$\tau_M = 0$
Generalized Conformance Checking answers the question where the model would best be adopted, and where the log would best be adopted for a better overall fit. This goes beyond alignments, as the latter only detect the misalignments without specifying which side is to "blame".	$0 < \tau_L < 1$	$0 < \tau_M < 1$

and some first attempts at solving it in its generality have been made in [91]. However, conformance checking is still a novel and active area of research and more advanced solutions can be expected soon.

In fact, some of the research results we presented in this book are already sophisticated enough to find their way into software products. We shall explore conformance checking software in the next chapter.

Bibliographical Notes

There are a few available techniques for repairing process models [10, 41, 81]. In Section 11.1 we shortly described part of the technique in [41], which is available in ProM through the package *Uma*. Techniques related to model repair that are more focused on the simplification of the model without degrading other quality metrics can be found in different publications [34, 40, 83].

Log repair can be simple and frequency-based with the assumption that whatever behaviour is rare, is also wrong behaviour that can be filtered out [25]. Many process

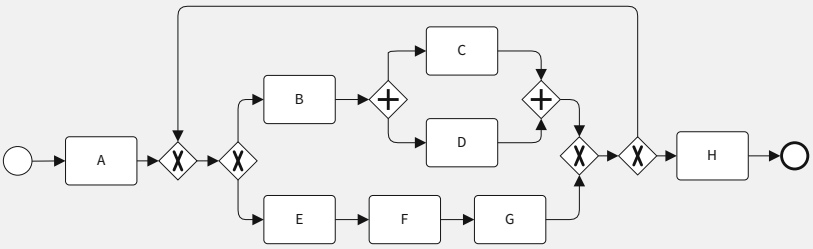
mining algorithms use this assumption to restrict the variability and focus on the most frequent behaviour [134]. The log "repair" in this case is implicitly done before mining the models. More explicit log repair approaches using logical rules to restore missing events can be found in [24, 43]. Bayesian statistics to restore missing events with their most likely timestamps are applied in [90]. A catalogue of patterns that indicate problems with event logs and corresponding solutions is given in [105].

The general view on conformance checking first appeared in [91], but the authors are continuing the research in this area.

11.4 Exercises

11.A) Model repair

Given the following process model:



Repair it according to the trace $\langle A, E, G, B, X, Y, D, H \rangle$. Remember that for repairing a model, you first need to align the trace with the model and then treat each one of the types of asynchronous moves individually as explained in Section 11.1.

11.B) Log repair

Given the process model and trace shown in the previous exercise, now perform the opposite operation: Repair the trace according to the process model.

11) In the lab: tool support for improving processes

Check out the lab session to practise with different tools implementing the techniques for improving processes introduced in this chapter.

http://www.conformancechecking.com/CC_book_Chapter_11

Chapter 12
Conformance Checking Software

The previous chapters of this part of the book introduced techniques that exploit alignments between recorded and modelled behaviour to understand or even improve the conduct of a process. Yet, our focus has been on the general concepts in terms of models, measures, and algorithms. The application of these techniques in practice, however, requires the implementation of these concepts in software solutions.

Against this background, this chapter turns to conformance checking software. Specifically, our goal is twofold. First, in Section 12.1 we show how conformance checking techniques can be implemented by reviewing the ProM Framework, a widely established open-source framework. It covers most of the techniques discussed in this book, making them easily accessible and enabling researchers to create prototypes for testing novel approaches to conformance checking. Clearly, the ProM Framework is not aimed at end users, who would simply like to apply conformance checking in an organization. A second goal of this chapter is, therefore, a discussion of commercial offerings for conformance checking, provided in Section 12.2. These offerings represent a highly dynamic market with new solutions being released every few months. Hence, instead of reviewing specific solutions, this chapter details some general considerations, which include requirements in terms of functionality beyond the core algorithms as well as non-functional aspects. This way, we strive for a discussion that supports users in the assessment and comparison of commercial offerings for conformance checking, in the light of their specific application scenario.

12.1 The ProM Framework

The ProM Framework is an open-source framework for Process Mining (Figure 12.1). The first version of ProM was released in 2004. The initial goal of ProM was to unify process mining efforts at Eindhoven University of Technology

J. Carmona et al., *Conformance Checking*,
https://doi.org/10.1007/978-3-319-99414-7_12

Figure 12.1 ProM splash screen

and other cooperating groups. The framework allows researchers from all over the world to contribute their work and expose their ideas to a large community. Over the years, ProM has grown to be the world-leading proof-of-concept for all kinds of Process Mining technology, ranging from process discovery to social network analysis, and from event visualization to conformance checking.

In this section, we focus on the conformance checking abilities of ProM. We show, tutorial style, which steps need to be taken to do conformance checking on a process model and an event log. As an example, we use a real-life event log published by the 4TU Centre for Research Data in 2012: http://data.4tu.nl/repository/uuid:3926db30-f712-4394-aebc-75976070e91f

This event log is a log of a loan application process in a global financial institute. It contains 13,087 traces with 262,200 events. Apart from some anonymization, the log contains all data as it came from the financial institute. The process represented in the event log is an application process for a personal loan or overdraft within a global financing organization. The amount requested by the customer is indicated in the case attribute AMOUNT_REQ, which is global, i.e., every case contains this attribute. The event log is a merger of three intertwined sub-processes, namely the (A)pplication process, the (O)ffer process, and the (W)orkflow. The first letter of each activity name identifies from which sub-process (source) it originated.

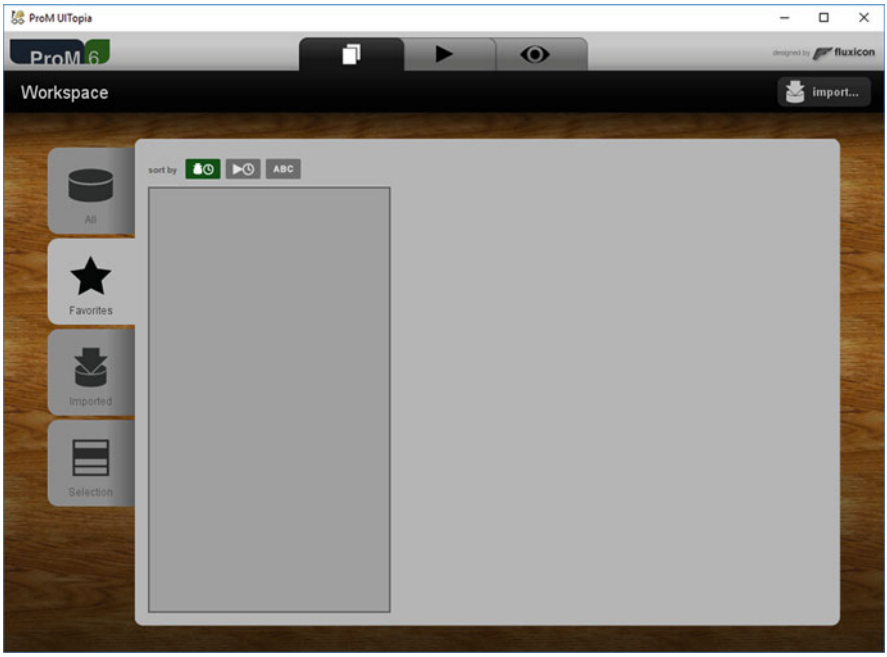


Figure 12.2 ProM workspace

The version of ProM we use for this section is ProM Lite.[1] ProM Lite is a release for end users. It only contains the most common packages, shows the most common plug-ins, and it allows a user to select whether to install new packages and/or updates when starting. We refer to http://www.promtools.org/ for more information on ProM, ProM Lite, and online courses on how to use ProM in common scenarios.

We start by opening ProM, which shows us the workspace as shown in Figure 12.2. In the top-right corner is a button to import files into the workspace. We import both the event log file and the Petri net (.pnml) file. Note that opening the log file results in ProM asking for the appropriate import plugin. The default will do here.

After import, ProM shows the two objects opened as depicted in Figure 12.3.

The first step is now to view the contents of the log by first selecting the log and then clicking the eye icon to the right of the log. This shows the log visualization depicted in Figure 12.4.

This logview allows the user to inspect the log and its contents. On the first page, an overview is shown indicating the number of events (262,200) and cases (13,087). The inspector and summary tabs can be used to inspect individual events and cases. In Figure 12.5, one case is shown (case number 173688) which was a

[1]The figures in this chapter are made using ProM Lite version 1.2.

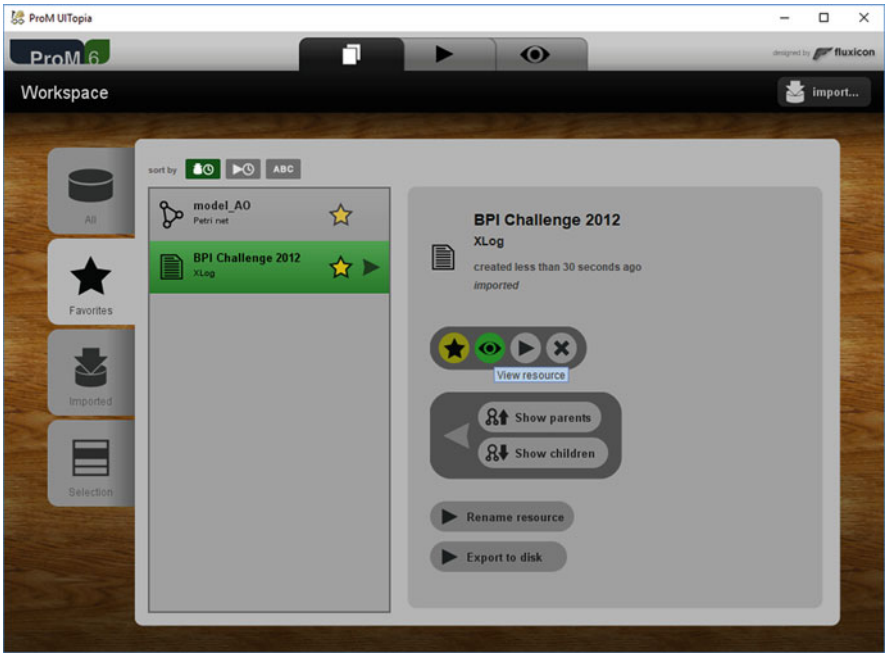


Figure 12.3 ProM workspace showing imported log and model

Figure 12.4 ProM visualization of the log

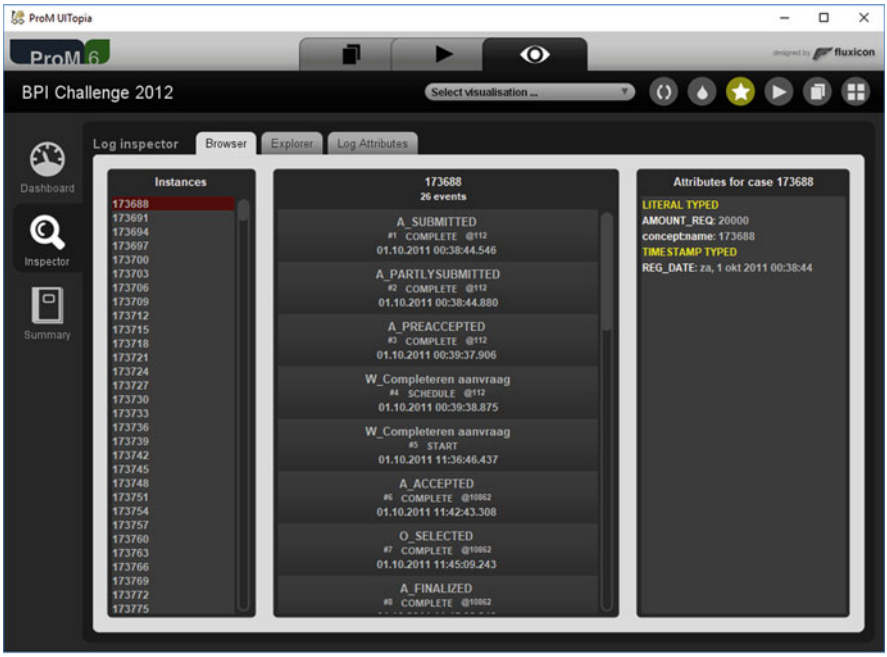


Figure 12.5 ProM visualization of one case

loan for 20,000 Euro (AMOUNT_REQ). Furthermore, some events are shown with names and timestamps.

Going back to the workspace, we can also visualize the model by clicking the eye icon. The result is depicted in Figure 12.6. Please note that the layout information for this model is stored in the PNML file. However, if no layout information is present, ProM does an automatic layout of the model.

The model depicted contains only transitions relating to (A)pplications and (O)ffers. Essentially, the middle part of the model is the logical flow of the process. The top part deals with cancellation and the bottom part with rejection. Applications go through a number of steps and, if applicable, offers are sent to the customer in a loop. Offers are sent one by one, until the customer cancels the application, rejects the offer, or accepts the offer. In the latter case, the process continues with a load activation, i.e., money is transferred to the customer.

In order to investigate the conformance between the log and the model, we start by filtering the log for events pertaining to applications and offers. We do so by going back to the workspace. Then, we select the log and click the play icon. This opens the Actions view and here we search for the plugin called “Filter Log using Simple Heuristics” as depicted in Figure 12.7.

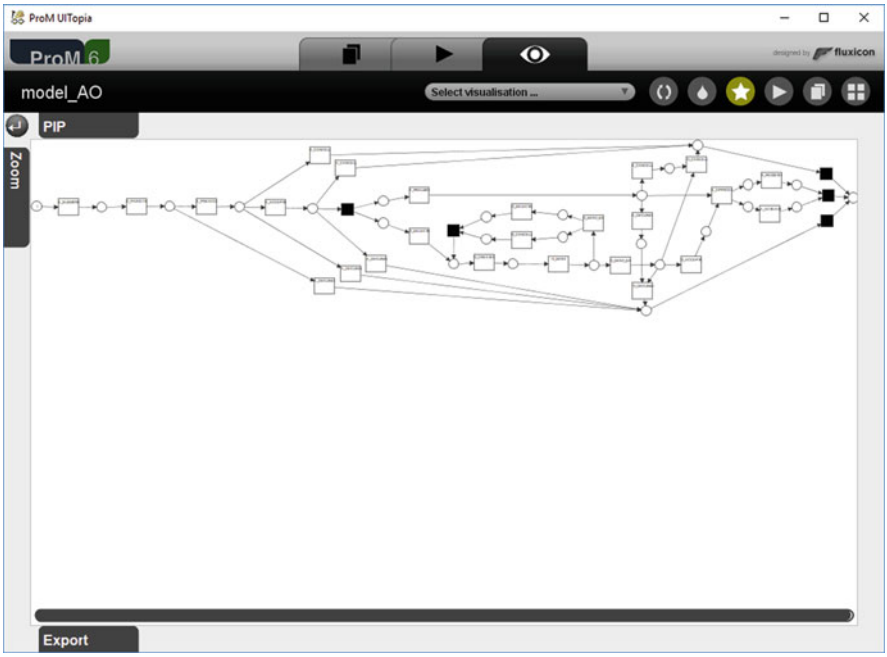


Figure 12.6 ProM visualization of the Petri net

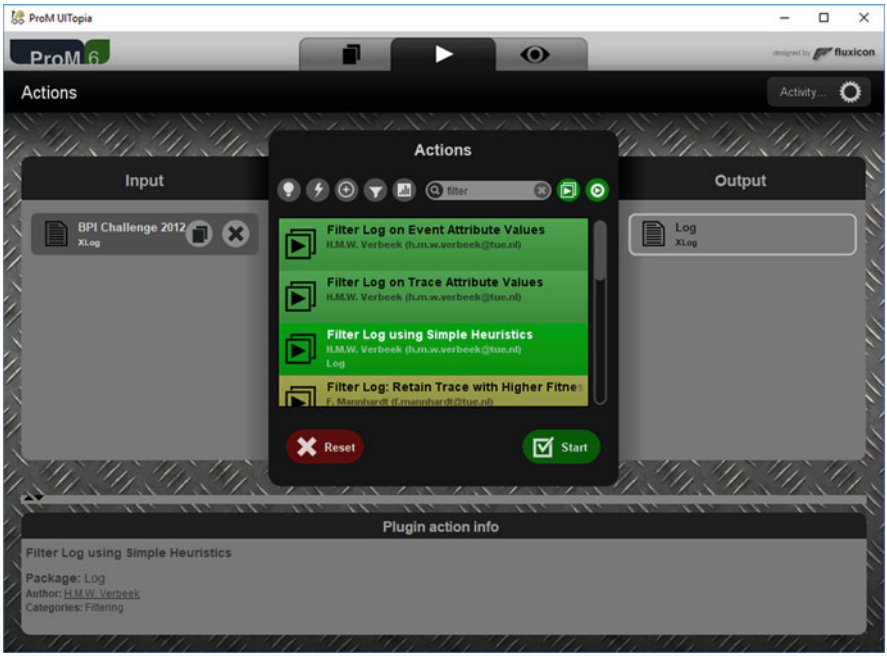


Figure 12.7 Actions view with filter plugin selected

Figure 12.8 Filtering steps in the "Filter Log using Simple Heuristics" ProM-plugin. (**a**) Filtering for event types. (**b**) Filtering for start events. (**c**) Filtering for end events. (**d**) Filtering for events starting with "A" and "O"

When clicking "start", this filtering plugin takes the user through a number of steps during which we select:

1. To filter out all events with "schedule" and "start" in the name (Figure 12.8a),
2. To keep all cases starting with "A_submitted" (Figure 12.8b),
3. To keep all cases ending with any of the end events (Figure 12.8c),
4. To keep only events starting with "A" or "O" (Figure 12.8d).

After filtering, we have a new event log with the name "BPI Challenge 2012 (filtered for AO)" with 13,087 cases and 92,093 events (Figure 12.9).

By clicking the play icon again on the top right, we go back to the actions view, where we search for the plugin "Replay a Log on Petri Net for Conformance Analysis". On the input side, we select both the filtered event log and the model we imported earlier and we can click start again as depicted in Figure 12.10.

The conformance checking plugin now takes the user through a number of configuration steps, which are necessary to compute alignments. First, the user is asked to provide a final marking for the Petri net, i.e. what is the desired final state for the model? Most Petri nets do not come with a pre-defined final marking and

Figure 12.9 Result of first filtering: 13,087 cases and 92,093 events remained

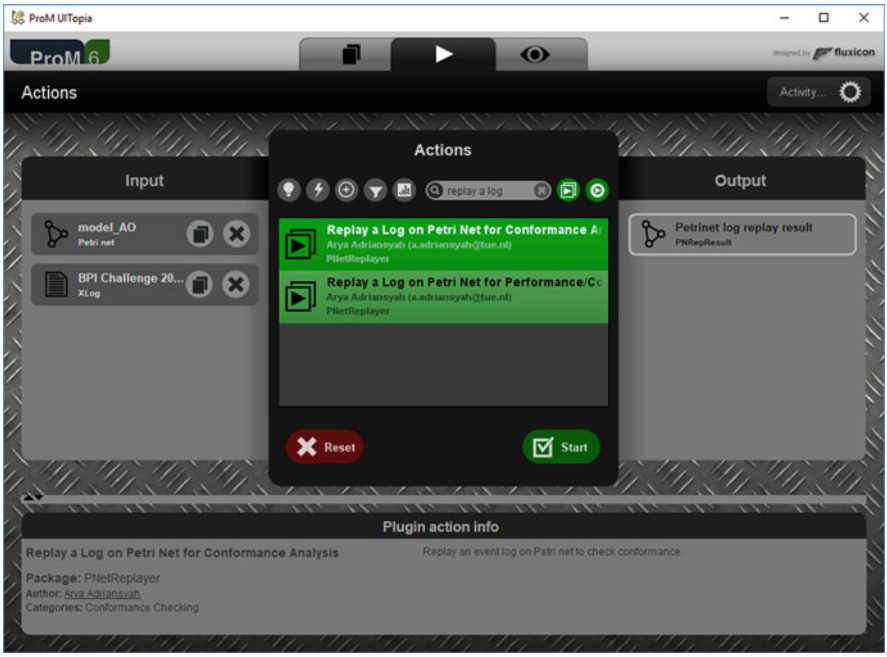


Figure 12.10 Actions view for conformance checking

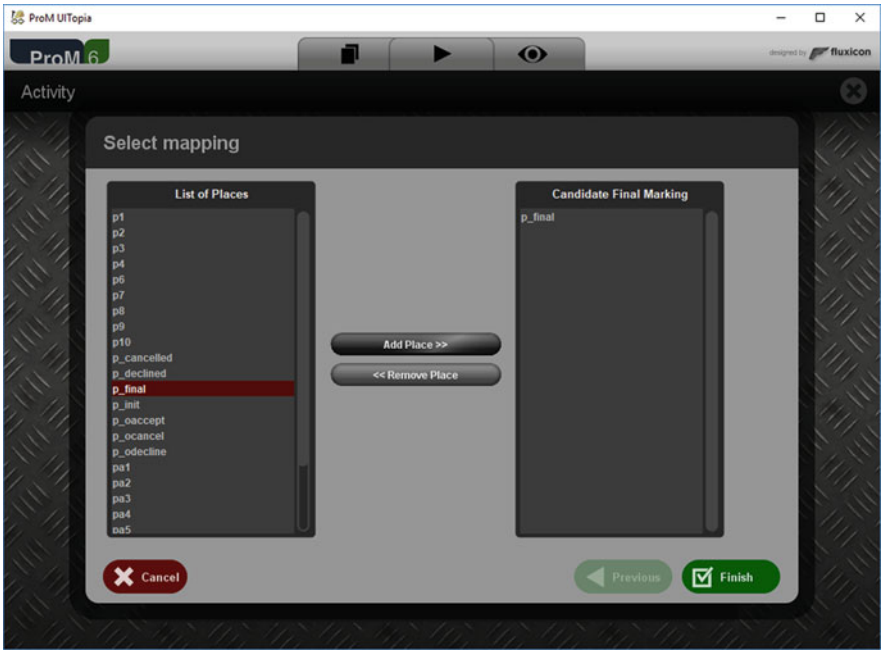


Figure 12.11 Wizard for creating a final marking in the Petri net

often, the file formats do not even support storing it. Hence, this question is asked for each Petri net (but only once per ProM session). In our case, we specify the final marking to have a single token in the place "p_final" as depicted in Figure 12.11.

The next step is specification of the relation between the events in the log and the transitions in the model. Each transition of the model needs to be mapped to zero or one activity recorded in the log. In ProM, activities are called "event classes" and they are derived from a log based on so-called classifiers. In our example, the classifier "Event Name" should be selected and then the appropriate mapping is suggested automatically as depicted in Figure 12.12. Please note that the yellow entries show that the names do not exactly match. Furthermore, on the bottom of the list, there are some transitions that are not mapped to any activity. Some of these transitions are the τ-labelled transitions we discussed in the book, others may be transitions referring to activities which are not recorded in the log. Like the final marking, this mapping is also stored for the entire ProM session.

After setting up the relation between log and model, we need to select a Conformance Checking algorithm as well as the parameters for that specific algorithm. In this section, we select "A^* Cost-based Fitness Express with ILP" as an algorithm and we leave the parameters as default as depicted in Figure 12.13.

The next step is the cost function associating costs to deviations. The wizard to set this cost function is shown in Figure 12.14. By default, move on model costs for transitions that are mapped to activities is set to 1 and for transitions

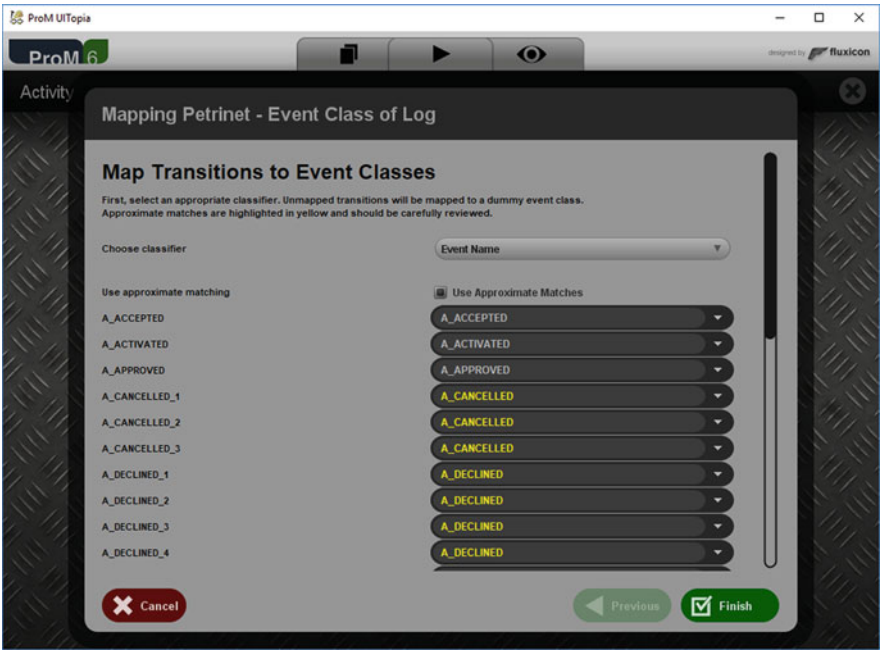


Figure 12.12 Wizard for mapping events to transitions in the Petri net

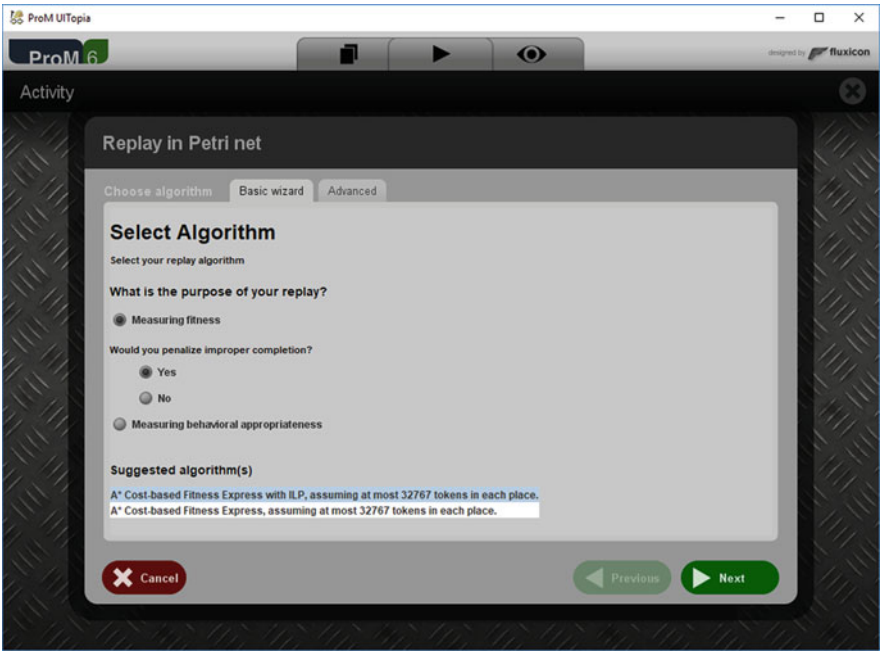


Figure 12.13 Replay algorithm selection

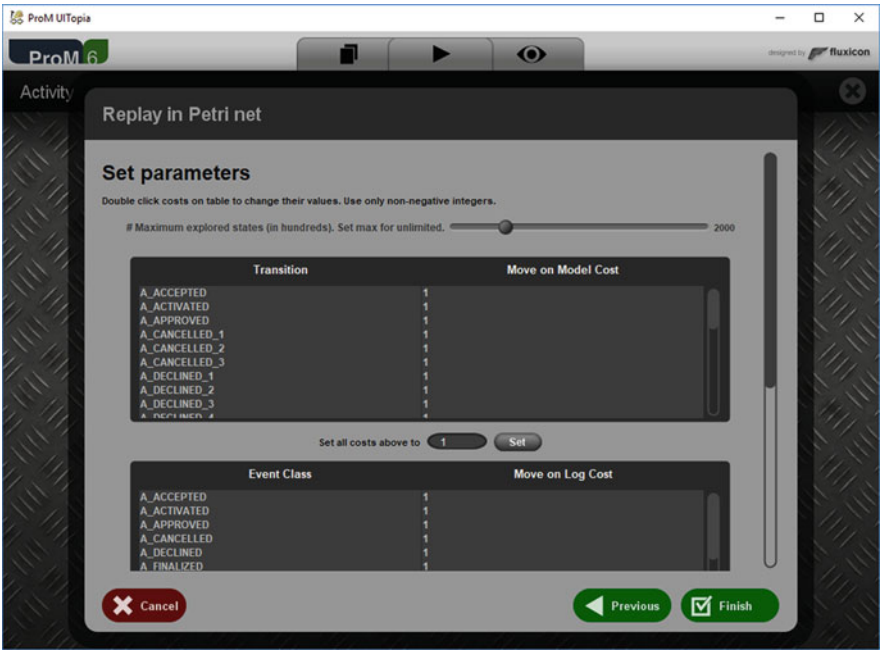


Figure 12.14 Replay cost function selection

that are not mapped to transitions to 0. Also move on log costs are by default set to 1 for all activities and finally the costs of synchronous moves are set to 0. On the bottom of this dialog, there is also an option to improve the performance of the alignment algorithm for event logs where multiple events within a case have the same timestamp. Checking this option may improve the result, but may also decrease performance.

Finally, after clicking "Finish", the conformance checker starts aligning each trace to the process model. This takes a few seconds for this example, but can take considerably longer. During the computations, progress is reported as shown in Figure 12.15.

After completing the alignment computations, the result is (by default) projected on the model and a visualization is shown, as depicted in Figure 12.16.

On the top right, another visualization can be chosen for the conformance result, namely "Project Alignment to Log", which shows a different visualization which we discussed earlier in the book, i.e., the visualization shown in Figure 12.17 with explicit synchronous moves, log moves, and model moves. Furthermore, in this view, we can also obtain the fitness values from the menu on the right-hand side, which for this event log and model combination are as follows: The alignment-based fitness (trace fitness in the tool) is 0.9476. The move model fitness is 0.9808 and the move log fitness is 0.9580. Overall, this model and log fit very well together.

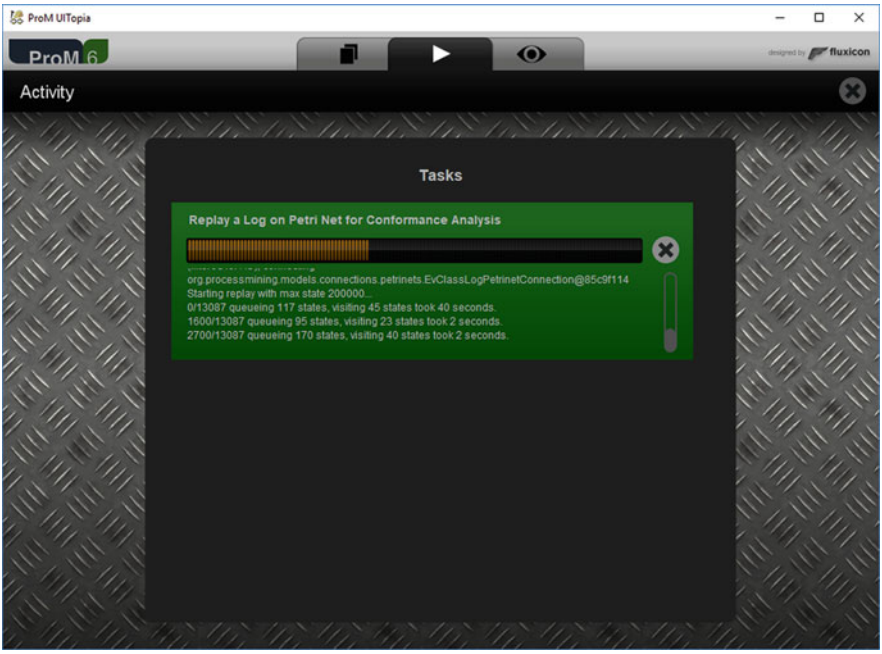


Figure 12.15 Replaying might take time, progress is reported

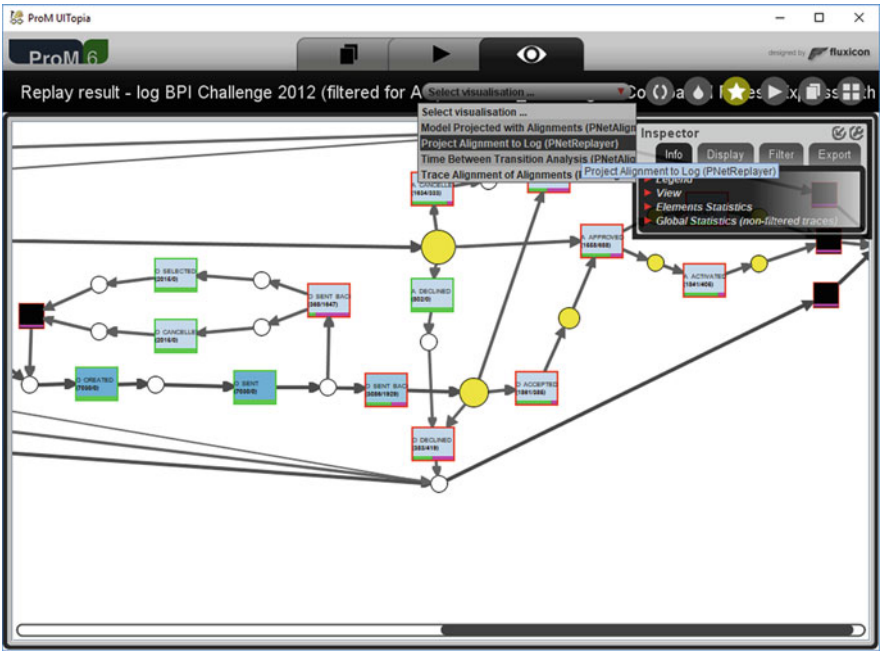


Figure 12.16 Default visualization of a replay result

Figure 12.17 Log-based visualization of a replay result

Using the play button on the top right corner of Figure 12.17, the alignments that have been computed can be used to compute precision and generalization. Figure 12.18 shows the necessary input for the precision plugin and, using the default settings, the precision for this model (using the technique outlined in Section 8.5.1) on the filtered log is reported to be 0.62559.

The small tutorial here only gives a brief introduction to conformance checking in ProM. Using the inspector in the model projection, users can filter traces with specific deviations, export them to a separate log, and inspect the respective cases. In this particular example, there are three problematic cases, namely 177083, 180310 and 198310. In the three respective traces, the activity "O_Accepted" never happened, while the activities "A_Accepted" and "A_Activated" did. This implies that in these cases, the customer never accepted an offer, but still the financial institute transferred money to their bank accounts, in total: 63,000 Euro.

Furthermore, as is already visible in Figure 12.16, the activity "O_sent_back" is often not seen in the log. This turned out to be related to the fact that call agents did not actually send offers one by one, but they would send multiple offers in one envelope. Only one of the offers would often be sent back by the customer; hence, for the others, the corresponding events would then be missing.

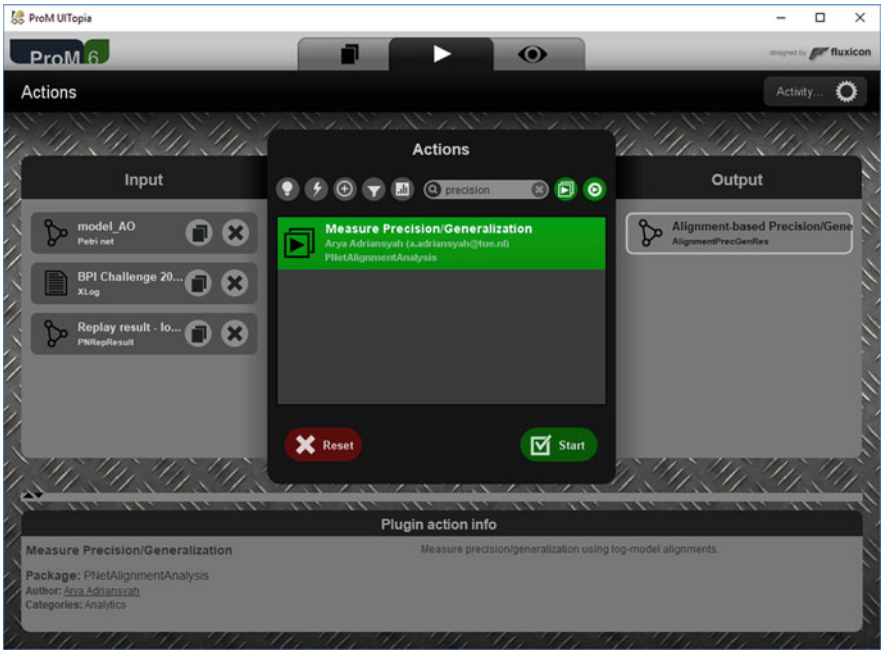


Figure 12.18 Action view with precision/generalization plugin selected

12.1.1 Beyond ProM-Lite

Many of the techniques presented in this book have been implemented in ProM packages. These packages are however not always available in ProM Lite. In this section, we give an overview of the relevant ProM packages and we refer to the documentation of ProM on how to install these packages and how to use the plugins in there.

AntiAlignment This package contains the implementations in ProM for computing anti-alignments as presented in Section 8.5.2.

DataAwareReplayer Is a package covering the data-aware techniques reported in Section 9.1.

DecomposedReplayer Is a package with the plugins for decomposed replaying as presented in Section 9.3.

StreamConformance Contains the work on online conformance checking presented in Section 9.2.

Mixed Paradigm This package allows the alignment plugin presented in this chapter to work on mixed-paradigm models, i.e. models with declarative constraints between transitions as presented in Section 9.5.2.

Uma This package contains several model repair techniques, including those ones presented in Section 11.1.

ProM, not ProM Lite, but the larger framework, contains many more plugins in various stages of maturity and it is beyond the scope of this book to present them all. However, we encourage the reader to try ProM.

12.2 Software for Conformance Checking in Industry

In recent years, several commercial offerings for process mining have entered the market. Some of these software solutions offer conformance checking capabilities. Since the market of process mining solutions is changing rapidly, we refrain from discussing these solutions in detail in this section. Rather, we elaborate on some general considerations in terms of functional and non-functional requirements that provide a basis for the assessment of conformance checking solutions in practice.

Our goal is to outline the spectrum of aspects that may be considered when evaluating software for conformance checking. Yet, this discussion can neither be exhaustive, nor argue for the importance of individual aspects for particular application domains. Rather, the discussion shall be understood as some general guidance that may be useful in the assessment of a software solution with respect to specific analysis questions.

12.2.1 The Functional Perspective

From a functional point of view, first and foremost, software solutions for conformance checking differ in the actual algorithms that detect deviations between modelled and recorded behaviour. However, as detailed below, practical support for conformance checking may include functionality beyond the actual conformance checking algorithms.

Conformance Checking Algorithms Software solutions may adopt different formalisms for conformance checking. In Section 4, we reviewed three such formalisms, in terms of rule checking, token replay, and alignments. Also, we argued that these three methods are complementary, as they strive for a different balance, for instance, between the completeness of conformance checking and its computational efficiency. As such, tools that are based on either type of method may be more suitable for a particular use case, than another software solution that adopts a different formalism.

Independently of the general approach taken for conformance checking, software solutions shall also be assessed in the details of the specific realization. In rule checking, the set of supported rule definitions has far-reaching consequences; see Section 4.1. In token replay, as discussed in Section 4.2, algorithmic choices on how to handle non-determinism in a process model influence the obtained conformance checking results. Similarly, when adopting the paradigm of alignments

for conformance checking, tools may differ in how they compute alignments. From a qualitative point of view, differences may relate to the question of whether the computation is guaranteed to be optimal and deterministic; see Chapter 7.

Event Log Preparation In many application scenarios, conformance checking is preceded by some data preparation phase. Starting with raw data as extracted from information systems, event logs in the structure required by conformance checking techniques need to be derived. Beyond basic data normalisation and filtering, this includes functionality to relate events and activities, and events and traces; see Chapter 6.

Since event log preparation may require significant manual effort, it often represents a major obstacle to conducting conformance checking in practice. Tool support for the event log preparation, therefore, is of crucial importance. Potentially, such support includes adapters for data extraction from legacy systems, methods for data profiling to explore the structure of extracted data, mechanism for filtering and outlier detection to handle noise and define the scope of the analysis, automatic drift detection and data abstraction, learning methods to correlate events into traces, or (semi-)automated matching to link events with activities.

Result Abstraction: Drill-Down and Roll-Up In order to make effective use of conformance checking, the obtained link established between modelled and recorded behaviour needs to be interpreted. For instance, in Section 8, we discussed how alignments can be visualized and provide the basis for manifold measures to assess the conformance of an event log and a process model.

Tools differ widely in how they support such interpretation of conformance checking results. Even when relying on the same formalism and adopting the same visualization, important differences may relate to the functionality offered in terms of result abstraction. Instead of providing solely low-level feedback on the detected deviations per trace of an event log, tools may derive more abstract representations. Abstractions here potentially relate to the notion of the reported deviations. For instance, when considering alignments, feedback may not be limited to listing the moves in log and moves in model per trace. Rather, it can be given in terms of high-level behavioural patterns, e.g., on the level of sub-processes. Similarly, abstractions may be grounded in the scope of the considered events. Deviations can be reported per trace, group of similar traces, or the log as a whole.

To support the interpretation of conformance checking results, an implementation of the above abstractions shall be accompanied with support for navigating through these abstractions. Software solutions with drill-down and roll-up functionality that enable a user to derive more or less abstract feedback on deviations are valuable means to make sense of conformance checking results.

Reference Models Conformance checking solutions can support process analysis based on predefined reference models. That is, behaviour as recorded by information systems as part of process execution is compared against models that are shipped with the conformance checking tool. For application domains such as order processing in the automotive industry or IT incident management, reference

models are widely established, sometimes even promoted by standardization bodies. Commercial offerings for conformance checking that are tailored for these domains may provide functionality to customize the respective reference models, thereby largely releasing the end user from the burden to explicitly provide process models for the analysis.

Integration with Process Discovery Conformance checking may be conducted in application scenarios, in which a specification of a system's behaviour is not directly available. In that case, software solutions may also offer techniques for process discovery that construct a process model from the event log. Common techniques for process discovery filter the information in an event log, excluding rare behaviour in particular. At the same time, discovery often generalizes, i.e., creates a process model that allows for more behaviour than what has been recorded in the log. Consequently, even when a discovered model is used as the input for conformance checking, one typically finds deviations between this model and the original event log.

Software solutions differ in their support for automated process discovery, e.g., in terms of the applied representational bias and the implemented discovery algorithms. In addition, some tools include actual process modelling functionality, which allows for manual adaptation of the model derived by automated discovery.

Comparative Analysis Conformance checking typically is not only a one-off operation, but relates to the evolution of the consistency between modelled and recorded behaviour over time. One could argue that, in order to understand this evolution, it is sufficient to compare the conformance checking results at different points in time, or for models and event logs capturing different time periods, respectively. Yet, software solutions to conformance checking may offer more elaborate support for such comparative analysis. Exemplary functionality in this regard includes an explicit detection of changes in the deviations between a model and a log, thereby highlighting, e.g., the outcome of process improvement or redesign initiatives.

In addition to functionality to analyse the evolution of process conformance for user-defined time periods, tool support may also cover change point detection. That is, the point in time at which significant changes in a process' conformance materialize are identified automatically.

Impact Assessment In practice, deviations between modelled and recorded behaviour of a system shall be assessed in terms of their impact. Clearly, some deviations are more important than others. Considering the process of processing a loan application, an activity related to informing the applicant on the progress of processing is less crucial in terms of business impact than the activities that refer to the acceptance of a request. Skipping the former has implications in terms of client inconvenience, whereas deviations regarding the latter activity may have severe legal and financial consequences.

While the assessment of the impact of deviations is domain-specific, software solutions typically offer means to configure the importance of particular activities.

Also, the impact of deviations can be evaluated against the importance of cases as a whole. For instance, in the above example, the amount of the requested loan provides for a simple indicator of the importance of individual cases. Based thereon, the impact of a specific deviation in terms of the importance of the affected cases can be determined. By providing means to compute such impact metrics, software solutions enable users to prioritize deviations in their analysis of conformance checking results.

Root Cause Analysis The actual computation of conformance checking results is typically succeeded by an in-depth analysis that aims at revealing the origins of deviations. Such root cause analysis strives for a characterization of the context in which particular process conformance issues materialize. Here, the context can be defined along various dimensions. A conformance issue may be limited to a time-of-day or day-of-week of process execution, involve specific roles and resources, or be attributable to data assigned to individual cases. Understanding such contexts is valuable when aiming at an understanding of the factors that contribute to non-conformance.

Software solutions to conformance checking can support such root cause analysis by implementing basic machine learning algorithms. For instance, a characterization of the context of a deviation may be identified by solving a classification problem. Using contemporary learning algorithms, a classifier is learned that separates traces for which the deviation has been observed, from those that do not show the respective conformance issue.

Operational Support Most of the techniques covered in this book target a posteriori analysis, i.e., conformance is established after the fact, based on event data recorded in the past. Yet, as outlined in Section 9.2, conformance checking may also be conducted as an online procedure that assesses the conformance of a running case based on a stream of events. Since such online processing requires different algorithms than offline analysis and imposes new challenges in terms of interpreting the result, tools for a posteriori conformance checking cannot directly be employed. Rather, software solutions need to provide dedicated support for online analysis.

Software solutions may further offer functionality to move from reactive conformance checking to predictive analytics. For a running case, forecast models may be able to anticipate the occurrence of a conformance issue in the near future, e.g., based on behavioural patterns as well as the data assigned to cases. The functionality provided to learn such forecast models and evaluate them in an online manner, therefore, is a feature that may turn out to be important when comparing different software solutions.

12.2.2 The Non-functional Perspective

The functional requirements discussed above are not sufficient to assess the suitability of some conformance checking software for a particular application

scenario. They shall be complemented by an investigation of additional aspects, often summarized under the umbrella of non-functional requirements. Below, several of these aspects are reviewed. Again, this list cannot be exhaustive, and may refer to aspects that are not important under all circumstances. The judgement ultimately needs to be taken in the light of a specific application scenario.

Scalability Depending on the size of the event logs to consider, scalability of conformance checking may be important. Here, scalability relates to the sheer volume of data that can be handled by a tool in the first place, as well as optimizations that reduce the encountered run-time. If conformance checking is based on alignments, common algorithms show an exponential time complexity (see Section 7.6), and thus may become a bottleneck in the analysis.

Against this background, conformance checking software may implement decomposition schemes in order to speed up the computation. As discussed in Section 9.3, this may involve vertical decomposition of entire traces of an event log, and horizontal decomposition that slices traces by event groups. For instance, vertical decomposition provides a simple way to parallelize the computation of alignments, potentially using distributed infrastructures for data-parallel processing.

Security and Privacy Since conformance checking may refer to sensitive data, respective software shall provide the appropriate security and privacy mechanism. This is not limited to securing the process models and event logs used as input for the analysis, which may denote valuable business assets. Security and privacy mechanism shall also extend to the actual results of the analysis, i.e., information on conformance issues. In particular, monitoring of work forces can only be conducted within the bounds set by the respective legal frameworks. Conformance checking may therefore provide mechanisms for data anonymization or aggregation, in order to ensure that conformance issues cannot be attributed to individual resources or workers, if needed.

Version Management As argued above, conformance checking is rarely a one-off operation. In addition to providing support for comparative analysis, software solutions shall enable management of the meta-information related to analysis setups. To this end, version management of the artefacts used as input, event logs and process models, as well as the analysis output are crucial to achieve traceability of conformance checking.

Usability To make conformance checking easily accessible, usability aspects need to be considered when assessing and comparing software solutions. The respective offerings shall tailor their interaction concepts and interfaces to the background of particular user types. Beyond the adopted visualizations (see also Section 8.2), an intuitive graphical user interface, the supported languages, built-in help functions, and search mechanisms contribute to high usefulness of a tool.

Documentation Conformance checking solutions may differ in the availability of accompanying documentation. Such documentation shall be tailored to specific types of stakeholders and shall potentially span various perspectives, from system

setup and customization, through administration and maintenance, to guidance for end users.

Deployment and License Models Commercial offerings for conformance checking differ in the adopted and deployment and license models. For instance, some solutions are available as stand-alone desktop applications, whereas others rely on server-based architectures that may be deployed in-house or used in a software-as-a-service model. Such differences are typically also reflected in the license models, being based, for instance, on the number of installations or users, the richness of the used functionality, or the volume of the event data that may be processed.

12) In the lab: tool support for conformance checking

Check out the lab session to practise with different tools available for conformance checking.

http://www.conformancechecking.com/CC_book_Chapter_12

Chapter 13
Epilogue

To conclude this book, we first summarize the reasons why conformance checking is an important field on its own. Then we report on the different aspects of conformance checking, and how they have been introduced through the book. Finally, we illustrate challenges that can be tackled by the research community and industry to ensure that the widespread application of conformance checking is possible.

Conformance Checking Without doubt, aligning recorded and modelled behaviour is a key enabler for understanding how the processes in an organization work in reality. By looking at the reality from the model's perspective, several insights can be obtained and through continuous monitoring using conformance checking techniques, improved processes can be attained. Last, but not least, through conformance checking, operational processes can automatically be monitored for compliance to regulations and alignment with business goals.

Description of the Book In Part I of the book, the reader is introduced to the field of conformance checking. First, important factors that contextualize the field are considered, as well as the positioning of conformance checking with respect to other disciplines. Also the spectrum and setting of the field is described.

Using intuitive explanations and with the help of a running example, the main inputs of conformance checking techniques are described: event logs and process models. Quality dimensions are then informally introduced, for having an informal understanding on metrics that evaluate the relation between recorded and modelled behaviour. Finally, a qualitative analysis is presented that considers three complementary ways of relating recorded and modelled behaviour: rule checking, token replay and alignments.

Part II of the book is devoted to providing an algorithmic insight to the core techniques for conformance checking. Preliminaries as well as formal definitions of the main elements are introduced. Then we show an in-depth algorithmic description of the state-of-the-art technique for computing so-called *alignments*, which provide

J. Carmona et al., *Conformance Checking*,
https://doi.org/10.1007/978-3-319-99414-7_13

a mathematical foundation to the conformance checking techniques presented in this book. The discussion on variations in the algorithmics and heuristic approaches may trigger future research on improving the computation of alignments, which to date is the lion's share of conformance checking and its most challenging problem.

In Part II, we also reflect on important concerns related to alignments, such as the non-uniqueness of optimal alignments and properties that quality metrics computed on top of alignments should adhere to. Finally, we broaden the scope of alignment-based conformance checking by reporting on variations of the main technique, so that it can be applied for different purposes. We, for example, show how to include the data and resource perspective in conformance checking and we show how conformance checking can be performed in an online setting or in scenarios where the process descriptions are not Petri nets. Also, an alternative for computing alignments using so-called event structures is described.

Part III is focused on applications of conformance checking. It starts by illustrating that conformance artefacts are not the end of the story, but instead enable further analyses to reflect on the process beyond control flow. Among them, we highlight performance analysis and decision point analysis as prominent examples. We also discuss the generalizing ability of process models. Furthermore, we show that conformance checking opens the door to repairing either process models or event logs, so that the representations are modified to better represent the underlying process. We stress the importance of trust in both the log and the model in this section.

Finally we turn our focus to software support for conformance checking, by describing an open-source initiative that contains most of the techniques described in this book. On a more general setting, we then list requisites on the functionality that any software for conformance checking should have in order to satisfy important requirements for its successful application.

Challenges in Conformance Checking Although conformance checking is a well-established field, the maturity of the different techniques varies significantly. One example is the metrics available: Whilst fitness or even precision are considered well evaluated through current metrics, accurate generalization metrics that additionally can be evaluated efficiently are yet to come.

Alignments are a central pillar of current techniques for conformance checking, as has been illustrated through this book. However, the complexity requirements of the state-of-the-art techniques hamper their application for large instances. Alternative approaches, like the decomposition or structural techniques presented in Part II of this book, only alleviate the problem, at the expense of losing the guarantee of important properties like optimality. Also, when incorporating other dimensions like data or resources, so that multi-perspective conformance checking is enabled, the complexity of the problem increases significantly, making it difficult to be applied for real-life problems. We envision new contributions also for multi-perspective conformance checking in the near future that can overcome this limitation. A similar situation is observed with online techniques for conformance checking.

With remarkable exceptions, the industry uptake of conformance checking has been limited so far. This holds both for developing commercial software that incorporates conformance checking features, and for applying conformance checking practices into the daily activities of organizations. Clearly, the former enables the latter, i.e., if more tools were made available for organizations, conformance checking would be applied more often.

In brief, in order to have a population of commercial software for conformance checking, more research should be conducted, so that the important computational challenges are overcome. In spite of the aforementioned limited industry uptake, there exist few tools in the market with conformance checking features. In general, the available tool support for conformance checking mostly applies simple techniques, so that the software can provide results in reasonable time.

The future for conformance checking is ahead us; are you willing to join?

References

1. Arya Adriansyah. *Aligning Observed and Modeled Behavior*. PhD thesis, Technische Universiteit Eindhoven, 2014.
2. Arya Adriansyah and Joos C. A. M. Buijs. Mining process performance from event logs. In *Business Process Management Workshops — BPM 2012 International Workshops, Tallinn, Estonia, September 3, 2012. Revised Papers*, pages 217–218, 2012.
3. Arya Adriansyah, Jorge Munoz-Gama, Josep Carmona, Boudewijn F. van Dongen, and Wil M. P. van der Aalst. Alignment based precision checking. In *Business Process Management Workshops — BPM 2012 International Workshops, Tallinn, Estonia, September 3, 2012. Revised Papers*, pages 137–149, 2012.
4. Arya Adriansyah, Jorge Munoz-Gama, Josep Carmona, Boudewijn F. van Dongen, and Wil M. P. van der Aalst. Measuring precision of modeled behavior. *Inf. Syst. E-Business Management*, 13(1):37–67, 2015.
5. Arya Adriansyah, Natalia Sidorova, and Boudewijn F. van Dongen. Cost-based fitness in conformance checking. In *11th International Conference on Application of Concurrency to System Design, ACSD 2011, Newcastle Upon Tyne, UK, 20–24 June, 2011*, pages 57–66, 2011.
6. Arya Adriansyah, Boudewijn F. van Dongen, and Wil M. P. van der Aalst. Towards robust conformance checking. In *Business Process Management Workshops — BPM 2010 International Workshops and Education Track, Hoboken, NJ, USA, September 13–15, 2010, Revised Selected Papers*, pages 122–133, 2010.
7. Arya Adriansyah, Boudewijn F. van Dongen, and Nicola Zannone. Controlling break-the-glass through alignment. In *International Conference on Social Computing, SocialCom 2013, SocialCom/PASSAT/BigData/EconCom/BioMedCom 2013, Washington, DC, USA, 8–14 September, 2013*, pages 606–611, 2013.
8. Arya Adriansyah, Boudewijn F. van Dongen, and Nicola Zannone. Privacy analysis of user behavior using alignments. *Information Technology*, 55(6):255–260, 2013.
9. Charu C. Aggarwal. *Data Streams: Models and Algorithms*, volume 31 of *Advances in Database Systems*. Springer US, Boston, MA, 2007.
10. Abel Armas-Cervantes, N. R. T. P. van Beest, Marcello La Rosa, Marlon Dumas, and Luciano García-Bañuelos. Interactive and incremental business process model repair. In *On the Move to Meaningful Internet Systems. OTM 2017 Conferences — Confederated International Conferences: CoopIS, C&TC, and ODBASE 2017, Rhodes, Greece, October 23–27, 2017, Proceedings, Part I*, pages 53–74, 2017.
11. Thomas Baier, Jan Mendling, and Mathias Weske. Bridging abstraction layers in process mining. *Inf. Syst.*, 46:123–139, 2014.

J. Carmona et al., *Conformance Checking*,
https://doi.org/10.1007/978-3-319-99414-7

12. Ekaterina Bazhenova and Mathias Weske. Deriving decision models from process models by enhanced decision mining. In *Business Process Management Workshops — BPM 2015, 13th International Workshops, Innsbruck, Austria, August 31 – September 3, 2015, Revised Papers*, pages 444–457, 2015.
13. Seyed-Mehdi-Reza Beheshti, Boualem Benatallah, Sherif Sakr, Daniela Grigori, Hamid Reza Motahari-Nezhad, Moshe Chai Barukh, Ahmed Gater, and Seung Hwan Ryu. *Process Analytics — Concepts and Techniques for Querying and Analyzing Process Data*. Springer, 2016.
14. Albert Bifet, Geoff Holmes, Richard Kirkby, and Bernard Pfahringer. MOA: Massive Online Analysis Learning Examples. *Journal of Machine Learning Research*, 11:1601–1604, 2010.
15. Vincent Bloemen, Sebastiaan van Zelst, Wil van der Aalst, Boudewijn van Dongen, and Jaco van de Pol. Maximizing synchronization for aligning observed and modelled behaviour. In *Business Process Management — 16th International Conference, BPM 2018, Sydney, Australia, September 11–23, 2018. Proceedings*, page Accepted, 2018.
16. R. P. Jagadeesh Chandra Bose, Wil M. P. van der Aalst, Indrė Žliobaitė, and Mykola Pechenizkiy. Dealing with concept drifts in process mining. *IEEE Trans. Neural Netw. Learning Syst.*, 25(1):154–171, 2014.
17. Max Bramer. *Principles of Data Mining*. Springer, 2007.
18. Joos C. A. M. Buijs, Boudewijn F. van Dongen, and Wil M. P. van der Aalst. Quality dimensions in process discovery: The importance of fitness, precision, generalization and simplicity. *Int. J. Cooperative Inf. Syst.*, 23(1), 2014.
19. Andrea Burattin and Josep Carmona. A framework for online conformance checking. In *Proceedings of the 13th International Workshop on Business Process Intelligence (BPI 2017)*, 2017.
20. Andrea Burattin, Fabrizio Maria Maggi, and Alessandro Sperduti. Conformance checking based on multi-perspective declarative process models. *Expert Syst. Appl.*, 65:194–211, 2016.
21. Andrea Burattin, Sebastiaan van Zelst, Abel Armas-Cervantes, Josep Carmona, and Boudewijn F. van Dongen. Online conformance checking using behavioural patterns. In *Business Process Management — 16th International Conference, BPM 2018, Sydney, Australia, September 11–23, 2018. Proceedings*, page Accepted, 2018.
22. Josep Carmona and Ricard Gavaldà. Online techniques for dealing with concept drift in process mining. In Jaakko Hollmén, Frank Klawonn, and Allan Tucker, editors, *Advances in Intelligent Data Analysis XI — 11th International Symposium, IDA 2012, Helsinki, Finland, October 25–27, 2012. Proceedings*, volume 7619 of *Lecture Notes in Computer Science*, pages 90–102. Springer, 2012.
23. Semra Catalkaya, David Knuplesch, Carolina Ming Chiao, and Manfred Reichert. Enriching business process models with decision rules. In *Business Process Management Workshops — BPM 2013 International Workshops, Beijing, China, August 26, 2013, Revised Papers*, pages 198–211, 2013.
24. Federico Chesani, Riccardo De Masellis, Chiara Di Francescomarino, Chiara Ghidini, Paola Mello, Marco Montali, and Sergio Tessaris. Abducing workflow traces: A general framework to manage incompleteness in business processes. In *ECAI 2016 - 22nd European Conference on Artificial Intelligence, 29 August–2 September 2016, The Hague, The Netherlands — Including Prestigious Applications of Artificial Intelligence (PAIS 2016)*, pages 1734–1735, 2016.
25. Raffaele Conforti, Marcello La Rosa, and Arthur H. M. ter Hofstede. Filtering out infrequent behavior from business process event logs. *IEEE Trans. Knowl. Data Eng.*, 29(2):300–314, 2017.
26. Massimiliano de Leoni, Marlon Dumas, and Luciano García-Bañuelos. Discovering branching conditions from business process execution logs. In *Fundamental Approaches to Software Engineering — 16th International Conference, FASE 2013, held as part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16–24, 2013. Proceedings*, pages 114–129, 2013.

27. Massimiliano de Leoni, Fabrizio Maria Maggi, and Wil M. P. van der Aalst. An alignment-based framework to check the conformance of declarative process models and to preprocess event-log data. *Inf. Syst.*, 47:258–277, 2015.
28. Massimiliano de Leoni, Jorge Munoz-Gama, Josep Carmona, and Wil M. P. van der Aalst. Decomposing alignment-based conformance checking of data-aware process models. In *On the Move to Meaningful Internet Systems: OTM 2014 Conferences — Confederated International Conferences: CoopIS, and ODBASE 2014, Amantea, Italy, October 27–31, 2014, Proceedings*, pages 3–20, 2014.
29. Massimiliano de Leoni and Wil M. P. van der Aalst. Aligning event logs and process models for multi-perspective conformance checking: An approach based on integer linear programming. In *Business Process Management — 11th International Conference, BPM 2013, Beijing, China, August 26–30, 2013. Proceedings*, pages 113–129, 2013.
30. Massimiliano de Leoni and Wil M. P. van der Aalst. Data-aware process mining: Discovering decisions in processes using alignments. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, Coimbra, Portugal, March 18–22, 2013*, pages 1454–1461, 2013.
31. Massimiliano de Leoni, Wil M. P. van der Aalst, and Marcus Dees. A general process mining framework for correlating, predicting and clustering dynamic behavior based on event logs. *Inf. Syst.*, 56:235–257, 2016.
32. Massimiliano de Leoni, Wil M. P. van der Aalst, and Boudewijn F. van Dongen. Data- and resource-aware conformance checking of business processes. In *Business Information Systems — 15th International Conference, BIS 2012, Vilnius, Lithuania, May 21–23, 2012. Proceedings*, pages 48–59, 2012.
33. Eduardo González López de Murillas, Wil M. P. van der Aalst, and Hajo A. Reijers. Process mining on databases: Unearthing historical data from redo logs. In Hamid Reza Motahari-Nezhad, Jan Recker, and Matthias Weidlich, editors, *Business Process Management — 13th International Conference, BPM 2015, Innsbruck, Austria, August 31 – September 3, 2015, Proceedings*, volume 9253 of *Lecture Notes in Computer Science*, pages 367–385. Springer, 2015.
34. Javier de San Pedro, Josep Carmona, and Jordi Cortadella. Log-based simplification of process models. In *Business Process Management — 13th International Conference, BPM 2015, Innsbruck, Austria, August 31 – September 3, 2015, Proceedings*, pages 457–474, 2015.
35. Javier de San Pedro and Jordi Cortadella. Discovering duplicate tasks in transition systems for the simplification of process models. In *Business Process Management — 14th International Conference, BPM 2016, Rio de Janeiro, Brazil, September 18–22, 2016. Proceedings*, pages 108–124, 2016.
36. Remco M. Dijkman, Marlon Dumas, and Chun Ouyang. Semantics and analysis of business process models in BPMN. *Information & Software Technology*, 50(12):1281–1294, 2008.
37. Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
38. Marlon Dumas, Wil M. P. van der Aalst, and Arthur H. M. ter Hofstede, editors. *Process-Aware Information Systems: Bridging People and Software Through Process Technology*. Wiley, 2005.
39. Reinhold Dunkl, Stefanie Rinderle-Ma, Wilfried Grossmann, and Karl Anton Fröschl. A method for analyzing time series data in process mining: Application and extension of decision point analysis. In *Information Systems Engineering in Complex Environments — CAiSE Forum 2014, Thessaloniki, Greece, June 16–20, 2014, Selected Extended Papers*, pages 68–84, 2014.
40. Dirk Fahland and Wil M. P. van der Aalst. Simplifying discovered process models in a controlled manner. *Inf. Syst.*, 38(4):585–605, 2013.
41. Dirk Fahland and Wil M. P. van der Aalst. Model repair — aligning process models to reality. *Inf. Syst.*, 47:220–243, 2015.

42. Francesco Folino, Massimo Guarascio, and Luigi Pontieri. Discovering context-aware models for predicting business process performances. In *On the Move to Meaningful Internet Systems: OTM 2012, Confederated International Conferences: CoopIS, DOA-SVI, and ODBASE 2012, Rome, Italy, September 10–14, 2012. Proceedings, Part I*, pages 287–304, 2012.
43. Chiara Di Francescomarino, Chiara Ghidini, Sergio Tessaris, and Itzel Vázquez Sandoval. Completing workflow traces using action languages. In *Advanced Information Systems Engineering — 27th International Conference, CAiSE 2015, Stockholm, Sweden, June 8–12, 2015, Proceedings*, pages 314–330, 2015.
44. Mohamed Medhat Gaber, Arkady Zaslavsky, and Shonali Krishnaswamy. Mining data streams: A review. *ACM Sigmod Record*, 34(2):18–26, jun 2005.
45. João Gama. *Knowledge Discovery from Data Streams*. Chapman & Hall/CRC, 2010.
46. Luciano García-Bañuelos, Nick van Beest, Marlon Dumas, Marcello La Rosa, and Willem Mertens. Complete and interpretable conformance checking of business processes. *IEEE Transactions on Software Engineering*, 44(3):262–290, 2018.
47. Luciano García-Bañuelos, Nick R.T.P. van Beest, Marlon Dumas, Marcello La Rosa, and Willem Mertens. Complete and interpretable conformance checking of business processes. *IEEE Transactions on Software Engineering*, 2017.
48. Johny Ghattas, Pnina Soffer, and Mor Peleg. Improving business process decision making based on past experience. *Decision Support Systems*, 59:93–107, 2014.
49. Lukasz Golab and M. Tamer Özsu. Issues in Data Stream Management. *ACM SIGMOD Record*, 32(2):5–14, jun 2003.
50. Daniela Grigori, Fabio Casati, Malú Castellanos, Umeshwar Dayal, Mehmet Sayal, and Ming-Chien Shan. Business process intelligence. *Computers in Industry*, 53(3):321–343, 2004.
51. Alexander Grosskopf, Gero Decker, and Mathias Weske. *The Process: Business Process Modeling Using BPMN*. Meghan-Kiffer Press, 2009.
52. Christian W. Günther and Wil M. P. van der Aalst. Fuzzy mining - Adaptive process simplification based on multi-perspective metrics. In *Business Process Management, 5th International Conference, BPM 2007, Brisbane, Australia, September 24–28, 2007, Proceedings*, pages 328–343, 2007.
53. Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Systems Science and Cybernetics*, 4(2):100–107, 1968.
54. C. A. R. (Tony) Hoare. Quicksort. *Comput. J.*, 5(1):10–15, 1962.
55. Mieke Jans. From relational database to valuable event logs for process mining purposes: A procedure, 2016. https://businessinformatics.be/wp-content/uploads/2016/12/JANS%20ELB%20PROCEDURE.pdf (last accessed 2018-01-08).
56. Wirat Jareevongpiboon and Paul Janecek. Ontological approach to enhance results of business process mining and analysis. *Business Proc. Manag. Journal*, 19(3):459–476, 2013.
57. Felix Kossak, Christa Illibauer, Verena Geist, Jan Kubovy, Christine Natschläger, Thomas Ziebermayr, Theodorich Kopetzky, Bernhard Freudenthaler, and Klaus-Dieter Schewe. *A Rigorous Semantics for BPMN 2.0 Process Diagrams*. Springer, 2014.
58. Sander J. J. Leemans, Dirk Fahland, and Wil M. P. van der Aalst. Scalable process discovery and conformance checking. *Software & Systems Modeling*, Jul 2016.
59. Niels Lohmann, H. M. W. (Eric) Verbeek, and Remco Dijkman. Petri net transformations for business processes - A Survey. In *Transactions on Petri Nets and Other Models of Concurrency II*, volume 5460 of *LNCS*, pages 46–63. Springer Berlin Heidelberg, 2009.
60. Xixi Lu, Dirk Fahland, and Wil M. P. van der Aalst. Conformance checking based on partially ordered event data. In *Business Process Management Workshops — BPM 2014 International Workshops, Eindhoven, The Netherlands, September 7–8, 2014, Revised Papers*, pages 75–88, 2014.
61. Xixi Lu, Marijn Nagelkerke, Dennis van de Wiel, and Dirk Fahland. Discovering interacting artifacts from ERP systems. *IEEE Trans. Services Computing*, 8(6):861–873, 2015.

62. Abderrahmane Maaradji, Marlon Dumas, Marcello La Rosa, and Alireza Ostovar. Detecting sudden and gradual drifts in business processes from execution traces. *IEEE Trans. Knowl. Data Eng.*, 29(10):2140–2154, 2017.
63. Marco Maisenbacher and Matthias Weidlich. Handling concept drift in predictive process monitoring. In Xiaoqing (Frank) Liu and Umesh Bellur, editors, *2017 IEEE International Conference on Services Computing, SCC 2017, Honolulu, HI, USA, June 25–30, 2017*, pages 1–8. IEEE Computer Society, 2017.
64. Felix Mannhardt, Massimiliano de Leoni, Hajo A. Reijers, and Wil M. P. van der Aalst. Balanced multi-perspective checking of process conformance. *Computing*, 98(4):407–437, 2016.
65. Felix Mannhardt, Massimiliano de Leoni, Hajo A. Reijers, and Wil M. P. van der Aalst. Decision mining revisited - Discovering overlapping rules. In *Advanced Information Systems Engineering — 28th International Conference, CAiSE 2016, Ljubljana, Slovenia, June 13–17, 2016. Proceedings*, pages 377–392, 2016.
66. Felix Mannhardt, Massimiliano de Leoni, Hajo A. Reijers, Wil M. P. van der Aalst, and Pieter J. Toussaint. From low-level events to activities — A pattern-based approach. In *Business Process Management — 14th International Conference, BPM 2016, Rio de Janeiro, Brazil, September 18–22, 2016. Proceedings*, pages 125–141, 2016.
67. Ronny Mans, Wil M. P. van der Aalst, and Rob J. B. Vanwersch. *Process Mining in Healthcare - Evaluating and Exploiting Operational Healthcare Processes*. Springer Briefs in Business Process Management. Springer, 2015.
68. Morten Marquard, Muhammad Shahzad, and Tijs Slaats. Web-based modelling and collaborative simulation of declarative processes. In *Business Process Management — 13th International Conference, BPM 2015, Innsbruck, Austria, August 31 – September 3, 2015, Proceedings*, pages 209–225, 2015.
69. Marco Ajmone Marsan, Gianfranco Balbo, Andrea Bobbio, Giovanni Chiola, Gianni Conte, and Aldo Cumani. The effect of execution policies on the semantics and analysis of stochastic Petri nets. *IEEE Trans. Software Eng.*, 15(7):832–846, 1989.
70. Jan Mendling, Gustaf Neumann, and Wil M. P. van der Aalst. Understanding the occurrence of errors in process models based on metrics. In *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS, OTM Confederated International Conferences CoopIS, DOA, ODBASE, GADA, and IS 2007, Vilamoura, Portugal, November 25–30, 2007, Proceedings, Part I*, pages 113–130, 2007.
71. Jorge Munoz-Gama. *Conformance Checking and Diagnosis in Process Mining - Comparing Observed and Modeled Processes*, volume 270 of *Lecture Notes in Business Information Processing*. Springer, 2016.
72. Jorge Munoz-Gama and Josep Carmona. A fresh look at precision in process conformance. In *Business Process Management — 8th International Conference, BPM 2010, Hoboken, NJ, USA, September 13–16, 2010. Proceedings*, pages 211–226, 2010.
73. Jorge Munoz-Gama and Josep Carmona. Enhancing precision in process conformance: Stability, confidence and severity. In *Proceedings of the IEEE Symposium on Computational Intelligence and Data Mining, CIDM 2011, part of the IEEE Symposium Series on Computational Intelligence 2011, April 11–15, 2011, Paris, France*, pages 184–191, 2011.
74. Jorge Munoz-Gama, Josep Carmona, and Wil M. P. van der Aalst. Single-entry single-exit decomposed conformance checking. *Inf. Syst.*, 46:102–122, 2014.
75. Tadao Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.
76. Hamid R. Motahari Nezhad, Régis Saint-Paul, Fabio Casati, and Boualem Benatallah. Event correlation for process discovery from web service interaction logs. *VLDB J.*, 20(3):417–444, 2011.
77. Mogens Nielsen, Gordon D. Plotkin, and Glynn Winskel. Petri nets, event structures and domains, part I. *Theor. Comput. Sci.*, 13:85–108, 1981.
78. Council of the European Union. General Data Protection Regulation (GDPR). Final version, released 6 April 2016. http://data.consilium.europa.eu/doc/document/ST-5419-2016-INIT/en/pdf.

79. Maja Pesic, Helen Schonenberg, and Wil M. P. van der Aalst. Declarative workflow. In Arthur H. M. ter Hofstede, Wil M. P. van der Aalst, Michael Adams, and Nick Russell, editors, *Modern Business Process Automation — YAWL and its Support Environment*, pages 175–201. Springer, 2010.
80. Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Technische Hochschule Darmstadt, 1962.
81. Artem Polyvyanyy, Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and Moe Thandar Wynn. Impact-driven process model repair. *ACM Trans. Softw. Eng. Methodol.*, 25(4):28:1–28:60, 2017.
82. Artem Polyvyanyy, Matthias Weidlich, Raffaele Conforti, Marcello La Rosa, and Arthur H. M. ter Hofstede. The 4C spectrum of fundamental behavioral relations for concurrent systems. In Gianfranco Ciardo and Ekkart Kindler, editors, *Application and Theory of Petri Nets and Concurrency — 35th International Conference, PETRI NETS 2014, Tunis, Tunisia, June 23–27, 2014. Proceedings*, volume 8489 of *Lecture Notes in Computer Science*, pages 210–232. Springer, 2014.
83. Hernán Ponce de León, Lucio Nardelli, Josep Carmona, and Seppe K. L. M. vanden Broucke. Incorporating negative information to process discovery of complex systems. *Inf. Sci.*, 422:480–496, 2018.
84. Shaya Pourmirza, Remco M. Dijkman, and Paul Grefen. Correlation miner: Mining business process models and event correlations without case identifiers. *Int. J. Cooperative Inf. Syst.*, 26(2):1–32, 2017.
85. J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.
86. Wolfgang Reisig. *Understanding Petri Nets — Modeling Techniques, Analysis Methods, Case Studies*. Springer, 2013.
87. Daniel Reißner, Raffaele Conforti, Marlon Dumas, Marcello La Rosa, and Abel Armas-Cervantes. Scalable conformance checking of business processes. In *25th International Conference On Cooperative Information Systems (CoopIS 2017)*, Rhodes, Greece, March 2017.
88. Andreas Rogge-Solti, Wil M. P. van der Aalst, and Mathias Weske. Discovering Stochastic Petri Nets with Arbitrary Delay Distributions From Event Logs. In *BPM Workshops*, volume 171 of *LNBIP*, pages 15–27. Springer, 2014.
89. Andreas Rogge-Solti and Gjergji Kasneci. Temporal anomaly detection in business processes. In *Business Process Management — 12th International Conference, BPM 2014, Haifa, Israel, September 7–11, 2014. Proceedings*, pages 234–249, 2014.
90. Andreas Rogge-Solti, Ronny S. Mans, Wil M. P. van der Aalst, and Mathias Weske. Improving Documentation by Repairing Event Logs. In *The Practice of Enterprise Modeling*, volume 165 of *LNBIP*, pages 129–144. Springer Berlin Heidelberg, 2013.
91. Andreas Rogge-Solti, Arik Senderovich, Matthias Weidlich, Jan Mendling, and Avigdor Gal. In log and model we trust? A generalized conformance checking framework. In *Business Process Management — 14th International Conference, BPM 2016, Rio de Janeiro, Brazil, September 18–22, 2016. Proceedings*, pages 179–196, 2016.
92. Andreas Rogge-Solti and Mathias Weske. Prediction of business process durations using non-Markovian stochastic Petri nets. *Inf. Syst.*, 54:1–14, 2015.
93. Anne Rozinat. *Process Mining Conformance and Extension*. PhD thesis, Technische Universiteit Eindhoven, 2010.
94. Anne Rozinat, R. S. Mans, Minseok Song, and Wil M. P. van der Aalst. Discovering simulation models. *Inf. Syst.*, 34(3):305–327, 2009.
95. Anne Rozinat and Wil M. P. van der Aalst. Conformance testing: Measuring the fit and appropriateness of event logs and process models. In *Business Process Management Workshops, BPM 2005 International Workshops, BPI, BPD, ENEI, BPRM, WSCOBPM, BPS, Nancy, France, September 5, 2005, Revised Selected Papers*, pages 163–176, 2005.
96. Anne Rozinat and Wil M. P. van der Aalst. Decision mining in ProM. In *Business Process Management, 4th International Conference, BPM 2006, Vienna, Austria, September 5–7, 2006, Proceedings*, pages 420–425, 2006.

97. Anne Rozinat and Wil M. P. van der Aalst. Conformance checking of processes based on monitoring real behavior. *Inf. Syst.*, 33(1):64–95, 2008.
98. Alexander Schrijver. *Theory of linear and integer programming*. Wiley-Interscience series in discrete mathematics and optimization. Wiley, 1999.
99. Arik Senderovich, Andreas Rogge-Solti, Avigdor Gal, Jan Mendling, and Avishai Mandelbaum. The ROAD from sensor data to process instances via interaction mining. In *Advanced Information Systems Engineering — 28th International Conference, CAiSE 2016, Ljubljana, Slovenia, June 13–17, 2016. Proceedings*, pages 257–273, 2016.
100. Arik Senderovich, Matthias Weidlich, Avigdor Gal, and Avishai Mandelbaum. Queue mining — Predicting delays in service processes. In *Advanced Information Systems Engineering - 26th International Conference, CAiSE 2014, Thessaloniki, Greece, June 16–20, 2014. Proceedings*, pages 42–57, 2014.
101. Arik Senderovich, Matthias Weidlich, Liron Yedidsion, Avigdor Gal, Avishai Mandelbaum, Sarah Kadish, and Craig A. Bunnell. Conformance checking and performance improvement in scheduled processes: A queueing-network perspective. *Inf. Syst.*, 62:185–206, 2016.
102. Manuel Silva Suárez, Enrique Teruel, and José Manuel Colom. Linear algebraic and linear programming techniques for the analysis of place or transition net systems. In 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, held in Dagstuhl, September 1996*, pages 309–373, 1996.
103. Jin-Liang Song, Tie-Jian Luo, Su Chen, and Wei Liu. A clustering based method to solve duplicate tasks problem. *Journal of the Graduate School of the Chinese Academy of Sciences*, 26(1):107–113, 2009.
104. Shaoxu Song, Yue Cao, and Jianmin Wang. Cleaning timestamps with temporal constraints. *PVLDB*, 9(10):708–719, 2016.
105. Suriadi Suriadi, Robert Andrews, Arthur H. M. ter Hofstede, and Moe Thandar Wynn. Event log imperfection patterns for process mining: Towards a systematic approach to cleaning event logs. *Inf. Syst.*, 64:132–150, 2017.
106. Niek Tax, Xixi Lu, Natalia Sidorova, Dirk Fahland, and Wil M.P. van der Aalst. The imprecisions of precision measures in process mining. *Information Processing Letters*, 135:1–8, 2018.
107. Niek Tax, Ilya Verenich, Marcello La Rosa, and Marlon Dumas. Predictive business process monitoring with LSTM neural networks. In *Advanced Information Systems Engineering — 29th International Conference, CAiSE 2017, Essen, Germany, June 12–16, 2017, Proceedings*, pages 477–492, 2017.
108. Farbod Taymouri and Josep Carmona. Model and event log reductions to boost the computation of alignments. In *Proceedings of the 6th International Symposium on Data-driven Process Discovery and Analysis (SIMPDA 2016), Graz, Austria, December 15–16, 2016.*, pages 50–62, 2016.
109. Farbod Taymouri and Josep Carmona. A recursive paradigm for aligning observed behavior of large structured process models. In *Business Process Management — 14th International Conference, BPM 2016, Rio de Janeiro, Brazil, September 18–22, 2016. Proceedings*, pages 197–214, 2016.
110. Farbod Taymouri and Josep Carmona. An evolutionary technique to approximate multiple optimal alignments. In *Business Process Management — 16th International Conference, BPM 2018, Sydney, Australia, September 11–23, 2018. Proceedings*, page Accepted, 2018.
111. Wil M. P. van der Aalst. The application of Petri nets to workflow management. *Journal of Circuits, Systems, and Computers*, 8(1):21–66, 1998.
112. Wil M. P. van der Aalst. Decomposing process mining problems using passages. In *Application and Theory of Petri Nets — 33rd International Conference, PETRI NETS 2012, Hamburg, Germany, June 25–29, 2012. Proceedings*, pages 72–91, 2012.
113. Wil M. P. van der Aalst. Decomposing Petri nets for process mining: A generic approach. *Distributed and Parallel Databases*, 31(4):471–507, 2013.

114. Wil M. P. van der Aalst. *Process Mining: Data Science in Action*. Springer, second edition, 2016.
115. Wil M. P. van der Aalst, Arya Adriansyah, and Boudewijn F. van Dongen. Replaying history on process models for conformance checking and performance analysis. *Wiley Interdisc. Rew.: Data Mining and Knowledge Discovery*, 2(2):182–192, 2012.
116. Wil M. P. van der Aalst, Marlon Dumas, Chun Ouyang, Anne Rozinat, and H. M. W. (Eric) Verbeek. Choreography conformance checking: An approach based on BPEL and Petri nets. In *The Role of Business Processes in Service Oriented Architectures, 16.07. - 21.07.2006*, 2006.
117. Wil M. P. van der Aalst, Marlon Dumas, Chun Ouyang, Anne Rozinat, and H. M. W. (Eric) Verbeek. Conformance checking of service behavior. *ACM Trans. Internet Techn.*, 8(3):13:1–13:30, 2008.
118. Wil M. P. van der Aalst, M. H. Schonenberg, and Minseok Song. Time prediction based on process mining. *Inf. Syst.*, 36(2):450–475, 2011.
119. Wil M. P. van der Aalst, A. J. M. M. (Ton) Weijters, and Laura Maruster. Workflow mining: Discovering process models from event logs. *IEEE Trans. Knowl. Data Eng.*, 16(9):1128–1142, 2004.
120. Boudewijn F. van Dongen. Efficiently computing alignments using the extended marking equation. In *Business Process Management — 16th International Conference, BPM 2018, Sydney, Australia, September 11–23, 2018. Proceedings*, page Accepted, 2018.
121. Boudewijn F. van Dongen, Josep Carmona, and Thomas Chatain. A unified approach for measuring precision and generalization based on anti-alignments. In *Business Process Management — 14th International Conference, BPM 2016, Rio de Janeiro, Brazil, September 18–22, 2016. Proceedings*, pages 39–56, 2016.
122. Boudewijn F. van Dongen, Josep Carmona, Thomas Chatain, and Farbod Taymouri. Aligning modeled and observed behavior: A compromise between computation complexity and quality. In *Advanced Information Systems Engineering — 29th International Conference, CAiSE 2017, Essen, Germany, June 12–16, 2017, Proceedings*, pages 94–109, 2017.
123. Sebastiaan J. van Zelst, Alfredo Bolt, Marwan Hassani, Boudewijn F. van Dongen, and Wil M. P. van der Aalst. Online conformance checking: Relating event streams to process models using prefix-alignments. *International Journal of Data Science and Analytics*, Oct 2017.
124. Sebastiaan J. van Zelst, Alfredo Bolt, and Boudewijn F. van Dongen. Tuning alignment computation: An experimental evaluation. In *Proceedings of the International Workshop on Algorithms & Theories for the Analysis of Event Data 2017 Satellite event of the conferences: 38th International Conference on Application and Theory of Petri Nets and Concurrency Petri Nets 2017 and 17th International Conference on Application of Concurrency to System Design ACSD 2017, Zaragoza, Spain, June 26–27, 2017.*, pages 6–20, 2017.
125. Seppe K. L. M. vanden Broucke, Jorge Munoz-Gama, Josep Carmona, Bart Baesens, and Jan Vanthienen. Event-based real-time decomposed conformance analysis. In *On the Move to Meaningful Internet Systems: OTM 2014 Conferences — Confederated International Conferences: CoopIS, and ODBASE 2014, Amantea, Italy, October 27–31, 2014, Proceedings*, pages 345–363, 2014.
126. Seppe K. L. M. vanden Broucke, Jochen De Weerdt, Jan Vanthienen, and Bart Baesens. Determining process model precision and generalization with weighted artificial negative events. *IEEE Trans. Knowl. Data Eng.*, 26(8):1877–1889, 2014.
127. H. M. W. (Eric) Verbeek and Wil M. P. van der Aalst. Merging alignments for decomposed replay. In *Application and Theory of Petri Nets and Concurrency — 37th International Conference, PETRI NETS 2016, Toruń, Poland, June 19–24, 2016. Proceedings*, pages 219–239, 2016.
128. Jianmin Wang, Shaoxu Song, Xuemin Lin, Xiaochen Zhu, and Jian Pei. Cleaning structured event logs: A graph repair approach. In Johannes Gehrke, Wolfgang Lehner, Kyuseok Shim, Sang Kyun Cha, and Guy M. Lohman, editors, *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13–17, 2015*, pages 30–41. IEEE Computer Society, 2015.

129. Ingo Weber, Andreas Rogge-Solti, Chao Li, and Jan Mendling. CCaaS: Online Conformance Checking as a Service. In *Proceedings of the BPM Demo Session 2015 Co-located with the 13th International Conference on Business Process Management (BPM 2015), Innsbruck, Austria, September 2, 2015.*, pages 45–49, 2015.
130. Ingo Weber, Xiwei Xu, Régis Riveret, Guido Governatori, Alexander Ponomarev, and Jan Mendling. Untrusted business process monitoring and execution using blockchain. In *Business Process Management — 14th International Conference, BPM 2016, Rio de Janeiro, Brazil, September 18–22, 2016. Proceedings*, pages 329–347, 2016.
131. Matthias Weidlich. *Behavioural Profiles : A Relational Approach to Behaviour Consistency*. doctoral thesis, Universität Potsdam, 2011.
132. Matthias Weidlich, Artem Polyvyanyy, Nirmit Desai, Jan Mendling, and Mathias Weske. Process compliance analysis based on behavioural profiles. *Inf. Syst.*, 36(7):1009–1025, 2011.
133. Matthias Weidlich, Artem Polyvyanyy, Jan Mendling, and Mathias Weske. Causal behavioural profiles — Efficient computation, applications, and evaluation. *Fundam. Inform.*, 113(3–4):399–435, 2011.
134. A. J. M. M. (Ton) Weijters, Wil M. P. van der Aalst, and A. K. Alves De Medeiros. Process mining with the heuristics miner-algorithm. Technical Report 166, Technische Universiteit Eindhoven, 2006.
135. Mathias Weske. *Business Process Management - Concepts, Languages, Architectures, 2nd Edition*. Springer, 2012.
136. XES Working Group. 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.
137. Haiping Zha, Jianmin Wang, Lijie Wen, Chaokun Wang, and Jiaguang Sun. A workflow net similarity measure based on transition adjacency relations. *Computers in Industry*, 61(5):463–471, 2010.

Index

J. Carmona et al., *Conformance Checking*,
https://doi.org/10.1007/978-3-319-99414-7