BISE Student

https://bise-student.io

MASTER'S THESIS

Object-Centric Process Constraints using Variable Bindings

Publication Date: 2024-10-25

Author
Aaron KÜSTERS
Aachen, Germany
aaron.kuesters@rwth-aachen.de
0xc71e30529d01Fb81e87b6920B6c9ece6861916bA

Abstract

With increasing interest and availability of Object-Centric Event Data (OCED), the focus of process mining research is shifting towards object-centric techniques. OCED removes the requirement of a single case notion, i.e., that all events only belong to exactly one case. Thus, OCED can capture real-life processes much more accurately. In this thesis, we introduce a declarative querying and constraint approach for OCED, focusing specifically on very high expressiveness while still allowing for efficient execution in practice. We first present and formalize a way to formulate nested queries of combinations of objects and events, so-called variable bindings. In contrast to prior work, our approach allows for querying combinations of multiple objects and events of any types. For example, also permitting queries for two orders placed by the same customer, one placed after the other. Constraints are presented as an extension to the querying approach, additionally specifying for each queried binding if it should be considered...

Keywords: process mining, machine learning, data science

Submission Date: 2024-10-24
Submission Contract: 0xC20149ca5623c29c0B0bcB0B455AeF44585Db2cC

This work was submitted to:

Chair of Process and Data Science (PADS - Informatik 9), RWTH Aachen University

Object-Centric Process Constraints using Variable Bindings

Master's Thesis

Author:	**Aaron Küsters**
Student ID:	**395258**

Supervisor:	Prof. Dr. Wil van der Aalst
Examiners:	Prof. Dr. Wil van der Aalst Prof. Dr. Stefan Decker

Registration Date:	2024-04-22
Submission Date:	2024-09-11

Abstract

With increasing interest and availability of Object-Centric Event Data (OCED), the focus of process mining research is shifting towards object-centric techniques. OCED removes the requirement of a single case notion, i.e., that all events only belong to exactly once case. Thus, OCED can capture real-life processes much more accurately. In this thesis, we introduce a declarative querying and constraint approach for OCED, focusing specifically on very high expressiveness while still allowing for efficient execution in practice. We first present and formalize a way to formulate nested queries of combinations of objects and events, so-called *variable bindings*. In contrast to prior work, our approach allows for querying combinations of multiple objects and events of any types. For example, also permitting queries for two orders placed by the same customer, one placed after the other. Constraints are presented as an extension to the querying approach, additionally specifying for each queried binding if it should be considered *satisfied* or *violated*. We introduce a visual notation for the queries and constraints of our approach and present a supporting tool implementation. Apart from the approach formalization, we also describe how the proposed types of declarative queries and constraints can be efficiently algorithmically evaluated on input OCED. Additionally, we outline how some types of constraints, which are very relevant for real-life processes, can be discovered automatically based on input OCED. Finally, we also evaluate the presented query and constraint approach by showcasing example constraints, demonstrating the high expressiveness and convenient visual representation of simple and complex constraints. Moreover, we explore the scalability and runtime performance of our approach implementation, showing excellent performance even for large real-life datasets with more than one million events.

Contents

Chapter 1

Introduction

Process mining aims to provide insights about processes based on event data, i.e., recorded process execution records. Most commonly, business processes are analyzed. In larger organizations, event data is often recorded by software systems assisting in the real-life process execution, for instance ERP systems. Traditionally, most of the data is considered in a flat format, where events are associated with exactly one case. However, recently, research and industry started to focus on a more flexible data model: Object-Centric Event Data (OCED). In OCED, multiple objects are considered and events can be associated with multiple objects of the same type. The flexibility of OCED allows it to represent real-life processes much more accurately than previous, flat event data [1]. Depending on the application area, additional concepts, like relationships between objects or other event and object attributes can be supported. The OCEL 2.0 specification aims to standardize a reference file format for such object-centric data [2]. In the last few years, more and more *Object-Centric Process Mining* (OCPM) techniques based on OCED have been proposed [3]. These techniques allow analyzing process data without flattening it into a traditional format, like classical event logs. In particular, the problems of *Convergence*, *Divergence*, and *Deficiency* as described in [1, 4] are circumvented.

With an increase in interest and availability of OCED, many process mining techniques have been extended to the object-centric realm. For instance, [5] describes the discovery of an object-centric variant of Petri nets based on input OCED or [6] presents constrain-based conformance checking for OCED. Yet, many of such OCPM adaptations of existing techniques limit their expressiveness and complexity by only *combining traditional, flat constructs across multiple object types*, without actually connecting them. For instance, prior work on constraints, like [6], allows considering multiple constraint constructs involving only one object type (e.g., constraints like “there should be exactly one `pay order` event per `orders` object”), across multiple object types (e.g., `orders`, `items`, ...). However, these constraint parts are not combined. Thus, they cannot express advanced constraints involving more than two object types or consider multiple instances of the same type. As such, many of the previously presented object-centric process mining approaches can be categorized as being primarily *multi-object* techniques instead of fully *object-centric*. This also offers some advantages, as it allows transferring a broad set of previously presented approaches to the object-centric setting, without increasing the complexity of the approach. Still, multi-object approaches do not leverage the full potential and flexibility of the OCED data model.

In this thesis, we aim to take a different direction. We present an object-centric query and constraint approach, focusing on the full flexibility and expressiveness of OCED. The instances of an OCED are objects and events. Our querying approach allows specifying names and types for the different objects and events that should be bound as well as declarative predicates, which specify in what relationships the queried instances should be. Other filters, for example on instance attributes (e.g., an order object's `price`), can also be added. Thus, our approach uses the full flexibility and structure of OCED, allowing queries and constraints involving any number of objects or events of arbitrary types.

Next, we motivate the need for expressive object-centric queries and constraints, and the advantage of an easy-to-use visual and declarative approach.

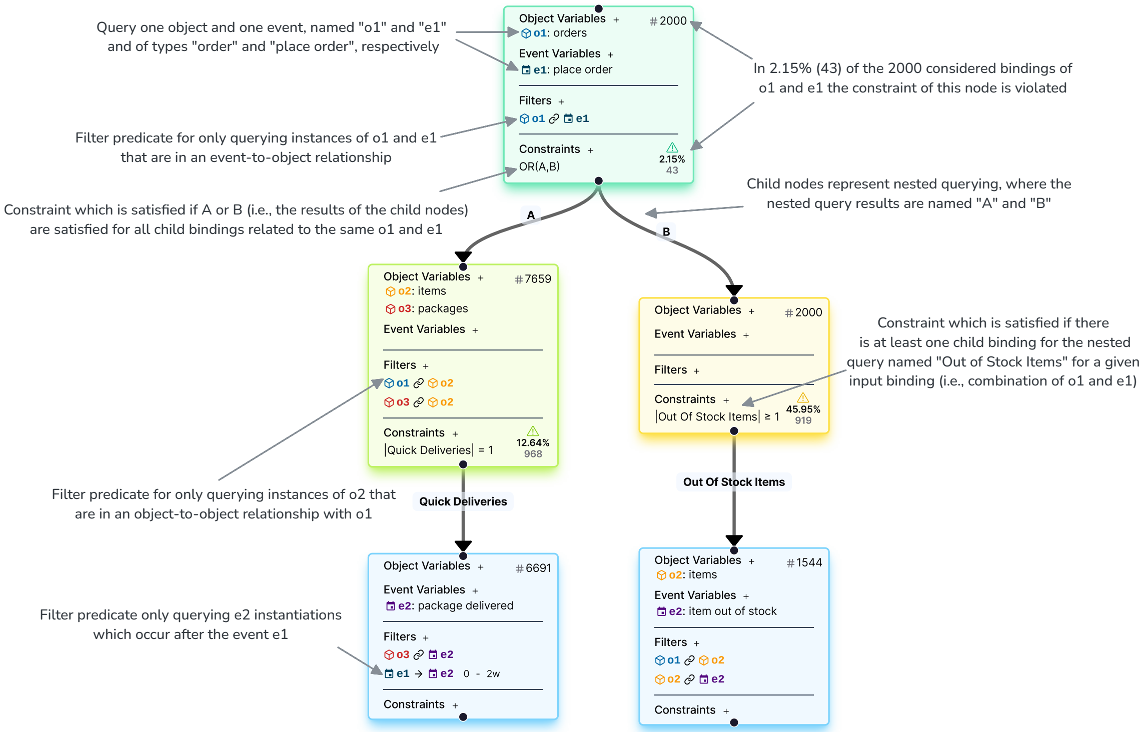
1.1 Motivation

In organizations, process execution data contain valuable insights that are often not leveraged to their full extent. Allowing stakeholders to identify and query interesting scenarios enables organizations to get an overview of their current situation, as well as further use the query results for advanced analysis. In particular, such analysis possibilities are crucial for real-life improvements of the process. An especially intriguing analyzation facet lays in identifying undesired behavior in the process. Undesired behavior can have various shapes. Consider a procurement process, for example, where sometimes invoices are paid more than once, imposing significant costs to an organization either due to excess work or because the lost funds are never recovered. Or consider an order management process, for instance, where failing to send a payment reminder to customers with overdue payments is clearly undesirable and has real-life negative consequences. Undesired behavior can also be more complex, for instance when a customer in an order management process places two orders after each other, but receives confirmations in the opposite order, causing confusion.

For real-life adoption, it is especially important to allow stakeholders and analysts to design such queries for interesting scenarios themselves. Thus, approaches that require advanced technical skills, like SQL or other general-purpose solutions, cannot be used. In particular, to be usable also for non-technical users, a graphical representation of the querying language, as well as a tool with a user interface for modeling and executing queries, are integral.

Most prior work on querying process instances and defining constraints is focused on flat event data. However, as demonstrated in [4], OCED can represent real-life business process much better than traditional, flat event logs. Moreover, OCED is much closer to the execution data recorded in the databases of large ERP systems. In the context of querying and constraints, the flexible data model of OCED enables large leaps in expressiveness, for instance formulating constraints across instances of different objects and event types. However, this flexibility also introduces more complexity: Previously, with a single case notion, it was clear what constraints like "There should be exactly one `pay order` event" or "If three `failed delivery` events are recorded, a `cancel shipment` event should occur afterwards" expressed. For object-centric data, it is unclear *what events* related to *which objects* are meant. Most of the object-centric query and constraint approaches presented so far are still based on implicit case notions. In these approaches, the case notions are just considered to be more flexible, with the option to choose different object types for the case notion for different constraints. As such, these techniques are best described as *multi-object* instead of fully *object-centric*.

For instance, consider the following textual constraint formulation for an order management process: “An order should be fully delivered within 2 weeks of order placement, unless an item of the order is out of stock.” This constraint is not representable in previously proposed constraint approaches, even multi-objects ones, as it involves querying and connecting events and objects through different object types. For instance, it can be assumed that the `package delivered` event is only associated with objects of type `packages`, which are never associated with `place order` events. In our approach, this constraint can be expressed through nested querying and is presented visually in Figure 1.1.


Query one object and one event, named "o1" and "e1" and of types "order" and "place order", respectively

In 2.15% (43) of the 2000 considered bindings of o1 and e1 the constraint of this node is violated

Filter predicate for only querying instances of o1 and e1 that are in an event-to-object relationship

Child nodes represent nested querying, where the nested query results are named "A" and "B"

Constraint which is satisfied if A or B (i.e., the results of the child nodes) are satisfied for all child bindings related to the same o1 and e1

Constraint which is satisfied if there is at least one child binding for the nested query named "Out of Stock Items" for a given input binding (i.e., combination of o1 and e1)

Filter predicate for only querying instances of o2 that are in an object-to-object relationship with o1

Filter predicate only querying e2 instantiations which occur after the event e1

Figure 1.1: A visualized constraint in our approach, consisting of five nodes, which are colored to indicate how often they are violated for queried bindings. Annotations are included to explain the different components of the constraint. In the top node, `orders` objects, named `o1`, together with their corresponding `place order` event (`e1`), are queried. Additionally, it contains an OR-constraint, which renders the root node satisfied exactly for those `o1` and `e1` combinations, where one of the direct child nodes is satisfied. The root node is violated for 2.15% of the 2000 queried combinations of `o1` and `e1`. The first node of the left child tree is satisfied for a given placed order (i.e., combination of `o1` and `e1`), if all items (`o2`) in the order `o1` and corresponding packages (`o3`) have exactly one result in the nested query, named `Quick Deliveries`. This nested query queries all `package delivered` events related to the package `o3`, which occur within 2 weeks after `e1`. Similarly, the first node of the right child tree is satisfied if there is at least one result in its subquery, named `Out of Stock Items`.

Next, we will present the problem statement, research questions and goals as well as contributions of this thesis and outline the remaining thesis structure.

1.2 Problem Statement

Extracting insights from business process executions is challenging, especially when considering the added complexity that object-centric event data can model. Identifying interesting or problematic process instances, i.e., combinations of objects and events, based on queries and constraints is especially valuable for organizations. There is also a need for more expressive queries and constraints, leveraging the full flexibility of object-centric event data, to overcome the limitations of previously proposed multi-object approaches. Furthermore, it is important to also allow stakeholders not familiar with programming to easily design and execute queries and constraints.

1.3 Research Questions

In this thesis, we pose and subsequently address the following research questions:

RQ1 What are types of queries or constraints not expressible in previously suggested graphical query or constraint approaches?

RQ2 What are exploitable connections or similarities between queries and constraints for OCED?

RQ3 How to increase expressiveness, while still allowing for easy graphical modeling of constraints and queries for OCED?

RQ4 How to achieve good performance for executing queries and checking constraints even for larger event data, while maintaining high expressiveness?

RQ5 How can (a limited subset of) constraints automatically be discovered based on input OCED?

1.4 Research Goals

To answer these research questions, we identify the following research goals:

RG1 Conduct a literature review of traditional and object-centric querying and constraints, with a focus on the expressiveness of the proposed approaches.

RG2 Analyze the textual formulation for example constraints and identify or propose general patterns.

RG3 Conceptualize and formalize an expressive object-centric querying and constraint approach.

RG4 Develop an algorithm for efficient evaluation of the presented (declarative) query approach.

RG5 Implementation of a graphical tool supporting the proposed query and constraint approach, focusing on performance and usability.

RG6 Design and implement a basic discovery algorithm for specific types of constraints.

RG7 Evaluation of the implemented approach, both regarding runtime performance and expressiveness of the possible constraints and queries.

1.5 Contributions

The main contribution of this thesis is the proposed object-centric query and constraint approach based on the concept of variable bindings. Additionally, we contribute the implementation of the full-stack software tool *OCPQ* (Object-Centric Process Querying), featuring an intuitive graphical user interface for designing constraints and queries as well as a high-performance execution engine backend for computing the results of queries. This tool implementation not only demonstrates the feasibility of the approach, but also makes it easy to apply the presented approach in practice on real data.

In the following, we present a more detailed list of our contributions as part of this thesis:

CT1 An overview of related approaches and literature regarding traditional and object-centric querying and constraint models.

CT2 A conceptualization and formal definition of an object-centric nested querying approach with the possibility to model complex queries and constraints.

CT3 Algorithms that allow for efficient evaluation of the declarative queries formulated in our approach.

CT4 The user-friendly tool *OCPQ* for designing and evaluating queries and constraints, focusing on performance with the option to discover certain types of constraints automatically.

CT5 Evaluation of the proposed approach through the implemented tool *OCPQ*, investigating runtime performance and scalability as well as expressiveness.

CT6 Several upstream contributions to the process mining software landscape, in particular adding OCEL 2.0 data models, XML, and JSON importers to the *Rust4PM* project[1].

1.6 Thesis Structure

The remainder of this thesis is structured as follows. In Chapter 2, we first present and discuss related work on process querying and (declarative) process constraints. We additionally describe their limitations, especially concerning their expressiveness. Next, in Chapter 3, we introduce key concepts and definitions used throughout this thesis. Then, in Chapter 4, we present our main approach for object-centric queries and constraints. Additionally, this chapter contains descriptions of how the proposed declarative query formulations can be evaluated efficiently algorithmically. Furthermore, we also describe how certain types of constraints can be automatically mined from input data and sketch extensions for our presented approach. In Chapter 5, we give an overview of our implementation and provide details about the *OCPQ* tool, consisting of the execution engine backend and the graphical constraint editor frontend. We evaluate our approach through the implemented tool in Chapter 6, covering both runtime performance and a qualitative analysis using example queries and constraints. In Chapter 7, we discuss the design decisions we made for our method and implementation, as well as possible limitations. Finally, we conclude this thesis in Chapter 8 and give a brief outlook into interesting areas for future work.

[1] `https://github.com/aarkue/rust4pm`

Chapter 2

Related Work

In this chapter, we discuss related work on the topics of process querying and process constraints. As such, this chapter addresses **RQ1** by providing **RG1** and encompassing the corresponding contribution **CT1** of a literature overview over query and constraint approaches.

This chapter is structured as follows. We first examine prior work presenting declarative and constraint-based approaches for classical, flat (i.e., not object-centric) event data. Subsequently, we also explore process querying, as a related subject to our approach. Afterwards, we zoom in on constraint approaches that support object-centric event data. Finally, we shortly discuss process filtering and sketch how the introduced concepts, like declarative process models, constraints, querying, and filtering are fundamentally related.

2.1 Traditional Declarative Process Models

Most process models and languages, like Petri nets or BPMN, are imperative. They describe the control flow of how a process works, just like imperative programming involves a sequence of steps, specifying, what to do, in which order. In contrast, a declarative process model approach describes *what* behavior should be allowed or disallowed, just like declarative programming specifies, what a program should compute, without specifying explicit steps of *how* to do so.

Process constraints specify what process behavior should not be allowed, forming the very core of declarative process modeling. As such, they are a good fit for modeling processes with large flexibility and little clear structure [7]. For instance, knowledge-intensive processes, like those in health care or finance, commonly exhibit such loose structure, allowing workers a high amount of flexibility in how to handle process cases [7].

2.1.1 DECLARE

In 2006, Pesic and van der Aalst presented *ConDec*, a declarative process model approach based on constraints rooted in Linear Temporal Logic (LTL) [8]. Because LTL expressions can be complex and difficult to understand quickly, the authors used parameterized constraint templates as an intermediate representation for constraints. For instance, a *response constraint*, represented by a decorated arc between two activities A and B, corresponds to the LTL expression $\Box(A \rightarrow \Diamond B)$, expressing that events of type A should be eventually followed by events of type B [8]. The visual representation of the response constraint is shown in Figure 2.1. The same authors also created

DecSerFlow, a sister language of *ConDec*, focusing on service flows, and using the same concepts and tool implementation [9]. In 2007, Pesic et al. presented *DECLARE*, a workflow management system prototype using constraint-based process modeling [10]. While *DecSerFlow* and *ConDec* can be understood as specific languages created using the DECLARE framework, DECLARE itself allows for the definition of own custom languages (containing *constraint templates*) rooted in LTL [10]. This includes specifying the visual representation of *constraint templates*, with the goal of also allowing users without experience in LTL to work with DECLARE tools [8, 10].

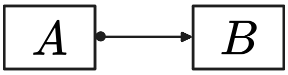


Figure 2.1: A *response constraint* between activities A and B in DECLARE (cf. [8–10]).

DECLARE consists of three parts: 1. *Designer* for modeling, 2. *Framework* for process enactment, and 3. *Worklist* for process execution by (multiple) workers [10]. To support the execution and enactment, LTL formulas are internally converted to finite-words automata. This also enables detecting whether a violated constraint is *temporarily* or *permanently* violated (e.g., if the automaton is in a sink state the violation is *permanent*) [10]. The *Worklist* view of DECLARE then highlights the violation status of a constraint and also disables actions which are forbidden by constraints.

Discovering DECLARE Constraints

DecMiner In [11], the authors proposed an approach for discovering DECLARE (DecSerFlow) constraints using Inductive Logic Programming (ILP). ILP is concerned with inducing a set of logical rules, generalized from training examples [11, 12]. The algorithm requires labeled event logs, i.e., a set of positive and negative traces, exhibiting allowed or disallowed behavior, respectively. These positively or negatively labeled traces are considered positive or negative interpretations in the context of ILP. Constraint clauses are added or refined, such that they rule out negative interpretations while still permitting (most of) the positive interpretations. The generated constraints are expressed in the $\mathcal{S}$CIFF language, where, for instance, $\mathbf{H}(\textit{place order}, T) \rightarrow \mathbf{E}(\textit{pay order}, T1) \wedge T1 > T$ expresses that, for any given case, an event with the *place order* activity at timestamp T is eventually followed by an event with the *pay order* activity at a later timestamp $T1$. Note, that the variable T is implicitly universally qualified ($\forall$), while $T1$ is implicitly existentially qualified ($\exists$) [11]. Finally, the constraints are translated into DecSerFlow constraints.

Declare Miner In 2010, Maggi et al. published a discovery algorithm for mining DECLARE models from event logs [13]. The algorithm uses a simple brute-force approach: Given a set of DECLARE templates to use, all possible DECLARE constraints following these templates are generated and translated to LTL [13]. Subsequently, the LTL constraints are checked for an historical event log. If an LTL constraint is not satisfied, the corresponding DECLARE constraint is removed. The algorithm features some parameters, allowing to ignore infrequent activities for generating constraints and allowing a certain percentage of traces in the historical event log to violate the constraint without filtering it out. Later, in 2012, some of the same authors improved on this brute-force discovery approach by reducing the considered search space using an Apriori algorithm[1] [15]. This approach not only reduces the execution time drastically and thus

[1] Inspired by the classic Apriori algorithm for mining association rules by Agrawal and Srikant published in [14].

enables mining DECLARE constraints for larger events logs, but also mitigates the discovery of unwanted constraints (e.g., constraints that are trivially true or implied by other constraints) [15]. The algorithm works like this: First, construct frequent item sets of activities using an Apriori algorithm. Next, for each DECLARE constraint template of size k, generate possible instantiations using frequent item sets of size k (and smaller frequent item sets while repeating some activities). The number of generated instantiations is expected to be much smaller than in the primitive approach presented in [13]. Finally, the set of instantiations is additionally filtered based on classic association rule metrics, adopted for DECLARE constraints (e.g., Support, Confidence, Interest Factor), and corresponding threshold parameters. In 2018, Maggi et al. extended this approach, focusing on faster execution time and parallelization [16].

MINERful In [17], Di Ciccio and Mecella presented the MINERful algorithm in detail. It works in two phases: First, statistical information is extracted from the event log to build certain abstractions in the form of knowledge bases. For instance, the knowledge base contains counts in how many traces an activity occurs $n \in \mathbb{N}_0$ times, or counts how often two activities occur within a distance of $d \in \mathbb{Z}$ (i.e., with d activities between them, where negative distances are considered right-to-left). Secondly, queries are used to construct fitting DECLARE constraints from these knowledge bases. For that, custom support functions for each DECLARE construct are proposed. For instance, the $\mathit{Existence}(n, a)$ constraint, which specifies that activity a occurs at least $n \in \mathbb{N}_0$ times, has an associated support function of $1 - \sum_{i=0}^{n-1} \gamma_a(i) \,/\, |L|$, where $\gamma_a(i)$ counts how frequently a occurs i times in traces of L and $|L|$ refers to the number of traces in the event log L. MINERful is especially focused on efficient runtime speed, taking only a few seconds for event logs, with a usual number of activities (e.g., up to 50 unique activities) [17]. The knowledge base is constructed using a single pass through all cases of the log, but is quadratic w.r.t. the number of activities in the log and the length of the traces [17].

2.1.2 DCR Graphs

Dynamic Condition Response Graphs (DCR Graphs) are declarative process models in the form of a directed graph of *event nodes*, representing executable elements of the model[2], and *edges*, which assign one of four relations between the connected event nodes [18]. DCR Graphs were first presented by Hildebrandt and Mukkamala in 2010 [18]. It was the authors' solution to an industry use case that required to *dynamically* remove certain constraints based on other constraints [7]. Such dynamic relaxations are not possible in DECLARE, where the models correspond to the conjunction of individual constraints [7]. The semantics of DCR Graphs are defined by transformation of markings, similar to how token markings are used for Petri nets [7, 18]. DCR Graph markings consist of three binary indicators per event node:

Ex indicating whether the event node has already been executed in the past
Re indicating whether the event node is pending (i.e., required to be executed or excluded)
In indicating whether the event node is included (i.e., currently relevant to the process)

In the context of finite traces, a DCR Graph marking is accepting, if there are no event nodes which are both included (In) and pending (Re) [7, 18, 19]. Playing out a DCR Graph corresponds to executing event nodes, which transfers its current marking to an updated one. In a given DCR Graph and marking, an event node e can be executed only if certain conditions apply (i.e., e has to be included and event nodes, that are a condition of e, have to be executed or excluded). This

[2] Not to be confused with past process execution elements, which we typically call *events*. Multiple DCR Graph events can have the same activity, similar to multiple transitions with the same activity label in a labeled Petri net.

transforms the given marking to an updated marking, based on the response, condition, excludes and includes relations involving e. For instance, a response relation $e \bullet\!\!\rightarrow e'$ means that e' will become pending in the updated marking [18].

This marking approach is different from the semantics of DECLARE, where LTL formulas are internally used to check if a given event sequence satisfies the constraints, without a stateful marking of the DECLARE constraints.

Discovering DCR Graphs

Iterative Relaxation In [20], the authors presented the first algorithm to automatically discover DCR Graphs [7]. It takes a similar approach to the *Declare Miner* from [13], and uses iterative relaxation: first considering a model with all possible constraints involving the activities of the log [20]. In particular, between all pairs of activities, there are condition, exclusion, and response relations in this initial model. Next, the traces of the input event log are iterated and all constraints prohibiting the observed behavior are removed. For instance, if at the end of a trace there is an activity with an unfulfilled response relation, this relation is removed. Through this relaxation approach, the final discovered model is guaranteed to have perfect fitness, i.e., allow all behavior in the input event log.

ParNek and DisCoveR In 2019 Nekrasaite et al. presented the DCR Graph discovery algorithm *ParNek* [21]. ParNek takes an opposite approach, starting with an empty model and then adding constraint relations using multiple independent algorithms. The authors evaluated ParNek and compared it to the iterative relaxation discovery algorithm from [20], as well as the MINERful algorithm for discovering DECLARE constraints from [17]. The evaluation suggested that ParNek discovered significantly simpler models compared to [20] while still providing comparable precision and also performed similarly well overall compared to MINERful. Later, Back et al. expanded the general approach of ParNek in [19] with *DisCoveR*, focusing especially on fast runtime.

2.1.3 Understandability of Declarative Models

A key advantage of declarative process model languages is the simplicity of declarative models for unstructured and knowledge-intensive processes [7]. However, a key issue still lays in the *understandability* of declarative models, especially considering that imperative modeling languages are more common and popular. In [22], the authors investigated the understandability of DECLARE models compared to imperative BPMN models. The subjects, students enrolled in business process management courses, were provided with reference material and were then asked to complete different tasks for equivalent BPMN and DECLARE models. On average, the subjects' answers were both more accurate and faster for the imperative BPMN model. The authors note, however, that the subjects were already more familiar with imperative process modeling languages than with declarative ones.

In [23], the authors conducted another study, instructing subjects to think out loud while describing DECLARE models. This also allowed analyzing the approach subjects take to *read* DECLARE models. The results indicate that people attempt to read the DECLARE models sequentially, for instance trying to first identify an entry point of the process, although a clear entry point might not exist for declarative models. In a follow-up study presented in the same paper, further scenarios, for instance, involving the layout of the presented DECLARE models, were tested. Most findings of the initial study could be confirmed and refined in the follow-up study. Further findings include, that subjects struggled especially with combinations of constraints, while single constraints posed fewer problems in understandability.

2.2 Process Querying

Process Querying research covers filtering and manipulation of process repositories, based on a (formal) query [24]. There exist many types of approaches, with different input, output and goals, under the umbrella of Process Querying. For instance, process repositories might contain process models as well as process executions. Some process querying methods are concerned with selecting process models matching a given query from the repository, others with selecting instances (i.e., cases) from event data. In [24], Polyvyanyy et al. categorize Process Querying research into four groups: *Structural Querying*, *Behavioral Querying*, *Process Execution Querying*, and *Event Log Querying*. In the following, we will focus only on *Event Log Querying* approaches, which are based on only event data as input, without a corresponding process model, as this category is most closely related to our presented approach.

Fazzinga et al. presented *Log Activity Queries* in [25], allowing for both aggregated and unaggregated queries of activity executions (i.e., events) or process executions (i.e., cases). The approach allows specifying both case-level and event-level query criteria, and also permits specifying event-level criteria for the events in a queried case.

In [26], the authors describe the *Process Instance Query Language* (PIQL). It allows querying the number of either process instances (i.e., cases) or process tasks (i.e., events) fulfilling specified criteria. PIQL is specified with formally defined syntax, but English language patterns, that translate to this formal syntax, are also presented with the goal of allowing usage by non-technical people.

The proprietary *Celonis Process Query Language* (Celonis PQL) introduced in [27] is a very comprehensive domain-specific querying language, with more than 150 implemented operators. Like many other querying languages (e.g., [28]), Celonis PQL is heavily inspired by SQL but focuses specifically on process mining functionality, while omitting less relevant general query features. Celonis PQL is tightly integrated in the commercial product offered by Celonis. For example, this allows queries to use an already defined underlying data model schema, which eliminates the need to specify table joins manually or explicitly (as would be the case with `JOIN` in SQL). Celonis PQL also allows both aggregated and unaggregated queries, called *KPI* and *dimension*, respectively. An example aggregating Celonis PQL query is showcased in Code 1.

Celonis PQL also supports querying process cases based on a regex of their activity trace. For example, the regex `'A' >> 'B' >> ('C' | 'D')` matches cases where first `A` and then `B` is executed with either `C` or `D` following afterwards.

All the previously mentioned work on process querying is mostly focused on traditional, flat event data. However, in [29], Esser et al. describe storing multidimensional (i.e., object-centric) event data in graph databases. The authors describe how the input event log is represented in the graph database and detail the required transformation to insert input event data (provided as CSV files) to the graph database. The authors consider event data in the traditional, flat XES format as example input event logs (e.g., the BPI Challenge 2017 log from [30]) which are enhanced with domain knowledge to identify and model different involved entities. The declarative graph querying language *Cypher* can then be used to query entities or subgraphs as well as aggregated

```
COUNT(DISTINCT CASE WHEN
            "Acts"."Activity" = 'Check Application'
            AND "Acts"."Activity" = ACTIVITY_LEAD("Acts"."Activity",2)
            AND "Acts"."Activity" != ACTIVITY_LEAD("Acts"."Activity",1)
        THEN "Acts"."CaseID"
        ELSE NULL
        END)
/  COUNT_TABLE("Cases")
```

Code 1: An example Celonis PQL query calculating the fraction of cases exhibiting *ping-pong behavior*, i.e., cases where there are two `Check Application` activities with one other activity in between. The `ACTIVITY_LEAD` is a process-mining specific function, which returns the next event row for the given offset based on the current event row. Example inspired by [27].

values from the graph database. A simple example Cypher query is shown in Code 2. The authors also present more complex queries, for example for determining the distance, in the form of number of activities, between events of certain types associated with a common offer object.

```
MATCH (o:Entity {EntityType: 'Offer'}) <-[:CORR]- (e1:Event) -[:DF*
 ↪   {EntityType: 'Offer'}]-> (e2:Event)
WHERE e1.Activity = "O_Created" AND e2.Activity = "O_Cancelled"
RETURN e1,e2
```

Code 2: A Cypher graph database query for a graph database based on the BPI Challenge 2017 Loan Application process (see [30]). It queries all `Offer` objects with associated events $e1$ and $e2$, with activities `O_Created` and `O_Cancelled`, respectively, where $e1$ is *eventually followed* by $e2$. Adapted from [29].

For the mentioned dataset, the query shown in Code 2 yields 20,898 results and can be executed in around 140ms according to [29]. However, measuring the execution time of Cypher queries in Neo4J is nontrivial, and it is unclear what measure exactly the authors reported in [29]. For instance, the execution time reported by Neo4J by default only measures the time until the first result is available. As most query operations are streamed and evaluated lazily, the evaluation time reported by Neo4J might be misleading. Moreover, queries and their results might be cached, which improves performance in real-life use cases but makes it more difficult to conduct performance measurements reproducibly. Executing the query shown in Code 2 but returning only the total count (i.e., using `RETURN count(*)` instead of the last line) on the freshly restarted Neo4J database takes 1216ms on our machine (see Chapter 6 for hardware specifications). However, the returned result summary reports that the result was available after 447ms.

2.3 Object-Centric Process Constraints

Next, we present related work for object-centric constraints or object-centric declarative process models.

2.3.1 OCBC

In [31], the authors presented *Object-Centric Behavioral Constraint* (OCBC) models. OCBC models combine behavioral constraints, in the form of a Behavioral Constraint Model (BCM) inspired by DECLARE patterns, with object classes, in the form of a ClaM data model, a simplified version of UML class diagrams. Considered on their own, the BCM defines constraints similar to DECLARE, e.g., that for the `Place Order` activity there should be a `Pay Order` activity as a response. Similarly, the ClaM model on its own describes relationships between the object classes, for instance that every item is associated with exactly one order and an order contains at least one item. However, if put together the DECLARE-style constraint relations in the BCM are considered for the case notion of specified objects of the object classes, indicated by dashed and dotted arrows between the ClaM and BCM model. An example OCBC constraint model for an order management process is shown in Figure 2.2. For instance, the dashed line annotated with "1 : 1" between the `Order` object class in ClaM and the `Place Order` activity in BCM models that for each event of type `Place Order` there should be exactly one involved object of type `Order` and the other way around (i.e., each `Order` should be involved with exactly one `Place Order` event). The approach also allows for even more expressive constraints: For example, specifying that for every `Place Order` event there is a `Pick Item` response event, involving an `Item` object that is associated with the same `Order` object as the aforementioned `Place Order` event.

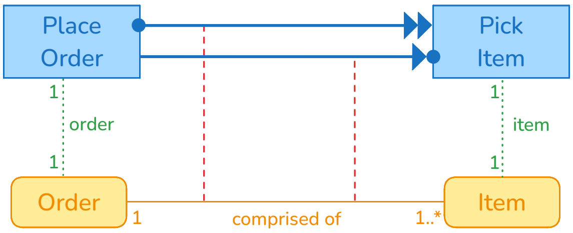


Figure 2.2: An example OCBC constraint. The BCM submodel, shown on the top in blue, contains DECLARE-like constructs. On the bottom, the ClaM data submodel is shown. Both models are connected to express constraints, for instance specifying that there should be exactly one `Place Order` event per `Order` object and vice versa. Figure inspired by [31].

In [32], the authors described a discovery method for OCBC models based on object-centric input event data. First, basic relation cardinalities constraints (e.g., the allowed number of `Pay Order` events per `Order` object or the allowed number of `Items` associated with an `Order`) can be derived by simply counting the relations in the input data and generalizing based on the counts. Discovering the DECLARE-style constraints between activities based on the data models is more involved: For that, two activities and the either one or two related object classes need to be identified, for which constraints between the activities, based on the object classes should be discovered. The authors present two methods for identification: 1) a *triangle pattern* involving a single object class and two activities, where instances of the object class are associated with events of both activities and 2) a *square pattern* involving two object classes with a class relation and two activities, where related instances of the object classes $o1$ and $o2$ are associated with events of the activities $a1$ and $a2$, respectively. Based on the identified activities and object classes, all possible behavioral constraints between the activities (corresponding to the possible DECLARE constraint relations between them), can be checked for fitness based on the data and, if appropriate, added to the discovered model. As extensions to this main approach, the authors also present additional filtering for simplifying the discovered OCBC models by considering the *support* of the constraints. Furthermore, the authors discuss how to deal with infrequent behavior in the input log (i.e., noise) with the goal to discover more precise models. Specifically, for selecting which behavioral constraints to add between two activities (given already identified object class(es)), different discrete variants of possible source and target event counts are considered, based on their frequency. For instance, if only the variant $(1;\ 1)$ (i.e., there is one allowed target activity before the reference event, and also one allowed after the reference activity) is frequent, response relations in both directions should be added to the model. If, on the other hand, only the variant $(0;\ 1)$ (i.e., there is one allowed target event after the reference event, but none before the reference event) is frequent, a unary response and a non-precedence between the reference and target activity should be added to the model.

2.3.2 OCCG

In [6], the authors introduced the concept of *Object-Centric Constraint Graphs* (OCCGs). These constraint graphs can capture interactions between objects and events, as well as control-flow between event types based on a given object type. Additionally, performance metrics regarding events can be included. Given an OCCG, an object-centric event log can be checked against it, resulting in a Boolean value, which indicates if the OCCG is violated for the log. An OCCG is a directed graph made up of nodes, which correspond to either an activity, an object type or a performance formula (e.g., waiting times). The nodes in the graph are connected through different types of edges: *Control-flow edges* connect two activity nodes and specify that there is a causal, concurrent or choice relationship between the activities. Additionally, there can be a *skip* control-flow edge involving just one activity node, specifying that the activity is not executed. All these control-flow edges are associated with a specific object type. This object type determines in relation to which objects the control-flow is considered. Thus, the individual control-flow edges only consider one object-type, effectively selecting this object type as the case notion in this context. Similarly, *Object-involvement edges* connect an object type node with an activity node. They are associated with a count range, specifying the number of objects of the connected object type should be associated with events of the specified activity. *Performance edges* connect a performance formula (e.g., average waiting time in for the last two days) with the activity for which the formula should apply. Figure 2.3 shows a few example OCCG constructs.

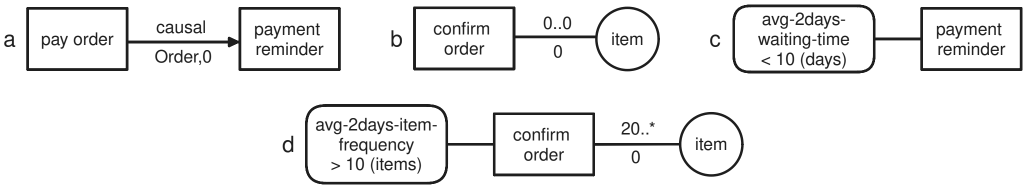


Figure 2.3: Four example OCCG constructs labeled a – d. Activities are represented as rectangles, object types as circles and performance metrics as rounded rectangles. Put into words these constraints encompass: a) No payment reminder after an order was paid; b) No order confirmation without any involved items; c) The 2-day-average waiting time for the activity “payment reminder” should be less than 10 days; d) No confirmation of orders with more than 20 items if the 2-day-average item count for the “confirm order” activity is over 10 items. Inspired by [6].

2.3.3 OCCM

In [33], Adams et al. introduced the concept of object-centric process executions, attempting to adapt the concept of cases for traditional event data to an object-centric setting. Object-centric process executions are graphs consisting of events as nodes, where directed edges between events indicate that there exists an object for which these events directly follow each other. These graphs can be constructed for a given subset of objects, with the requirement that all the objects have to be somehow connected by events. Naturally, there are many subsets of objects satisfying these criteria to choose. The authors describe two general methods to select such subsets: Choosing all maximal subsets satisfying the criteria (corresponding to connected components in a constructed object graph) or (manually) selecting a leading object type and for each object of that type constructing an object subset "centered" around this object.

Based on the concept of object-centric process execution from [33], Object-Centric Constraint Models (OCCMs) are presented in [34]. An OCCM consists of three different subgraphs: A Process Flow Cardinality Constraint Model (CCM), a Process Flow Temporal Constraint Model (TCM), and a Performance Constraint Model (PCM). The CCM contains directed edges between activity nodes, indicating that the events of the source activity should be followed by activities of the target activity. Additionally, each edge has a preceding and succeeding cardinality, which indicates the number of events of the source or target activities that should occur before or after one event of the source or target activity, respectively. TCMs also contain activity nodes and directed edges between them, which constrain the minimum or maximum time duration between events corresponding to the connected activities. Not all events in the process execution with the corresponding activity are considered, but only either the first or last one. For that, edges are assigned to a source and a target temporal pattern, which can be either be `first` or `last`. Finally, the PCM component can include multiple different types of constraints, for example regarding the number of objects of a given object type associated with events of a specified activity. Additionally, it also allows specifying lower and upper limits on the number of occurrences of a given activity there should be in a process execution. Furthermore, the waiting time of events for a specified activity can also be constrained in PCMs. In Figure 2.3 we present a few example OCCM constructs.

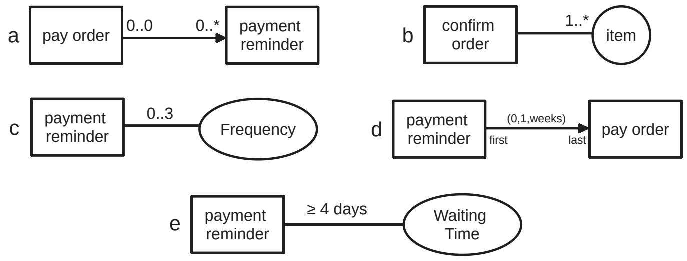


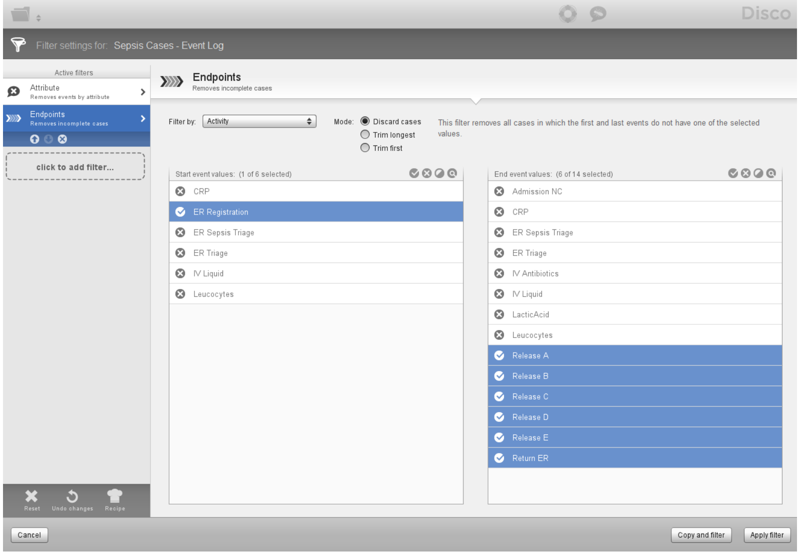
Figure 2.4: Five example OCCM constructs labeled a – e. Activities are represented as rectangles, object types as circles and time/frequency performance types as wide ellipses. Put into words these constraints encompass: a) No payment reminder after an order was paid; b) No confirmation of orders with 0 items; c) At most three payment reminders; d) The (last) pay order event should occur within 1 week of the first payment reminder; e) The waiting time for events of type "payment reminder" should be at least 4 days. Inspired by [34].

2.4 Process Filtering

Many process mining tools, like Fluxicon Disco[3], allow users to filter event data as well as explore and export the resulting filtered view. Disco, for instance, supports filtering both cases and

[3] See `https://fluxicon.com/disco/`.

individual events. A screenshot of its user interface is shown in Figure 2.5. For example, event-level attributes can be used to filter out all events for which the attribute value is not included in a specified list of allowed options. Case-level filter criteria includes the option to filter cases based on the first and last event activity of the case. Moreover, the event attribute filters can also be modified such that all cases, where at least one event fulfills the filter criteria, are included completely (i.e., retaining all events of the case).



Filter settings for: Sepsis Cases - Event Log

Endpoints
Removes incomplete cases

Filter by: Activity

Mode: ● Discard cases ○ Trim longest ○ Trim first

This filter removes all cases in which the first and last events do not have one of the selected values.

Start event values: (1 of 6 selected)

- ☒ CRP
- ✓ ER Registration
- ☒ ER Sepsis Triage
- ☒ ER Triage
- ☒ IV Liquid
- ☒ Leucocytes

End event values: (6 of 14 selected)

- ☒ Admission NC
- ☒ CRP
- ☒ ER Sepsis Triage
- ☒ ER Triage
- ☒ IV Antibiotics
- ☒ IV Liquid
- ☒ LacticAcid
- ☒ Leucocytes
- ✓ Release A
- ✓ Release B
- ✓ Release C
- ✓ Release D
- ✓ Release E
- ✓ Return ER

Figure 2.5: A screenshot of the Disco filtering interface for creating an *Endpoints* filter, which filters cases based on the activity of their first and last event.

There is not much work published which addresses filtering object-centric event data. While in [35], Berti presents approaches for filtering and sampling object-centric event data, the techniques largely focus on simple filters for removing noise or infrequent data.

2.5 Discussion

The related work we discussed in this chapter covered *declarative process models*, *process constraints*, *process querying* and *process filtering*. All these concepts are relevant for our presented approach. While there are some substantial differences between these concepts, we also consider them to be closely related to each other. First off, declarative process models commonly contain multiple (declarative) constraint constructs, which can mostly also be considered separately on their own. A connection between constraints and querying then manifests for many approaches, where constraints or declarative models do not impose restrictions globally, but for a specific process case or event (or also for objects if OCED is considered). Then, identifying the violated instances has a strong correspondence with querying instances based on some criteria (i.e., the negated constraint). Similarly, filtering event logs is usually also done by imposing certain filter predicates for cases or events, which define if the instance should be included or filtered out. Those filter predicates can again also be understood as constraints, where only the non-violating instances should be included in the output. However, filtering additionally is assumed to produce a complete, well-formed event log as output. In contrast, querying techniques oftentimes only output log fragments, for example individual events without a case concept.

Chapter 3

Preliminaries

In this chapter, we define preliminary mathematical notation and other concepts used throughout the later parts of this thesis. We first start with mathematical basics, like basic logical expressions, sets, or partial functions. Next, we cover the concept of universes, used to express all entities of a certain type (e.g., all events, event types, variable names, etc.). Finally, we define Object-Centric Event Data (abbreviated as OCED), which lays the formal basis for our presented approach, which operates on such object-centric data.

3.1 Basics

We first introduce the general notation we use to express logic and conditions.

> **Definition 3.1 (Basic Logic and Conditions):** We use the usual notation for logic statements and conditions. For instance, we say that the expression $1 = 1$ is true and $1 = 2$ is false. For two expressions φ and ψ, we use $\varphi \land \psi$ for the conjunction (i.e., AND) and $\varphi \lor \psi$ for disjunction (i.e., OR). The negation of an expression φ is written as $\neg\ \varphi$. Sometimes, operators are negated using strikethroughs, for instance, $1 \neq 2$ is true and $1 \neq 1$ is false.
> For convenience, we sometimes map truth values of expressions to the numbers 0 (for false) and 1 (for true). As such, we can, for example, define a function $f(x) = (x \geq 5) \lor (x = 3)$ with the domain $f : \mathbb{N} \rightarrow \{0, 1\}$, where we then simply write $f(1) = 0$ and $f(3) = 1$.

Next, we introduce our notation style for *sets of elements.*

> **Definition 3.2 (Sets):** We write $X = \{x_1, x_2, x_3, \ldots\}$ to denote an (unordered) set of elements. The size (or cardinality) of X is written as $|X|$. If X is infinite, we write $|X| = \infty$. Otherwise, if X is finite, we can also name all elements $X = \{x_1, x_2, x_3, \ldots, x_n\}$ (where n implicitly denotes the size of the set; i.e., $n = |X|$). The empty set (i.e., the set of size 0) is denoted with $\emptyset$ or $\{\}$.
> We write $x \in X$ to specify that x is an element of X and $x \notin X$ otherwise. We use set-builder notation for constructing filtered sets based on a condition: $\{x \in X \mid \textit{condition on } x\}$.
> Given two sets X and Y, we write $X \cap Y = \{x \in X \mid x \in Y\}$ for the intersection. We write $X \subseteq Y$ if all elements of X are also elements of Y (i.e., if for all $x \in X$ it holds that $x \in Y$). We call X a *subset* of Y.
> For a set Z and two subsets $X \subseteq Z$ and $Y \subseteq Z$, we write $X \cup Y = \{x \in Z \mid x \in X \lor x \in Y\}$ for the union of the sets. Most of the time, Z is not given explicitly.

A set which is often used throughout this thesis is the set of natural numbers $\mathbb{N}_0$.

Definition 3.3 (Natural Numbers and Integers $\mathbb{N}_0$ $\mathbb{N}_0^\infty$ $\mathbb{Z}$): The set of natural numbers is defined as $\mathbb{N}_0 = \{0, 1, 2, \dots\}$. We define $\mathbb{N}_0^\infty = \mathbb{N}_0 \cup \{\infty\}$ to be the set of natural numbers with an additional element representing infinity (∞).
The set of integers $\mathbb{Z}$ contains both negative and positive numbers, i.e., $\mathbb{Z} = \{0, 1, -1, 2, -2, \dots\}$.

A special subset of the natural numbers we frequently use to keep definitions short is the set of numbers 1 to k, for a given $k \in \mathbb{N}_0$.

Definition 3.4 (Set of $\{1, \dots, n\}$): For brevity, we write $\underline{k} := \{1, \dots, k\}$ with $\underline{k} = \emptyset$ if $k < 1$.

We also consider timestamps and durations. For simplicity, we formally consider timestamps (and thus also durations) to correspond to the set of real numbers.

Definition 3.5 (Timestamps and Durations $\mathbb{T}$): We represent timestamps as values in $\mathbb{T} := \mathbb{R}$. As such, a time value $t \in \mathbb{T}$ can be considered as the number of seconds since the UNIX epoch. We also use $\mathbb{T}$ for durations (i.e., $t_2 - t_1 \in \mathbb{T}$ for $t_1, t_2 \in \mathbb{T}$). For durations, we sometimes give values using commonly used units (e.g., in *hours* or *days*). For convenience, we write $\mathbb{T}^\infty := \mathbb{T} \cup \{-\infty, \infty\}$ for arbitrarily large or small delays or timestamps.

Given a set, we can also consider its *powerset*, which contains all possible subsets of the given set.

Definition 3.6 (Powerset $\mathcal{P}$): Given a set A, we write $\mathcal{P}(A) := \{B \mid B \subseteq A\}$ for the powerset of A.

Partial functions are similar to standard functions, however only a subset of the source elements have to be mapped to a value.

Definition 3.7 (Partial functions): We make use of partial functions to formalize assignments of variable names to values. In particular, for a partial function $f : A \nrightarrow B$, we write $f(a) = \perp$ for any $a \in A$ where $a \notin \mathrm{dom}(f)$. We also use set notation (e.g., $\{1 \mapsto 2, 2 \mapsto 3\}$) for partial functions. For convenience, we sometimes use $\perp$ as a value (e.g., as an element in a set). For two partial functions $f : A \nrightarrow B$ and $g : A \nrightarrow B$, we can form the (asymmetric) union, where g takes precedence over f:

$$f \overrightarrow{\cup} g = a \mapsto \begin{cases} g(a), & \text{if } a \in \mathrm{dom}(g) \\ f(a), & \text{otherwise} \end{cases}$$

For the special case, where $\mathrm{dom}(f) \cap \mathrm{dom}(g) = \emptyset$ holds, we also write $f \cup g = f \overrightarrow{\cup} g$.

Normal functions which additionally map distinct elements to distinct values are called *injective functions.*

Definition 3.8 (Injective Function): A function $f : A \longrightarrow B$ is injective, if distinct elements of A are mapped to distinct values in B, in particular:

$$\forall_{b,b' \in B} \; b \neq b' \Rightarrow f(b) \neq f(b')$$

To identify the closest values in $\mathbb{Z}$ either below or above a given value in $\mathbb{R}$, we define floor and ceiling functions.

Definition 3.9 (Floor and Ceiling Functions): For a real number $x \in \mathbb{R}$ we define the floored and ceiled value of x as:

- $\lfloor x \rfloor = \max\{n \in \mathbb{Z} \mid n \leq x\}$ (Floor)
- $\lceil x \rceil = \min\{n \in \mathbb{Z} \mid n \geq x\}$ (Ceil)

For instance $\lfloor 3.2 \rfloor = 3$ and $\lceil 3.2 \rceil = 4$.

Finally, we introduce multisets as extensions of sets, which additionally represent the number of times a value is contained in it.

Definition 3.10 (Multiset): A multiset is an extension of the concepts of a set allowing for duplicate values. In particular, a multiset of values in a set A can be represented as a function $c : A \to \mathbb{N}_0$, which assigns each element $a \in A$ to a count. We write $\mathcal{B}(A) = A \to \mathbb{N}_0$ for the set of all multisets (also called *bags*) of A. To simplify notation, we represent multisets using square brackets and by listing duplicates separately.
For instance, $m = [2, 2, 4, 5, 6, 5]$ is a multiset of the natural numbers (i.e., $m \in \mathcal{B}(\mathbb{N}_0)$) where the values 2 and 5 occur two times each. Alternatively, duplicate values can also be represented by including their count as a superscript, for instance as $m = [2^2, 4, 5^2, 6]$.

3.2 Universes

We introduce multiple universes for different purposes in Definition 3.11. For most of these universes, elements of them can already be distinguished based on the naming convention indicated by the mentioned example universe elements.

Definition 3.11 (Universes): Let $\mathcal{U}_\Sigma$ be the universe of strings. We use the following pairwise disjoint universes:

$\mathcal{U}_{ev} \subseteq \mathcal{U}_\Sigma$	Universe of events	(e.g., e_1)
$\mathcal{U}_{obj} \subseteq \mathcal{U}_\Sigma$	Universe of objects	(e.g., o_1)
$\mathcal{U}_{etype} \subseteq \mathcal{U}_\Sigma$	Universe of event types (aka *activities*)	(e.g., `confirm order`)
$\mathcal{U}_{otype} \subseteq \mathcal{U}_\Sigma$	Universe of object types	(e.g., `orders`)
$\mathcal{U}_{attr} \subseteq \mathcal{U}_\Sigma$	Universe of attribute names	(e.g., `time`)
$\mathcal{U}_{qual} \subseteq \mathcal{U}_\Sigma$	Universe of relationship qualifiers	(e.g., `places`)
$\mathcal{U}_{obVar} \subseteq \mathcal{U}_\Sigma$	Universe of object variable names	(e.g., `o1`)
$\mathcal{U}_{\mathrm{evVar}} \subseteq \mathcal{U}_\Sigma$	Universe of event variable names	(e.g., `e1`)
$\mathcal{U}_{\mathrm{setName}} \subseteq \mathcal{U}_\Sigma$	Universe of binding set names	(e.g., `A`)

Additionally, let $\mathcal{U}_{val}$ be the universe of all values (with, for instance, $\mathbb{T} \subseteq \mathcal{U}_{val}$, $\mathcal{U}_\Sigma \subseteq \mathcal{U}_{val}$, $\mathcal{U}_{qual} \times \mathcal{U}_{obj} \subseteq \mathcal{U}_{val}$). We also reserve some special symbols like $\perp$ or $*$, which are used to indicate missing values or wildcard placeholders, and assume they are included in any of these universes.

3.3 Object-Centric Event Data (OCED)

Based on the universes introduced before, we formally introduce *object-centric event data*. The formalization of OCED presented in Definition 3.12 is inspired by the OCEL 2.0 standard [2]. However, we keep our formalization very flexible. For instance, we do not define a set of available object or event attributes based on the type of an event or object.

Definition 3.12 (OCED): Object-centric event data (OCED) can be described as a tuple $L = (E, O, eaval, oaval)$ of the following components:

- **Events** $E \subseteq \mathcal{U}_{ev}$ set of events
- **Objects** $O \subseteq \mathcal{U}_{obj}$ set of objects
- **Event Attributes** $eaval : E \to (\mathcal{U}_{attr} \nrightarrow \mathcal{U}_{val})$, which provides attribute values for events. For convenience, we write $eaval_e = eaval(e)$ for an $e \in E$ as a shorthand. The following properties should hold for $eaval$:
  - $\forall_{e \in E}\ eaval_e(\texttt{activity}) \in \mathcal{U}_{etype}$: each event has exactly one *event type*
  - $\forall_{e \in E}\ eaval_e(\texttt{objects}) \subseteq \mathcal{U}_{qual} \times O \wedge eaval_e(\texttt{objects}) \neq \emptyset$: each event has at least one qualified reference to an object
  - $\forall_{e \in E}\ eaval_e(\texttt{time}) \in \mathbb{T}$: each event has a timestamp
- **Object Attributes** $oaval : O \to (\mathcal{U}_{attr} \times \mathbb{T} \nrightarrow \mathcal{U}_{val})$, which provides the attribute

values of an object $o \in O$ at a concrete timestamp. For convenience, we write $oaval_o^t(attr) = oaval(o)(attr, t)$ for a given $o \in O, t \in \mathbb{T}$ and $attr \in \mathcal{U}_{attr}$ as a shorthand. The following properties should hold for *oaval*:

- For the *time-stable* attributes $a \in \{\texttt{objects}, \texttt{type}\} \subseteq \mathcal{U}_{attr}$, the assigned value should not change over time. In particular, it should hold that $\forall_{o \in O} \forall_{t \in \mathbb{T}} \forall_{t' \in \mathbb{T}}\ oaval_o^t(a) = oaval_o^{t'}(a)$. For ease of notation, we simply write $oaval_o(a) = oaval_o^t(a, t)$ with an arbitrary $t \in \mathbb{T}$ for these attributes.
- $\forall_{o \in O}\ oaval_o(\texttt{type}) \in \mathcal{U}_{otype}$: every object has exactly one *object type*
- $\forall_{o \in O}\ oaval_o(\texttt{objects}) \subseteq \mathcal{U}_{qual} \times O$: an object can, optionally, contain qualified references to (other) objects

In Example 3.1, we present a small example OCED of an imaginary order management process inspired by [36]. Later examples are also themed around an order management OCED, but are not directly based on this example and the given object and event sets.

Example 3.1 (Simple OCED): To represent an example OCED, we first define the set of objects as $O = \{o_1, o_2, o_3, o_4\}$ and the set of events $E = \{e_1, e_2, e_3, e_4, e_5, e_6\}$. The attribute values of all objects and events are then presented in the tables below. Time-stable object attributes are marked with an $*$ in the timestamp column. Apart from the mandatory attributes, the following two custom attributes are added: The customer o_1 has an attribute `city`, indicating the city of residence of the customer. After providing the city initially in 2016, the attribute was updated in 2018, as the customer moved residency. Moreover, the `payment reminder` event e_5 has an attribute `fee`, indicating the additional fine incurred by the late payment (e.g., 15€).

Table 3.1: Object Attribute Table

Object	Attribute	Timestamp	Value
o_1	`type`	$*$	`customers`
o_1	`objects`	$*$	$\{(\texttt{places}, o_2)\}$
o_1	`city`	`2016-01-06T14:15`	`Bonn`
o_1	`city`	`2018-09-03T10:32`	`Aachen`
o_2	`type`	$*$	`orders`
o_2	`objects`	$*$	$\{(\texttt{contains}, o_3), (\texttt{contains}, o_4)\}$
o_3	`type`	$*$	`items`
o_4	`type`	$*$	`items`

Table 3.2: Event Attribute Table

Event	Attribute	Value
e_1	`activity`	`place order`
e_1	`objects`	$\{(\texttt{customer}, o_1), (\texttt{order}, o_2), (\texttt{item}, o_3), (\texttt{item}, o_4)\}$
e_2	`activity`	`pack item`
e_2	`objects`	$\{(\texttt{item}, o_3)\}$
e_3	`activity`	`pack item`
e_3	`objects`	$\{(\texttt{item}, o_4)\}$
e_4	`activity`	`ship items`
e_4	`objects`	$\{(\texttt{ships}, o_3), (\texttt{ships}, o_4)\}$
e_5	`activity`	`payment reminder`
e_5	`objects`	$\{(\texttt{recipient}, o_1), (\texttt{order}, o_2)\}$
e_5	`fee`	15
e_6	`activity`	`pay order`
e_6	`objects`	$\{(\texttt{order}, o_2)\}$

The resulting metamodel of our OCED formalization is shown in Figure 3.1. This metamodel includes both the formal definition and further formalized assumptions (e.g., that every event has exactly one activity attribute).

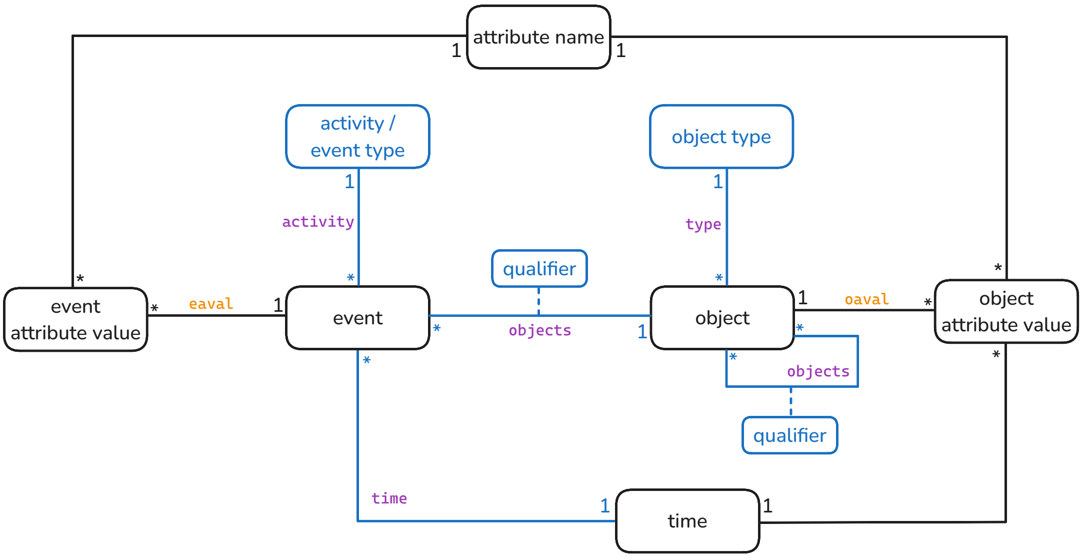


Figure 3.1: Metamodel of OCED used in this thesis. Mainly, we allow for events and objects, each with optional attributes. Object attributes are generally also associated with a timestamp. Formally, the attribute values are provided by the functions *eaval* and *oaval*. Other relationships are implemented using these attributes and are marked in blue in the metamodel. For instance, the activity of an event is simply a special attribute that every event has exactly once. The special attribute names corresponding to other relationships are shown in purple, like, for instance, `activity`.

Next, we define some convenient shorthand notations for OCED, which allows us to keep later definitions and formalizations simpler and shorter.

Note 3.1 (OCED Notation Shorthands): For simplicity, we use the following notations for a given OCED $L = (E, O, eaval, oaval)$:

- $E_L = E$ and $O_L = O$ for the set of objects or events of L, respectively
- Function $type_L \in O \cup E \rightarrow \mathcal{U}_{otype} \cup \mathcal{U}_{etype}$, which assigns *object or event types* to objects or events, defined as:

$$type_L(x) = \begin{cases} oaval_x(\texttt{type}), & \text{if } x \in O \\ eaval_x(\texttt{activity}), & \text{if } x \in E \end{cases}$$

- Function $time_L \in E \rightarrow \mathbb{T}$ with $time_L(e) = eaval_e(\texttt{time})$, which maps an event to its timestamp
- Given an optional qualifier $q \in \mathcal{U}_{qual} \cup \{*\}$, we define the function $obj_L^q \colon (E \cup O) \rightarrow \mathcal{P}(O)$, which assigns an event or object to its set of object references, defined as:

$$obj_L^q(x) = \begin{cases} \{o \mid (q', o) \in eaval_x(\texttt{objects}) \land (q = * \lor q = q')\}, & \text{if } x \in E \\ \{o \mid (q', o) \in oaval_x(\texttt{objects}) \land (q = * \lor q = q')\}, & \text{if } x \in O \end{cases}$$

Additionally, for simplicity we also write $obj_L = obj_L^*$ for all object references without considering the qualifiers.

Chapter 4

Process Queries and Constraints using Variable Bindings

In this chapter, we present the query and constraint approach proposed in this thesis. We first give an introduction into the intuition of our approach and then present the structure of the remaining chapter. As such, this chapter as a whole will address research questions **RQ2**, **RQ3**, **RQ4** and **RQ5** as well as the corresponding research goals and contributions.

Our approach makes full use of the structure and relationships of OCED and is inspired by how constraints are formulated mathematically or in natural language. To motivate this intuition behind our concept of constraints, we first want to look at an example constraint in Example 4.1 and dissect its textual description, which contains multiple, nested queries for events and objects. This constraint, as well as later examples, are formulated for a fictional order management process OCED, inspired by the simulated dataset from [36].

> **Example 4.1 (Simple constraint):** Consider the following textual description of a constraint: "Every placed order not paid within two weeks *should* have at least one payment reminder sent". If dissected, this description contains multiple parts, which correspond to *nested querying of events and objects*:
>
> - "Every placed order": Query all objects `o1` of type `orders` with corresponding `confirm order` event `e1`.
>   - (A) "paid within two weeks': Query all `pay order` events `e2` associated with `o1` that occur at most two weeks after `e1`.
>   - (B) "payment reminder sent": Query all `payment reminder` events `e2` associated with `o1`.

We will later precisely define how object-centric querying and constraints according to our approach are constructed, but the basic idea revolves along the nested querying segments presented above. Before that, we need to introduce some basic concepts that allow forging the mathematical formulation of the nested queries and constraints in a general framework.

The first sections of this chapter presents our approach formally, addressing **RQ2** and **RQ3** as well as the research goal **RG3** and the corresponding contribution **CT2**. The approach formalization consists of three main parts:

1. **Variable Bindings** We first introduce the concept of variable bindings, which represent a mapping of variable names to actual objects and events of an OCED. They make it possible

to discuss *sets of bindings* fulfilling certain predicates.

2. **Nested Querying of Bindings** The querying and quantification of objects and events is conceptualized as a *binding box*, which constructs variable bindings, given variable names and the corresponding object or event types these variables should have, as well as additional filter predicates (e.g., that certain variable values should have a relation in the OCED). Next, to enable *nested* querying of bindings, we introduce the concept of *query trees*.

3. **Adding Constraints** Expanding on the process querying foundation, we add the ability to define allowed and disallowed behavior. In particular, we introduce labeling which determines, for each output binding of a query, if it is to be considered *satisfied* or *unsatisfied*. The labeling function can also be based on the child nodes of a node in the tree, for instance, requiring that at least one child binding in the nested query exists, like in Example 4.1.

To sum up, we start by formalizing variable bindings, binding boxes and other concepts of the presented object-centric process querying approach. Based on these concepts, we continue with building a constraint system based on the querying concepts. Additionally, after we presented the full formalization of our approach formally, we also detail additional aspects of our approach. In particular, we cover:

4. **Efficient Evaluation** Our proposed query and constraint approach is declarative. For efficiently computing the results of a query or constraint, however, an algorithmic formulation of the declarative approach needs to be constructed. To address this, we present a recursive algorithm which can compute the query and constraint results. Additionally, we focus on specific subroutines of this algorithm which implement the declarative approach of binding boxes efficiently. This section addresses **RQ4** and the research goal **RG5**, as well as contribution **CT3**.

5. **Automatic Constraint Discovery** Discovering constraints in our proposed declarative approach is a very interesting problem. It is also very challenging, as the high expressiveness permits many types of constraints. We describe two basic constraint types that can be discovered based on an input OCED, which are both highly relevant in practical processes and applications. Additionally, we also describe how generally, simple constraints can be combined to form more complex constraints, and demonstrate this by presenting the automatic discovery of OR-constructs of constraints. This section addresses research question **RQ5** and encompasses **RG6**, corresponding to part of our contribution **CT4**.

6. **Extensions** We also present extensions of our approach, specifically allowing for more expressive predicates in practice and general annotations of output bindings. For that, an expression programming language is used, which can be used to write simple yet very powerful predicates. Finally, we also mention how this extension can be advanced further, to allow annotating output bindings not only with a Boolean violation status but any general value. This also enables the computation and tracking of *Key Performance Indicators*.

4.1 Bindings and Binding Predicates

We first introduce the concept of *variable bindings.* A variable binding is a single instantiation of concrete objects and events of the OCED, which are referred to using *variable names.* In a natural language constraint like "*For all orders* $\texttt{o}$, *there is exactly one 'pay order' event* $\texttt{e}$ *associated with* $\texttt{o}$", the latter part (after the comma) references a concrete binding of $\texttt{o}$ and can only be evaluated for an actual object of an OCED (bound by the first part).

> **Definition 4.1 (Variable Binding):** Let L be an OCED. In the context of L, we define the set of variable bindings $\mathbb{B}_L$ as:
>
> $$\mathbb{B}_L = \{b_1 \cup b_2 \mid b_1 \in (\mathcal{U}_{\text{evVar}} \nrightarrow E_L) \land b_2 \in (\mathcal{U}_{\text{obVar}} \nrightarrow O_L)\}$$

Bindings make the implicit construct of a variable name referring to a variable value explicit. The universe of event and object variable names are disjoint (i.e., $\mathcal{U}_{\text{evVar}} \cap \mathcal{U}_{obVar} = \emptyset$). Thus, for an OCED L, a binding $b \in \mathbb{B}_L$ and an object variable $v \in \mathcal{U}_{obVar}$ either $b(v) = \perp$ or $b(v) \in O_L$. In Example 4.2, we present a few example bindings.

> **Example 4.2 (Variable Binding):** Let L be an OCED with the objects $o_1, o_2, o_3 \in O_L$ and events $e_1, e_2, e_3 \in E_L$, and let $\texttt{o1}, \texttt{o2}, \texttt{o3} \in \mathcal{U}_{\text{obVar}}$ and $\texttt{e1}, \texttt{e2}, \texttt{e3} \in \mathcal{U}_{\text{evVar}}$ be object and event variable names. Then the following are examples for bindings over L:
>
> - $b_1 = \{\}$
> - $b_2 = \{\texttt{o1} \mapsto o_1\}$
> - $b_3 = \{\texttt{e1} \mapsto e_1, \texttt{e2} \mapsto e_3, \texttt{e3} \mapsto e_2, \texttt{o2} \mapsto o_1, \texttt{o3} \mapsto o_3\}$

In the context of an OCED, we can also introduce the concept of child and parent bindings, where a child binding contains all the object and event variables of the parent, mapped to the same objects and events of the OCED, respectively, but can additionally introduce new object or event variables. Child bindings can thus be understood as *expansions* of parent bindings.

> **Definition 4.2 (Parent and Child Bindings $\sqsubseteq_L$):** Let L be an OCED. We define a parent-child relation $\sqsubseteq_L$ between bindings over L. For two bindings, $a \in \mathbb{B}_L$ and $b \in \mathbb{B}_L$, we define it as follows: $a \sqsubseteq_L b \Leftrightarrow \forall_{x \in \text{dom}(a)}\ a(x) = b(x)$.
> When $a \sqsubseteq_L b$, we call a a *parent binding* of b and b a *child binding* of a.
> Clearly, $\sqsubseteq_L$ is a partial order (i.e., reflexive, antisymmetric, and transitive).

The $\sqsubseteq_L$ relation brings some structure to the set of bindings $\mathbb{B}_L$. For every OCED L, the empty binding $\{\}$ is clearly the smallest element in $\mathbb{B}_L$ regarding $\sqsubseteq_L$. In Example 4.3, we showcase a few example bindings and their parent-child-relationships.

> **Example 4.3 (Parent and Child Bindings):** Consider the OCED L and the bindings b_1, b_2, b_3 from Example 4.2. It holds, that $b_1 \sqsubseteq_L b_2$ and $b_1 \sqsubseteq_L b_3$. But $b_2 \not\sqsubseteq_L b_3$, as $\texttt{o1}$ is not bound to a value in b_3. If we additionally consider $b_4 = \{\texttt{e2} \mapsto e_3, \texttt{o2} \mapsto o_1, \texttt{o3} \mapsto o_3\}$, it holds that $b_4 \sqsubseteq_L b_3$.

Occasionally, we want to consider only a subset of the variables involved in a binding. This is encompassed by restricting bindings to this subset of event and object variables.

> **Definition 4.3 (Restriction of Bindings):** Let L be an OCED. Let $b \in \mathbb{B}_L$ be a binding. Moreover, let $X \subseteq (\mathcal{U}_{\text{evVar}} \cup \mathcal{U}_{\text{obVar}})$ be a set of variable names. The restriction of b on X is then $b|_X \in \mathbb{B}_L$ with:
>
> $$b|_X(x) = \begin{cases} b(x), & \text{if } x \in X \\ \perp, & \text{otherwise} \end{cases}$$

The restriction of a binding simply only considers a subset of the variables bound by this binding. Example 4.4 demonstrates the restriction of an example binding.

Example 4.4 (Restriction of Bindings): Consider an example OCED L and the binding $b_4 \in \mathbb{B}_L$ considered before, where $b_4 = \{\texttt{e2} \mapsto e_3, \texttt{o2} \mapsto o_1, \texttt{o3} \mapsto o_3\}$. The restriction of b_4 to only the variables $\{\texttt{e2}, \texttt{o2}\}$ is then $b_4|_{\{\texttt{e2},\texttt{o2}\}} = \{\texttt{e2} \mapsto e_3, \texttt{o2} \mapsto o_1\}$.

As we show in Lemma 4.1, the restriction of a binding b is always a parent binding of b, as the remaining variables are mapped to the same values.

Lemma 4.1 (Restricted Binding is a Parent Binding): Let L be an OCED and let $b \in \mathbb{B}_L$ be a binding over L. Additionally, let $X \subseteq (\mathcal{U}_{\text{evVar}} \cup \mathcal{U}_{obVar})$ be a set of event and object variable names. Then the restricted binding $b|_X$ is a parent binding of b, i.e., $b|_X \sqsubseteq_L b$.

Proof. Let $b' = b|_X$. We want to show $b' \sqsubseteq_L b$. It holds that $\text{dom}(b') = \text{dom}(b) \cap X$ per Definition 4.3, and thus also $\text{dom}(b') \subseteq \text{dom}(b)$ and $\text{dom}(b') \subseteq X$. Then, for every $x \in \text{dom}(b')$, by Definition 4.3 it also holds that $b'(x) = b(x)$. □

Now that we have built a solid foundation of the concepts of bindings, we next aim to allow constructing a set of bindings. For that, we first introduce a way to specify criteria for bindings that enable filtering a set of bindings down, e.g., based on event-to-object or object-to-object relationships of the underlying values.

Definition 4.4 (Binding Predicates): Let L be an OCED. Given L, a *binding predicate* describes a set of bindings which satisfy this predicate. We write $\mathbb{P}_L$ for the set of all binding predicates under L, which we do not specify further. Notably, we only specify two important aspects of a predicate $s \in \mathbb{P}_L$: 1) A predicate induces a set of bindings that are satisfied for it and 2) A predicate has a certain identifier or type, which allows differentiating predicates even if they induce the same set of bindings. For a predicate statement $s \in \mathbb{P}_L$ we use the $\models$ relation to indicate that a binding is satisfied for this predicate. If a binding $b \in \mathbb{B}_L$ satisfies a predicate $s \in \mathbb{P}_L$, we write $b \models s$. Additionally, we use the same notation for a set of predicates $S \subseteq \mathbb{P}_L$: $b \models S \Leftrightarrow \forall_{s \in S}\ b \models s$.

We specify *collections of binding predicates* for different purposes. For instance, we will next define a predicate collection containing predicates based on object-to-object and event-to-object relationships in an OCED. Afterwards, we also introduce a predicate collection for other data attributes of events and object, like the total amount of an order.

First, however, consider a single example binding predicate $s \in \mathbb{P}_L$ with $b \models s \Leftrightarrow \in \mathbb{B}_L \mid b(\texttt{o1}) \in O_L \wedge b(\texttt{e1}) \in E_L \wedge b(\texttt{o1}) \in obj_L(b(\texttt{e1}))$. This predicate is satisfied for all variable bindings where the variables `o1` and `e1` are bound to objects or events of L, respectively, such that the values are in an event-to-object relationship.

We introduce the predicate collection $\mathbf{BASIC_L}$ in the following, and will later define additional collections. We assume that all introduced predicate collections are pairwise disjoint, even if some contained predicates might induce the same set of bindings for some OCED.

Definition 4.5 (Basic Collection of Predicates $\mathbf{BASIC_L}$): Initially, we define the binding collection $\mathbf{BASIC_L} \subseteq \mathbb{P}_L$ in the context of an OCED L. It contains some basic predicates based on object-to-object and event-to-object relationships in L and the time between events in L. Recall the shorthand notations from Note 3.1 on page 20, as they are extensively used in the predicate definitions. $\mathbf{BASIC_L}$ is made up by the following three predicate types:

- **Event-to-Object Relationship**: For an event variable $v \in \mathcal{U}_{\text{evVar}}$, an object variable $v' \in \mathcal{U}_{\text{obVar}}$ and an optional relationship qualifier $q \in \mathcal{U}_{qual} \cup \{*\}$, there is a predicate $\text{E2O}(v, v', q) \in \mathbf{BASIC_L}$, with for any $b \in \mathbb{B}_L$:

$$b \models \text{E2O}(v, v', q) \;\Leftrightarrow\; b(v) \in E_L \wedge b(v') \in O_L \wedge b(v') \in obj_L^q(b(v))$$

- **Object-to-Object Relationship**: For two object variables $v, v' \in \mathcal{U}_{\text{obVar}}$, and an optional qualifier $q \in \mathcal{U}_{qual} \cup \{*\}$, there is $\text{O2O}(v, v', q) \in \mathbf{BASIC_L}$, with for any $b \in \mathbb{B}_L$:

$$b \models \text{O2O}(v, v', q) \iff b(v) \in O_L \land b(v') \in O_L \land b(v') \in obj_L^q(b(v))$$

- **Time between Events**: For two event variables $v, v' \in \mathcal{U}_{\text{evVar}}$ and a duration interval $t_{min}, t_{max} \in \mathbb{T}$, there is $\text{TBE}(v, v', t_{min}, t_{max}) \in \mathbf{BASIC_L}$, with for any $b \in \mathbb{B}_L$:

$$b \models \text{TBE}(v, v', t_{min}, t_{max}) \iff b(v) \in E_L \land b(v') \in E_L \\ \land\ t_{min} \leq time_L(b(v')) - time_L(b(v)) \leq t_{max}$$

We then refer to a binding predicate $s \in \mathbf{BASIC_L}$ using the corresponding shorthand (e.g., $\text{E2O}(\texttt{e1}, \texttt{o1}, \texttt{order})$).

The predicates in the $\mathbf{BASIC_L}$ collection can be combined to model complex relationships involving multiple objects or events. However, let us first take a look at a simple example.

Example 4.5 (Simple Binding Predicate): Let L be an example OCED with $o_1, o_2, o_3 \in O_L$ and $e_1 \in E_L$. Consider the predicates $s_1, s_2 \in \mathbf{BASIC_L}$, with $s_1 = \text{O2O}(\texttt{o1}, \texttt{o2}, *)$ and $s_1 = \text{E2O}(\texttt{e1}, \texttt{o1}, *)$. Additionally, consider the bindings $b_1 = \{\texttt{o1} \mapsto o_1\}$, $b_2 = \{\texttt{o1} \mapsto o_1, \texttt{o2} \mapsto o_2, \texttt{e1} \mapsto e_1\}$ and $b_3 = \{\texttt{o1} \mapsto o_3, \texttt{o2} \mapsto o_4, \texttt{e1} \mapsto e_1\}$. As b_1 does not assign the variables o2 or e1, it holds that $b_1 \not\models s_1$ and $b_1 \not\models s_2$, respectively. Assuming an object-to-object relation between o_1 and o_2 exists in L, $b_2 \models s_1$ would hold. The same applies regarding o_3 and o_4 for b_3. Similarly, with knowledge about the objects involved in e_1, we could determine if $b_2 \models s_2$ and $b_3 \models s_2$ holds.

We also introduce a predicate collection for *data attribute predicates*, which allow formulating statements regarding general attributes of events or objects. For instance, in an order management process, order objects might have an attribute price, referring to its total price.

Definition 4.6 (Data Attribute Predicates): Let $L = (E, O, eaval, oaval)$ be an OCED. We define a predicate collection for *data attribute filters* as $\mathbf{ATTRS}_L \subseteq \mathbb{P}_L$, with filter predicates for the attribute values of events and objects.

- **Event Data Attribute Equality**: For an event variable $en \in \mathcal{U}_{\text{evVar}}$, an attribute name $\texttt{att} \in \mathcal{U}_{attr}$ and a value $v \in \mathbb{R} \cup \mathbb{T} \cup \mathcal{U}_\Sigma$, there is $\text{EAE}(en, \texttt{att}, v) \in \mathbf{ATTRS}_L$ with for any $b \in \mathbb{B}_L$: $b \models \text{EAE}(en, \texttt{att}, v) \iff b(en) \in E \land eaval_{b(en)}(\texttt{att}) = v$.
- **Event Data Attribute Range**: For an event variable $en \in \mathcal{U}_{\text{evVar}}$, an attribute name $\texttt{att} \in \mathcal{U}_{attr}$ and two values $v_{min}, v_{max} \in \mathbb{R} \cup \{-\infty, \infty\}$ or $v_{min}, v_{max} \in \mathbb{T}^\infty$, there is $\text{EAR}(en, \texttt{att}, v_{min}, v_{max}) \in \mathbf{ATTRS}_L$ with $b \models \text{EAR}(en, \texttt{att}, v_{min}, v_{max}) \iff b(en) \in E \land eaval_{b(en)}(\texttt{att}) \in \mathbb{R} \cup \mathbb{T} \land v_{min} \leq eaval_{b(en)}(\texttt{att}) \leq v_{max}$.
- **Object Data Attribute Equality**: For an object variable $on \in \mathcal{U}_{obVar}$, an attribute name $\texttt{att} \in \mathcal{U}_{attr}$, a value $v \in \mathbb{R} \cup \mathbb{T} \cup \mathcal{U}_\Sigma$ and a time-specifier $T \in \{\texttt{ALWAYS}, \texttt{SOMETIME}\} \cup \mathcal{U}_{\text{evVar}}$, there is $\text{OAE}(on, \texttt{att}, v, T) \in \mathbf{ATTRS}_L$ with:
  - For $T = \texttt{ALWAYS}$ with for all $b \in \mathbb{B}_L$:

$$b \models \text{OAE}(on, \texttt{att}, v, T) \iff b(on) \in O \\ \land\ \forall_{t \in \mathbb{T}} \left(oaval_{b(on)}^t(\texttt{att}) = \bot \lor oaval_{b(on)}^t(\texttt{att}) = v\right)$$

  - For $T = \texttt{SOMETIME}$ with for all $b \in \mathbb{B}_L$::

$$b \models \text{OAE}(on, \texttt{att}, v, T) \iff b(on) \in O \land \exists_{t \in \mathbb{T}}\ oaval_{b(on)}^t(\texttt{att}) = v$$

  - For $T \in \mathcal{U}_{\text{evVar}}$ with for all $b \in \mathbb{B}_L$:

$$b \models \text{OAE}(on, \texttt{att}, v, T) \Leftrightarrow b(on) \in O \wedge b(T) \in E \wedge oaval^{t}_{b(on)}(\texttt{att}) = v$$
$$\text{where, for } b(T) \in E \text{, we use } t = time_L(b(T))$$

- **Object Data Attribute Range**: Similar to the event data attribute range predicate types, we also allow object data attribute range predicates. For an object variable $on \in \mathcal{U}_{obVar}$, an attribute name $\texttt{att} \in \mathcal{U}_{attr}$ and two values $v_{min}, v_{max} \in \mathbb{R} \cup \{-\infty, \infty\}$ or $v_{min}, v_{max} \in \mathbb{T}^{\infty}$ as well as a time-specifier $T \in \{\texttt{ALWAYS}, \texttt{SOMETIME}\} \cup \mathcal{U}_{\text{evVar}}$, there is $\text{OAR}(on, \texttt{att}, v_{min}, v_{max}, T) \in \textbf{ATTRS}_L$. We omit a formal definition for brevity, as it is simply a combination of the previously presented data attribute predicates (i.e., OAE adapted to range values, like in EAR).

Data attribute predicates allow considering event and object attributes beyond the predefined standard attributes, like `type` or `objects`. In Example 4.6, we present how these data attribute predicates can be used.

Example 4.6 (Data Attribute Predicates): Consider an order management OCED L, where all events have an additional attribute `location`, specifying in which office location an event happened. The predicate $\text{EAE}(\texttt{e1}, \texttt{location}, \texttt{Aachen})$ is then satisfied for all bindings $b \in \mathbb{B}_L$, where the event variable `e1` is bound to an event value (i.e., $b(\texttt{e1}) \in E_L$), which happened in the office in Aachen (i.e., $eaval_{b(\texttt{e1})}(\texttt{location}) = \texttt{Aachen}$).
Object attributes are more complex to handle, as they are associated with a specific moment in time. For instance, an object $o_1 \in O_L$ of type `product` might have different prices (represented by the attribute name `price`) at different timestamps: Initially it costs 50€, but the price is increased to 75€ after a few months. Given the binding $b = \{\texttt{o1} \mapsto o_1\}$, the predicate $\text{OAR}(\texttt{o1}, \texttt{price}, 70, \infty, \texttt{ALWAYS})$ is then not satisfied by b, as the price is not *always* above 70€. The predicate $\text{OAR}(\texttt{o1}, \texttt{price}, 70, \infty, \texttt{SOMETIME})$, however, is satisfied by b, as the price is above 70€ *at some point in time.*

4.2 Nested Querying of Bindings using Binding Boxes

Next, we introduce the concept of *binding boxes*, which encompass simple querying (i.e., quantification and filtering) of bindings. For the quantification, it specifies a set of event and object variables that should be assigned to concrete values, as well as the event or object types these values should have. For filtering, a set of binding predicates specify if a binding should be part of the output of a binding box or not. As an example, a binding box could specify, that the event variable `e1` should be bound to events with the activity `pay order` and that only such bindings that additionally satisfy the data predicate $\mathrm{EAE}(\texttt{e1}, \texttt{location}, \texttt{Aachen})$ are of interest.

The allowed event or object types for quantification are specified as a set to allow for maximal flexibility. This enables, for instance, specifying that the event variable `e1` should be bound to events with the activity `pay order` or `cancel order` for queries where the actual outcome of an order is irrelevant. Next, in Definition 4.7, we formally define a binding box. For simplicity, we initially do not restrict the set of binding predicate a binding box can contain, and will specify such restrictions later on when needed.

Definition 4.7 (Binding Box): Let L be an OCED. A binding box $\mathfrak{b}_L = (\mathrm{Var}, \mathrm{Pred})$ over L is a tuple consisting of:

- $\mathrm{Var} \in \{ev \cup ob \mid ev \in \mathcal{U}_{\mathrm{evVar}} \nrightarrow \mathcal{P}(\mathcal{U}_{etype}) \wedge ob \in \mathcal{U}_{obVar} \nrightarrow \mathcal{P}(\mathcal{U}_{otype})\}$, a partial function which specifies to values of which event or object types selected variables should be bound[a].
- $\mathrm{Pred} \subseteq \mathbb{P}_L$, a set of binding predicates for filtering.

Intuitively, $\mathfrak{b}_L$ binds the variable names $\mathrm{dom}(\mathrm{Var})$ to all combination of values (i.e., events or objects of L) where the predicate set Pred holds. For convenience, we sometimes write $\mathrm{Var}(\mathfrak{b}_L) = \mathrm{Var}$ and $\mathrm{Pred}(\mathfrak{b}_L) = \mathrm{Pred}$. We define when a binding $b \in \mathbb{B}_L$ satisfies the binding box, written as $b \models \mathfrak{b}_L$, as follows:

$$\begin{aligned} b \models \mathfrak{b}_L \iff & b \models \mathrm{Pred} \wedge \mathrm{dom}(b) = \mathrm{dom}(\mathrm{Var}) \\ & \wedge \forall_{v \in \mathrm{dom}(\mathrm{Var})} \big(b(v) \in E_L \cup O_L \wedge type_L(b(v)) \in \mathrm{Var}(v)\big) \end{aligned}$$

Put into words, a binding box is satisfied for a binding if: 1) The binding is satisfied for the predicates and, 2) the binding assigns exactly the variables specified by the binding box, and 3) all assigned variable values in the binding have the correct types as specified by the binding box.
We write $\mathfrak{BOX}_L$ for the set of all binding boxes under L.

[a] In particular, formally, Var is a partial function of the type $\mathrm{Var}\colon (\mathcal{U}_{\mathrm{evVar}} \cup \mathcal{U}_{obVar}) \nrightarrow (\mathcal{P}(\mathcal{U}_{etype}) \cup \mathcal{P}(\mathcal{U}_{otype}))$, with the additional restriction that event variables are only mapped to sets of event types and object variables are only mapped to sets of object types.

We first present a simple example binding box in Example 4.7. Of course, binding boxes can also be more complex and can contain multiple variables and predicates. In general, a binding box can contain arbitrary filter predicates. However, for now, we only consider binding boxes $\mathfrak{b}_L = (\mathrm{Var}, \mathrm{Pred})$ where $\mathrm{Pred} \subseteq \mathbf{BASIC_L}$. Later on, we also introduce other predicate types for which this general definition of binding boxes is convenient.

Example 4.7 (Binding Box): Let $L = (E, O, eaval, oaval)$ be an OCED of an order-to-cash process, which is not fully defined here for brevity. Assume, however, that at least $\{\texttt{orders}\} \subseteq \mathcal{U}_{otype}$ and $\{\texttt{place order}, \texttt{confirm order}\} \subseteq \mathcal{U}_{etype}$.
Consider the simple binding box $\mathfrak{a}_L = (\mathrm{Var}, \mathrm{Pred})$ with:

- $\mathrm{Var} = \{\texttt{e1} \mapsto \{\texttt{place order}, \texttt{confirm order}\}, \texttt{o1} \mapsto \{\texttt{orders}\}\}$

- $\mathrm{Pred} = \{\mathrm{E2O}(\texttt{e1}, \texttt{o2}, \texttt{order})\}$

Filter predicates are also particularly useful to represent a binding box visually, as the very formal representation of binding box examples can quickly become cumbersome to read. Instead, we will use an alternative Z-notation style schema (cf. [37]) in further examples. For a binding box $\mathfrak{a}_L = (\mathrm{Var}, \mathrm{Pred})$, the elements of Var are split into multiple lines on the top. On the bottom (i.e., below the horizontal line), the filter predicates Pred are listed using their shorthand notation.

Example 4.8 (Visual Notation for Binding Box): Consider L and $\mathfrak{a}_L$ from Example 4.7. In the alternative notation style, $\mathfrak{a}_L$ can be represented like this:

$\mathfrak{a}_L$

$\texttt{o1} : \textsc{Object}(\texttt{orders})$

$\texttt{e1} : \textsc{Event}(\texttt{place order}, \texttt{confirm order})$

$\mathrm{E2O}(\texttt{e1}, \texttt{o1}, \texttt{order})$

Next, we formally define the output binding set $out_L(\mathfrak{b}_L)$ of a binding box $\mathfrak{b}_L$.

Definition 4.8 (Binding Box Output): Let L be an OCED. Let $\mathfrak{b}_L$ be a binding box over L. The *output* of $\mathfrak{b}_L$ is simply the set of bindings that satisfy $\mathfrak{b}_L$.

$$out_L(\mathfrak{b}_L) = \{b \in \mathbb{B}_L \mid b \models \mathfrak{b}_L\}$$

As mentioned before, the output binding set contains all bindings assigning object and event variables to instances of the specified type that additionally fulfill the filter predicates Pred. Next, we give an example binding box output set.

Example 4.9 (Binding Box Output): Consider the OCED L and the binding box $\mathfrak{a}_L$ from Example 4.7 and Example 4.8. Assume that L contains only the objects o_1, o_2, o_3 of type `orders`, where all of them except o_3 are associated with a `place order` event (i.e., with e_1 and e_2, respectively). Moreover, only o_1 is additionally associated with a `confirm order` event e_3. Under these assumptions, we can specify $out_L(\mathfrak{a}_L)$ as:

$$out_L(\mathfrak{a}_L) = \Big\{ \{\texttt{o1} \mapsto o_1, \texttt{e1} \mapsto e_1\}, \{\texttt{o1} \mapsto o_2, \texttt{e1} \mapsto e_2\}, \{\texttt{o1} \mapsto o_1, \texttt{e1} \mapsto e_3\} \Big\}$$

As all output binding set of a binding box share the same variables, the set can also be visualized as a table. The columns of the table then correspond to the variable names, and for each output binding there is one row in the table.

Example 4.10 (Binding Box Output Table): Consider the OCED L, the binding box $\mathfrak{a}_L$ and the corresponding output set given in Example 4.9. Below, the output set of $\mathfrak{a}_L$ is visualized as a table, where each row corresponds to one of the output bindings.

o1	e1
o_1	e_1
o_2	e_2
o_1	e_3

Next, we define a relation $\preceq_L$ between binding boxes over an OCED L. This relation encompasses the concept of *refined* binding boxes, where new object or event variables are (optionally) introduced, and the constraint filter is at least as strict as before.

Definition 4.9 ($\preceq_L$ Between Binding Boxes): Let L be an OCED. We define the relation $\preceq_L$ between binding boxes over L. For two binding boxes over L, $\mathfrak{a}_L, \mathfrak{b}_L \in \mathfrak{BOX}_L$, we say $\mathfrak{a}_L \preceq_L \mathfrak{b}_L$ holds if and only if:

- $\mathrm{Var}(\mathfrak{a}_L) \subseteq \mathrm{Var}(\mathfrak{b}_L)$

- $\text{Pred}(\mathfrak{a}_L) \subseteq \text{Pred}(\mathfrak{b}_L)$

We also call $\mathfrak{a}_L$ a parent binding box of $\mathfrak{b}_L$ and $\mathfrak{b}_L$ a child binding box of $\mathfrak{a}_L$.

Later, we will show how this relation can be used to construct trees of binding boxes. The core idea revolves around the following observations: 1) The variables introduced in the parent binding box are all also present in the child binding box. 2) All predicates of the parent are also present in the child binding box. In particular, this implies that projecting the output set of the child binding box on just the variables already introduced in the parent binding box is a subset of the output set of the parent.

However, first consider Example 4.11, demonstrating the $\preceq_L$ relation between two binding boxes.

Example 4.11 ($\preceq_L$ Between Binding Boxes): Consider the OCED L and the example binding box $\mathfrak{a}_L$ from Example 4.7 and a second binding box $\mathfrak{b}_L = (\{\texttt{o1} \to \{\texttt{orders}\}\}, \emptyset)$. Then, it holds that $\mathfrak{b}_L \preceq_L \mathfrak{a}_L$. Additionally, if we consider an additional third binding box $\mathfrak{c}_L = (\{\texttt{o1} \to \{\texttt{orders}\}\}, \{\text{OAE}(\texttt{o1}, \texttt{price}, 100, \texttt{ALWAYS})\})$, it additionally holds that $\mathfrak{b}_L \preceq_L \mathfrak{c}_L$ but $\mathfrak{a}_L \preceq_L \mathfrak{c}_L$ and $\mathfrak{c}_L \preceq_L \mathfrak{a}_L$.

Next, we define the restriction of binding box predicates on a subset of predicate types.

Definition 4.10 (Restricted Binding Box): Let L be an OCED and let $\mathfrak{a} = (\text{Var}, S) \in \mathfrak{BOX}_L$ be a binding box over L. Additionally, let $X \subseteq \mathbb{P}_L$ be a set of binding predicates. The filter-restriction of $\mathfrak{a}$ to X, denoted as $\mathfrak{a}|_X$, is the binding box $\mathfrak{a}|_X = (\text{Var}, S \cap X)$ over L.

The restriction of binding boxes enables us to only consider a subset of predicates, for instance only predicates of a specified type, for certain restrictions and definitions. Consider, for example, Example 4.12, which builds the restriction on only basic predicates (i.e., $\mathbf{BASIC_L}$).

Example 4.12 (Restricted Binding Box): Consider the OCED L from the previous examples. Additionally, consider a binding box $\mathfrak{a}_L$ with:
$\mathfrak{a}_L = (\{\texttt{o1} \to \{\texttt{orders}\}, \texttt{e1} \to \{\texttt{pay order}\}\}, \{\text{E2O}(\texttt{e1}, \texttt{o1}, *), \text{OAE}(\texttt{o1}, \texttt{price}, 100, \texttt{ALWAYS})\})$.
The restriction of $\mathfrak{a}_L$ only considering predicates from $\mathbf{BASIC_L}$ is then $\mathfrak{a}_L|_{\mathbf{BASIC_L}}$ with:
$\mathfrak{a}_L|_{\mathbf{BASIC_L}} = (\{\texttt{o1} \to \{\texttt{orders}\}, \texttt{e1} \to \{\texttt{pay order}\}\}, \{\text{E2O}(\texttt{e1}, \texttt{o1}, *)\})$.

So far, we introduced binding boxes and the $\preceq_L$ relation between them. Next, we introduce the last important concept for our process querying approach: *query trees*. A query tree is a rooted, directed tree containing nodes corresponding to binding boxes, where an edge between two nodes corresponding to binding boxes $\mathfrak{a}$ and $\mathfrak{b}$ implies that $\mathfrak{a}|_X \preceq_L \mathfrak{b}|_X$ for $X = \mathbf{BASIC_L} \cup \mathbf{ATTRS}_L$. In other words, the binding boxes get more *refined* per tree layer when only considering basic and attribute predicate filters. The reason for only considering those predicate collections is that the other predicate collections we will define later are based on the query tree structure itself, and in particular consider the child nodes of a given node. As such, these types of predicates should not be added to the binding box of child nodes themselves.

Definition 4.11 (Query Tree): Let L be an OCED. A query tree over L is a tuple $T = (V, F, r, l, \mathfrak{box})$, where:

- V is a finite set of nodes.
- $r \in V$ is the designated root node (i.e., the only node with no parent).
- $F \subseteq V \times V$ is a set of edges between nodes, such that in the directed graph (V, F) there is exactly one path from r to a for all $a \in V$ (i.e., (V, F) is a rooted tree).
- $l \colon F \to \mathcal{U}_{\text{setName}}$ is an injective function, assigning unique names to edges in F.
- $\mathfrak{box} \colon V \to \mathfrak{BOX}_L$ is a function which maps each node in V to a binding box over L, such that the following properties hold:
  - For all nodes $u \in V$, $\text{Pred}(\mathfrak{box}(u)) \subseteq \mathbf{BASIC_L} \cup \mathbf{ATTRS}_L \cup \mathbf{CHILD\ SET}_u^T$, where $\mathbf{CHILD\ SET}_u^T$ is a new collection of predicates we will introduce next.

- For all edges $(a, b) \in F$ with $\mathfrak{box}(a) = \mathfrak{a}$ and $\mathfrak{box}(b) = \mathfrak{b}$, it holds that $\mathfrak{a}|_{\mathbf{BASIC_L} \cup \mathbf{ATTRS}_L} \preceq_L \mathfrak{b}|_{\mathbf{BASIC_L} \cup \mathbf{ATTRS}_L}$.

As each node of the query tree is mapped to a binding box, we sometimes use the name of the node to refer to the corresponding binding box in text (i.e., v_0 instead of $\mathfrak{box}(v_0)$). Before we introduce the predicate collections $\mathbf{CHILD\ SET}_u^T$ formally, we first present an example query tree in Example 4.13 using only $\mathbf{BASIC_L}$ predicates. This query tree queries all confirmed orders and also contains subqueries for A) all `pay order` events related to the order occurring within 4 weeks of the order confirmation and B) all `payment reminder` events related to the order.

Example 4.13 (Query Tree): Consider the OCED L from Example 4.7. Additionally, consider the query tree $T_1 = (V, F, r, l, \mathfrak{box})_L$ with $V = \{v_0, v_1, v_2\}$, $r = v_0$ and $F = \{(v_0, v_1), (v_0, v_2)\}$, such that $l((v_0, v_1)) = \mathtt{A}$ and $l((v_0, v_2)) = \mathtt{B}$. The graph of T_1 is shown below on the top right annotated with exemplary output binding tables, while $\mathfrak{box}$ is presented on the left. When showing examples of a query tree, the binding boxes naturally contain many duplicates (highlighted in gray below). To ease readability, we often only present the *additions* between binding boxes and their parents. In particular, we omit variables and basic or data predicates that are already present in the binding box of the parent node. With these omissions (which we mark using Δ), $\mathfrak{box}(v_1)$ and $\mathfrak{box}(v_2)$ can be presented more compactly, as shown below on the bottom right.

$\mathfrak{box}(v_0)$
o1 : Object(orders)
e1 : Event(confirm order)

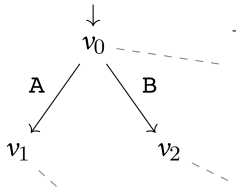
E2O(e1, o1, $*$)

$\mathfrak{box}(v_1)$
o1 : Object(orders)
e1 : Event(confirm order)
e2 : Event(pay order)

E2O(e1, o1, $*$)
E2O(e2, o1, $*$)
TBE(e1, e2, 0, 4w)

$\mathfrak{box}(v_2)$
o1 : Object(orders)
e1 : Event(confirm order)
e2 : Event(payment reminder)

E2O(e1, o1, $*$)
E2O(e2, o1, $*$)



o1	e1
o_1	e_1
o_2	e_2
o_3	e_3
o_4	e_4

o1	e1	e2
o_3	e_3	e_7
o_3	e_3	e_8

o1	e1	e2
o_1	e_1	e_5
o_2	e_2	e_6
o_3	e_3	e_9

$\Delta\mathfrak{box}(v_1)$
e2 : Event(pay order)

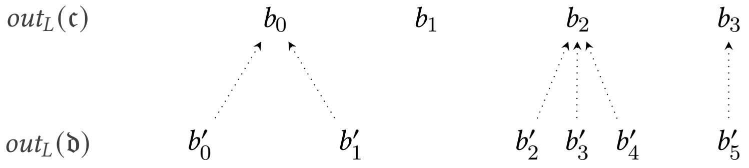
E2O(e2, o1, $*$)
TBE(e1, e2, 0, 4w)

$\Delta\mathfrak{box}(v_2)$
e2 : Event(payment reminder)

E2O(e2, o1, $*$)

On the top right, we show exemplary output sets of $\mathfrak{box}(v_0)$, $\mathfrak{box}(v_1)$ and $\mathfrak{box}(v_2)$ as tables next to the corresponding nodes. The rows for v_0 are colored in four different colors. For v_1 and v_2, each output row is colored in one of them, indicating from which parent binding in the output set of v_0 the row is derived. The first two output binding rows for v_0 (in cyan and magenta) have exactly one child binding in the output set of v_1 and none in the output set of v_2. For the third output binding row of v_0 (in teal), one child binding in v_1 exists, and there are also two child bindings in the output set of v_2. The last output row of v_0 (in orange) has no child binding in the output sets of v_1 or v_2.

We can also observe that the parent-child relation between output bindings is particularly interesting. Recall the definitions of the $\sqsubseteq_L$ and $\preceq_L$ relation introduced in Definition 4.2 and Definition 4.9. The $\preceq_L$ relation between binding boxes over L is closely related to the $\sqsubseteq_L$ relation on their output binding sets. Consider an OCED L and a query tree $(V, F, r, l, \mathfrak{box})$, as well as a specific edge $(c, d) \in F$. We write $\mathfrak{c} = \mathfrak{box}(c)|_{\mathbf{BASIC_L} \cup \mathbf{ATTRS}_L}$ and $\mathfrak{d} = \mathfrak{box}(d)|_{\mathbf{BASIC_L} \cup \mathbf{ATTRS}_L}$. Per definition, we then know that $\mathfrak{c} \preceq_L \mathfrak{d}$ holds. Below, we visually showcase exemplary output bindings $out_L(\mathfrak{c})$ and $out_L(\mathfrak{d})$ and the $\sqsubseteq_L$ relation between them. For each binding $b' \in out_L(\mathfrak{d})$, there is exactly one parent binding $b \in out_L(\mathfrak{c})$. Put in other words, for the parent bindings in $out_L(\mathfrak{c})$ the $\sqsubseteq_L$ relation *partitions* the elements of $out_L(\mathfrak{d})$ into sets of child bindings. We prove that this also holds in general, for any binding boxes $\mathfrak{a}, \mathfrak{b}$ with $\mathfrak{a} \preceq_L \mathfrak{b}$ in Lemma 4.2.



Lemma 4.2 (Unique Parent Binding in Parent Node Output): Let L be an OCED. Let $\mathfrak{a}_L$ and $\mathfrak{b}_L$ be binding boxes over L with $\mathfrak{a}_L \preceq_L \mathfrak{b}_L$. For any output binding $b \in out_L(\mathfrak{b}_L)$ of $\mathfrak{b}_L$, there is a unique $a \in out_L(\mathfrak{a}_L)$ with $a \sqsubseteq_L b$. In particular, a is the reduced binding $a = b|_{\mathrm{dom}(\mathrm{Var}(\mathfrak{a}_L))}$ of b. For brevity, we also write $b|_{\mathfrak{a}_L}$ for this reduced binding.

Proof. We first show that such an a exists and then show its uniqueness.

Existence As $\mathfrak{a}_L \preceq_L \mathfrak{b}_L$, we know that $b \models \mathfrak{a}_L$. Per definition of $\models$ for binding boxes, for the reduced binding $b' = b|_{\mathfrak{a}_L}$ it also holds that $b' \models \mathfrak{a}_L$, as the predicates of $\mathfrak{a}_L$ are a subset of those of $\mathfrak{b}_L$ and only filter based on the variables introduced by $\mathfrak{a}_L$ itself. And as b' only contains event and object variables introduced by $\mathfrak{a}_L$, clearly $b' \in out_L(\mathfrak{a}_L)$. Furthermore, $b' \sqsubseteq_L b$, per Lemma 4.1. ✓

Uniqueness Assume $a, a' \in out_L(\mathfrak{a}_L)$ with $a \sqsubseteq_L b$ and $a' \sqsubseteq_L b$. Clearly $\mathrm{dom}(a) = \mathrm{dom}(a')$, as both are in the output set of $\mathfrak{a}_L$. As $\mathfrak{a}_L \preceq_L \mathfrak{b}_L$, $\mathrm{Var}(\mathfrak{a}_L) \subseteq \mathrm{Var}(\mathfrak{b}_L)$ and thus $\mathrm{dom}(a) = \mathrm{dom}(a') \subseteq \mathrm{dom}(b)$. In particular, per definition of $\sqsubseteq_L$, it then holds that $\forall_{x \in \mathrm{dom}(a)}\ a(x) = a'(x) = b(x)$, or in other words: $a = a'$. ✓

□

In combination with these observations, the example query tree in Example 4.13 also motivates the need for the predicate collection $\mathbf{CHILD\ SET}_u^T$ for a node $u \in V$: Right now, the nested querying does not really have any use case. In the presented example, we would want, for instance, to identify bindings of $\mathfrak{box}(v_0)$ for which there is no child binding in the output of $\mathfrak{box}(v_1)$ and $\mathfrak{box}(v_2)$. Put into words, such bindings would correspond to all confirmed orders that were not paid within 4 weeks and also did not have a payment reminder sent. We introduce $\mathbf{CHILD\ SET}_u^T$, which allow specifying such predicates based on the set of child bindings in the context of a query tree. In these filter predicates, for an edge $(u, v) \in F$ the (unique) edge name $l((u, v))$ (e.g., $\mathtt{A}$) is used to conveniently reference the set of child bindings in v for a given output binding of u.

Definition 4.12 (Child Set Predicates $\mathbf{CHILD\ SET}_u^T$): Let L be an OCED and $T = (V, F, r, l, \mathfrak{box})$ a query tree over L. Given a node $u \in V$, the collection of set filter predicates $\mathbf{CHILD\ SET}_u^T$ available in $\mathfrak{box}(u)$ are:

- **Child Bindings Set Size**: For all $v \in V$ with $e = (u, v) \in F$ where $l(e) = \mathtt{A}$ as well as $c_{min}, c_{max} \in \mathbb{N}_0^\infty$, there is $\mathrm{CBS}(\mathtt{A}, c_{min}, c_{max}) \in \mathbf{CHILD\ SET}_u^T$, with for any $b \in out_L(\mathfrak{box}(u))$:

$$b \models \mathrm{CBS}(\mathtt{A}, c_{min}, c_{max}) \;\Leftrightarrow\; c_{min} \leq |\{b' \in out_L(\mathfrak{box}(v)) \mid b \sqsubseteq_L b'\}| \leq c_{max}$$

- **Projected Child Bindings Set Size**: For $v \in V$ where $e = (u, v) \in F$ holds with $l(e) = \mathtt{A}$ as well as $\mathtt{x} \in Ev(\mathfrak{box}(v)) \cup Ob(\mathfrak{box}(v))$ and $c_{min}, c_{max} \in \mathbb{N}_0^\infty$, there is $\mathrm{CBPS}(\mathtt{A}, \mathtt{x}, c_{min}, c_{max}) \in$ **CHILD SET**$_u^T$, with for any $b \in out_L(\mathfrak{box}(u))$:

$$b \models \mathrm{CBPS}(\mathtt{A}, \mathtt{x}, c_{min}, c_{max}) \;\Leftrightarrow\; c_{min} \leq |\{b'(\mathtt{x}) \mid b' \in out_L(\mathfrak{box}(v)) \wedge b \sqsubseteq_L b'\}| \leq c_{max}$$

- **Child Binding Sets Equal**: For $v, v' \in V$ where $e = (u, v) \in F$ and $e' = (u, v') \in F$ holds with $l(e) = \mathtt{A}$ and $l(e') = \mathtt{B}$, there is $\mathrm{CBE}(\mathtt{A}, \mathtt{B}) \in$ **CHILD SET**$_u^T$, with for any $b \in out_L(\mathfrak{box}(u))$:

$$b \models \mathrm{CBE}(\mathtt{A}, \mathtt{B}) \;\Leftrightarrow\; \{b' \in out_L(\mathfrak{box}(v)) \mid b \sqsubseteq_L b'\} = \{b' \in out_L(\mathfrak{box}(v')) \mid b \sqsubseteq_L b'\}$$

- **Projected Child Binding Sets Equal**: For $v, v' \in V$ where $e = (u, v) \in F$ and $e' = (u, v') \in F$ holds with $l(e) = \mathtt{A}$ and $l(e') = \mathtt{B}$ as well as $\mathtt{x} \in Ev(\mathfrak{box}(v)) \cup Ob(\mathfrak{box}(v))$ and $\mathtt{y} \in Ev(\mathfrak{box}(v')) \cup Ob(\mathfrak{box}(v'))$, there is $\mathrm{CBPE}(\mathtt{A}, \mathtt{x}, \mathtt{B}, \mathtt{y}) \in$ **CHILD SET**$_u^T$, with for any $b \in out_L(\mathfrak{box}(u))$:

$$\begin{aligned} b \models \mathrm{CBPE}(\mathtt{A}, \mathtt{x}, \mathtt{B}, \mathtt{y}) \;\Leftrightarrow\; & \{b'(\mathtt{x}) \mid b' \in out_L(\mathfrak{box}(v)) \wedge b \sqsubseteq_L b'\} \\ & = \{b'(\mathtt{x}) \mid b' \in out_L(\mathfrak{box}(v')) \wedge b \sqsubseteq_L b'\} \end{aligned}$$

Note, that the definition of **CHILD SET** is recursive, as the predicates for child nodes are assumed to be already defined when considering the parent node. However, as query trees are rooted, directed trees and thus in particular also acyclic, the semantics of **CHILD SET** are still well-defined. In particular, all predicates and output bindings can be considered from the bottom up, starting at the leaf nodes of the query tree. This recursion also indicates why these predicates should not be considered for the $\preceq_L$ relation in the tree. For more details, also see Section 4.4, where we describe how the output sets of query trees can be computed algorithmically.

In Example 4.14, filter predicates based on the number of child bindings are used.

Example 4.14 (Query Tree with Set Filters): Consider the OCED L from Example 4.7. Additionally, consider the count-filtered query tree $T_2 = (V, F, r, l, \mathfrak{box})$, with $V = \{v_0, v_1, v_2\}$, $r = v_0$ and $F = \{(v_0, v_1), (v_0, v_2)\}$. The graph (V, F) is visually represented on the right together with the edge label names assigned by l and exemplary output tables, while $\mathfrak{box}$ is indicated on the left. Overall, T_2 queries placed orders which *were not paid fast* (i.e., within 4 weeks after confirmation) and for which *also no payment reminder was sent.*

$\mathfrak{box}(v_0)$

o1 : Object(orders)
e1 : Event(confirm order)

E2O(e1, o1, order)
CBS(A, 0, 0) // ∈ **CHILD SET**$_{v_0}^{T_2}$
CBS(B, 0, 0) // ∈ **CHILD SET**$_{v_0}^{T_2}$

$\Delta\mathfrak{box}(v_1)$

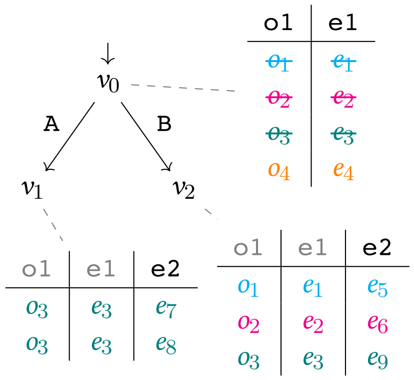
e2 : Event(pay order)

E2O(e2, o1, ∗)
TBE(e1, e2, 0, 4w)

$\Delta\mathfrak{box}(v_2)$

e2 : Event(payment reminder)

E2O(e2, o1, ∗)



o1	e1
~~o_1~~	~~e_1~~
~~o_2~~	~~e_2~~
~~o_3~~	~~e_3~~
o_4	e_4

o1	e1	e2
o_3	e_3	e_7
o_3	e_3	e_8

o1	e1	e2
o_1	e_1	e_5
o_2	e_2	e_6
o_3	e_3	e_9

Again, we annotate exemplary output tables for each node in the tree above. If bindings are removed only by a child set predicate of a binding box but fulfill the basic predicates of it (i.e., $\mathbf{BASIC_L}$ and $\mathbf{ATTRS}_L$), we indicate this by including this binding with a strikethrough. As per definition of query trees, children of the node might still contain child bindings for crossed out binding rows. For instance, the first (removed) output row for v_0 (in cyan) has a child binding in the output set of v_2. Note, however, that formally crossed-out binding rows are not part of $out_L(\mathfrak{box}(v_0))$, but only contained in $out_L(\mathfrak{box}(v_0)|_{\mathbf{BASIC_L} \cup \mathbf{ATTRS}_L})$.

Before continuing with the introduction of constraints, we first present a few ways the filters of a binding box (e.g., inside a query tree) can be quantified, for instance by considering the number of all possible bindings of the variables.

Definition 4.13 (Quantifying Filter Fractions): Let L be an OCED and $\mathfrak{b}_L = (\mathrm{Var}, \mathrm{Pred})$ a binding box over L. For a given event or object type, the number of instances of that type in the log is given by the function $\mathrm{NumInst}\colon \mathcal{P}(\mathcal{U}_{etype} \cup \mathcal{U}_{otype}) \rightarrow \mathbb{N}_0$ defined as $\mathrm{NumInst}(ts) = |\{x \in E_L \cup O_L \mid type_L(x) \in ts\}|$. The fraction of all possible bindings of the involved variables satisfying the predicates is then:

$$f = \frac{|out_L(\mathfrak{b}_L)|}{\prod_{v \in \mathrm{dom}(\mathrm{Var})} \mathrm{NumInst}\big(\mathrm{Var}(v)\big)}$$

For each object or event variable, one can also calculate what fraction of all its possible values are instantiated by a valid binding satisfying the filters of v. In particular, given an assigned variable $v \in \mathrm{dom}(\mathrm{Var})$ with $ts = \mathrm{Var}(v)$, this fraction is given as:

$$f_x = \frac{|\{b(x) \mid b \in out_L(\mathfrak{b}_L)\}|}{\mathrm{NumInst}(ts)}$$

All of these introduced fractions hold values between 0 and 1, as they are upper bounded exactly by the denominator.

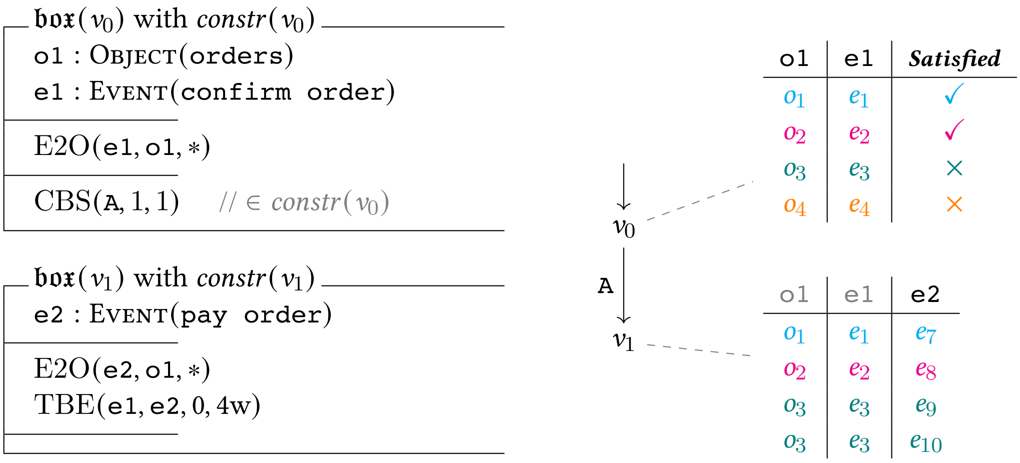
4.3 Process Constraints using Variable Bindings

In this section, we present how a query tree (compare Definition 4.11) can be combined with a set of constraint predicates for each tree node. The constraint predicates label the output bindings of the node, determining if an output binding is considered *violated* or *satisfied*. This labeling can be understood as a Boolean annotation of each output binding, indicating if the binding is violated or satisfied. Later, in Section 4.6, we will discuss more general annotation functions. In particular, this concept is not limited to constraints or boolean annotations, but can also encompass numerical values, like *Key Performance Indicators* (KPIs).

Definition 4.14 (Query Tree Constraint): Let $T = (V, F, r, l, \mathfrak{box})$ be a query tree over an OCED L. The expanded tuple $C = (T, constr)$ is called a *query tree constraint* under L, where $constr\colon V \to \mathcal{P}(\mathbb{P}_L)$ assigns each node $u \in V$ to an additional set of predicates. For each $u \in V$, $constr(u) \subseteq \mathbf{BASIC_L} \cup \mathbf{ATTRS}_L \cup \mathbf{CHILD\ SET}_u^T \cup \mathbf{CONSTR}_u^C$, where $\mathbf{CONSTR}_u^C$ is the collection of constraint predicates introduced next.
For a node $u \in V$, we say that an output binding $b \in out_L(\mathfrak{box}(u))$ of u is *satisfied* if $b \models constr(u)$ and *violated* otherwise (i.e., if $b \not\models constr(u)$).

It is important to note that a query tree constraint only combines a query tree with a set of constraint predicates for each node. The semantics of the binding boxes encompassed by the nodes of the query tree stay the same: The defined output binding sets do not change. Instead, the constraint function additionally *labels* the output bindings, either marking them as allowed behavior or forbidden. Before we introduce the new collection of predicates, $\mathbf{CONSTR}_u^C$, we first present an example query tree constraint in Example 4.15, which only uses the previously introduced constraint types.

Example 4.15 (Query Tree Constraint): Consider an example OCED L. Additionally, consider the query tree constraint $C = ((V, F, r, l, \mathfrak{box}), constr)$, with $V = \{v_0, v_1\}$, $r = v_0$ and $F = \{(v_0, v_1)\}$. The graph (V, F) with the edge labels l is shown on the right, while $\mathfrak{box}$ with $constr$ is presented on the left. For each $v \in V$, the set $constr(v)$ is visually shown in the corresponding binding box below an extra line. This constraint specifies that every confirmed order *should* be paid within 4 weeks after the confirmation exactly once. In the exemplary output binding tables shown on the right, this holds for all binding rows of $\mathfrak{box}(v_0)$ except the last row. In particular, all output rows except the last two have exactly one child binding in the output table for $\mathfrak{box}(v_1)$. The violation status of an output binding of $\mathfrak{box}(v_0)$ is then simply annotated to the corresponding output row as either ✓ or ×.



$\mathfrak{box}(v_0)$ with $constr(v_0)$

o1 : Object(orders)
e1 : Event(confirm order)

E2O(e1, o1, ∗)

CBS(A, 1, 1) // ∈ $constr(v_0)$

$\mathfrak{box}(v_1)$ with $constr(v_1)$

e2 : Event(pay order)

E2O(e2, o1, ∗)
TBE(e1, e2, 0, 4w)

o1	e1	*Satisfied*
o_1	e_1	✓
o_2	e_2	✓
o_3	e_3	×
o_4	e_4	×

o1	e1	e2
o_1	e_1	e_7
o_2	e_2	e_8
o_3	e_3	e_9
o_3	e_3	e_{10}

The new constraint collection $\mathbf{CONSTR}_u^C$ contains different predicates, which allow implementing advanced logic gates and constructs. For instance, such predicates allow propagating violations for child bindings of a child node to the parent binding. Thus, constraints can also be used in child nodes and used to derive the violation status of a parent node.

Transfering the semantics of logic gates, like OR, to binding predicate constraints requires also handling the possibility of none or multiple child binding per input parent binding. For example, there might be scenarios where a parent node should be satisfied for a parent binding only if *all child bindings* are satisfied in the specified child node (for-all semantics). However, sometimes scenarios might require that the parent node should be satisfied for a parent binding if *at least one child binding* in the child node is satisfied (exists-semantics). To handle this complexity, we first introduce two basic constraint predicate types representing those two options: ALL SAT for the for-all semantics and ANY SAT for the exists semantics. For the more complex gates NOT, OR, and AND, we then only include a for-all variant. Because query trees can be nested, it is then still possible to model, for example, an NOT gate with exists semantics, by chaining a simple node with the ALL SAT predicate constraint inbetween.

Next, we formally define the new predicate collection $\mathbf{CONSTR}_u^C$.

Definition 4.15 (Constraint Predicate Collection $\mathbf{CONSTR}_u^C$): Let L be an OCED and $C = ((V, F, r, l, \mathfrak{box}), constr)$ be a query tree constraint over L. For a node $u \in V$, $\mathbf{CONSTR}_u^C$ contains the following new predicate types:

- **All Child Bindings Satisfied (ALL SAT)**: For all $v \in V$ with $e = (u, v) \in F$ and $l(e) = \mathtt{A}$, there is $\mathrm{ALL\,SAT}(\mathtt{A}) \in \mathbf{CONSTR}_u^C$, with for any $b \in out_L(\mathfrak{box}(u))$:

$$b \models \mathrm{ALL\,SAT}(\mathtt{A}) \;\Leftrightarrow\; \forall_{b' \in out_L(\mathfrak{box}(v)) \wedge b \sqsubseteq_L b'}\; b' \models constr(v)$$

- **Any Child Binding Satisfied (ANY SAT)**: For all $v \in V$ with $e = (u, v) \in F$ and $l(e) = \mathtt{A}$, there is $\mathrm{ANY\,SAT}(\mathtt{A}) \in \mathbf{CONSTR}_u^C$, with for any $b \in out_L(\mathfrak{box}(u))$:

$$b \models \mathrm{ANY\,SAT}(\mathtt{A}) \;\Leftrightarrow\; \exists_{b' \in out_L(\mathfrak{box}(v)) \wedge b \sqsubseteq_L b'}\; b' \models constr(v)$$

- **No Child Binding Satisfied (ALL NOT)**: For all child nodes $v \in V$ with $e = (u, v) \in F$ and $l(e) = \mathtt{A}$, there is $\mathrm{ALL\,NOT}(\mathtt{A}) \in \mathbf{CONSTR}_u^C$, with for any $b \in out_L(\mathfrak{box}(u))$:

$$b \models \mathrm{ALL\,NOT}(\mathtt{A}) \;\Leftrightarrow\; \forall_{b' \in out_L(\mathfrak{box}(v)) \wedge b \sqsubseteq_L b'}\; b' \not\models constr(v)$$

- **For Any Child ALL SAT Holds (OR ALL)**: For all child nodes $v_1, \ldots, v_n \in V$ with $\forall_{i \in \underline{n}}\, e_i = (u, v_i) \in F$ and $l(e_i) = \mathtt{A}_i$, there is $\mathrm{OR\,ALL}(\{\mathtt{A}_1, \ldots, \mathtt{A}_n\}) \in \mathbf{CONSTR}_u^C$, with for any $b \in out_L(\mathfrak{box}(u))$:

$$b \models \mathrm{OR\,ALL}(\{\mathtt{A}_1, \ldots, \mathtt{A}_n\}) \;\Leftrightarrow\; \exists_{i \in \underline{n}} \forall_{b' \in out_L(\mathfrak{box}(v)) \wedge b \sqsubseteq_L b'}\; b' \models constr(v)$$

- **For All Children ALL SAT Holds (AND ALL)**: For all $v_1, \ldots, v_n \in V$ with $\forall_{i \in \underline{n}}\, e_i = (u, v_i) \in F$ and $l(e_i) = \mathtt{A}_i$, there is $\mathrm{AND\,ALL}(\{\mathtt{A}_1, \ldots, \mathtt{A}_n\}) \in \mathbf{CONSTR}_u^C$, with for any $b \in out_L(\mathfrak{box}(u))$:

$$b \models \mathrm{AND\,ALL}(\{\mathtt{A}_1, \ldots, \mathtt{A}_n\}) \;\Leftrightarrow\; \forall_{i \in \underline{n}} \forall_{b' \in out_L(\mathfrak{box}(v)) \wedge b \sqsubseteq_L b'}\; b' \models constr(v)$$

Now that we introduced query tree constraints, we can take a look at a few example constraints making use of these new predicates. Example 4.16 shows how the OR predicate types can be used to model a classical OR gate, connecting two sub-constraints.

Example 4.16 (Logic OR Gate): Consider an arbitrary OCED L. Given a constrained query tree, we denote the tree structure on the right, as usual. The template indicated below, v_0 implements a simple logic OR-gate. On the right, exemplary output tables with the annotated violation status are shown. Notice, that because of the *for all* semantics of OR ALL, bindings are also considered satisfied if for any child node there is no child binding in the output set. For instance, the second output binding row for v_0 (shown in magenta) is satisfied, as there is no child binding in the output set of v_2.

$\mathfrak{box}(v_0)$ with $constr(v_0)$

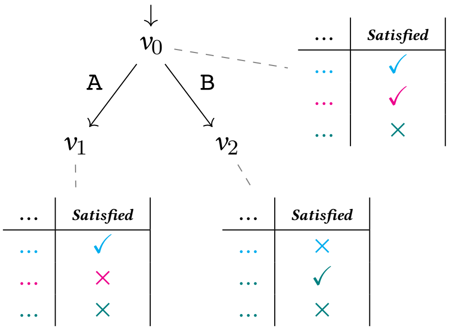
OR ALL($\{$A, B$\}$)

$\Delta\mathfrak{box}(v_1)$ with $constr(v_1)$

...

$\Delta\mathfrak{box}(v_2)$ with $constr(v_2)$

...



...	*Satisfied*
...	✓
...	✓
...	×

...	*Satisfied*
...	✓
...	×
...	×

...	*Satisfied*
...	×
...	✓
...	×

Next, we present an adapted version of Example 4.14 in Example 4.17. In contrast to the previous version without constraints, this query tree constraint *specifies* that for all orders by customers, where at least three payment reminders were sent out for the order, the customer *should* also been banned.

Example 4.17 (Query Tree Constraint): Consider the OCED L from Example 4.7. Additionally, consider the query tree constraint $C = ((V, F, r, l, \mathfrak{box}), constr)$, with $V = \{v_0, v_1, v_2\}$, $r = v_0$ and $F = \{(v_0, v_1), (v_0, v_2)\}$. The graph (V, F) with labels l is shown on the right with exemplary output tables, while $\mathfrak{box}$ with *constr* is shown on the left. For each $v \in V$, the set $constr(v)$ is visually shown in the corresponding binding box below an extra line. The constraint specifies that if there is an order for which at least 3 payment reminders were sent out, the customer of this order should be banned.

$\mathfrak{box}(v_0)$ with $constr(v_0)$
`o1` : Object(`orders`)
`o2` : Object(`customer`)

O2O(`o1`, `o2`, `customer`)
CBS(A, 3, ∞) // Set Filter

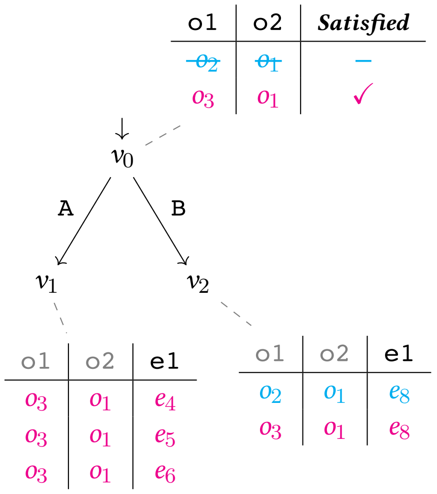
CBS(B, 1, ∞) // Constraint (part of $constr(v_0)$)

$\Delta\mathfrak{box}(v_1)$ with $constr(v_1)$
`e1` : Event(`payment reminder`)

E2O(`e1`, `o1`, `order`)

$\Delta\mathfrak{box}(v_2)$ with $constr(v_2)$
`e1` : Event(`ban customer`)

E2O(`e1`, `o2`, `customer`)



o1	o2	*Satisfied*
~~o_2~~	~~o_1~~	–
o_3	o_1	✓

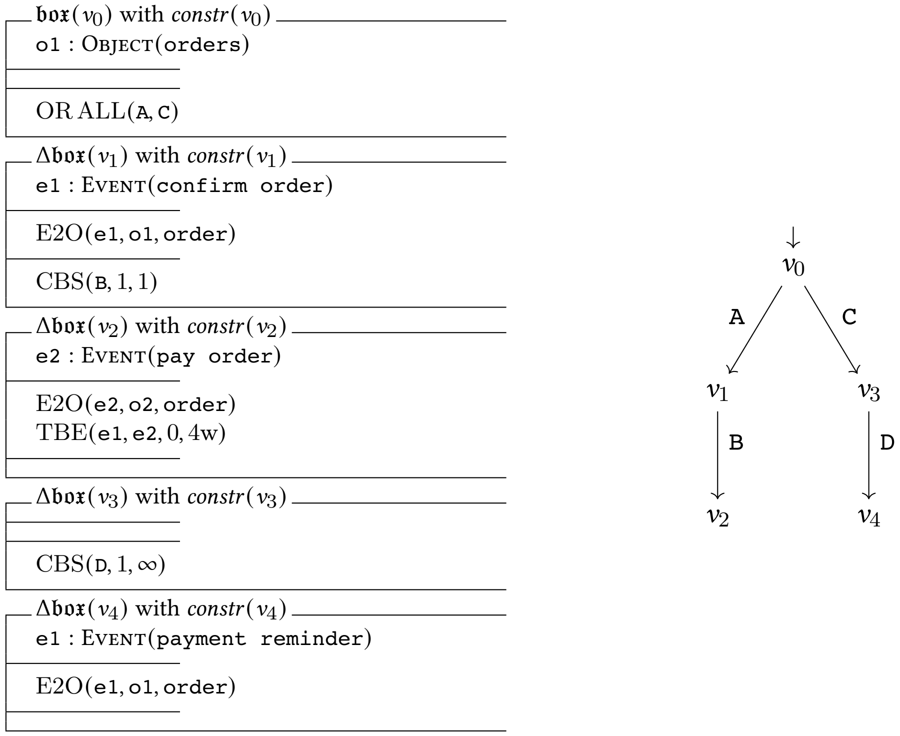
o1	o2	e1
o_3	o_1	e_4
o_3	o_1	e_5
o_3	o_1	e_6

o1	o2	e1
o_2	o_1	e_8
o_3	o_1	e_8

Finally, in Example 4.18, which is shown below, we present a more complex constraint consisting of five nodes, demonstrating how an OR predicate can be used in practice.

Example 4.18 (Query Tree Constraint): Consider the OCED L from Example 4.7. Additionally, consider the query tree constraint $C = ((V, F, r, l, \mathfrak{box}), constr)$, with $V =$

$\{v_0, v_1, v_2\}$, $r = v_0$ and $F = \{(v_0, v_1), (v_0, v_2)\}$. The graph (V, F) with labeling l is presented on the right, while $\mathfrak{box}$ with *constr* is shown on the left. For each $v \in V$, the set $constr(v)$ is visually shown in the corresponding binding box below an extra line. This constraint specifies that an order should be either paid fast after confirmation (i.e., within 4 weeks) *or* at least one payment reminder should be sent out for it.

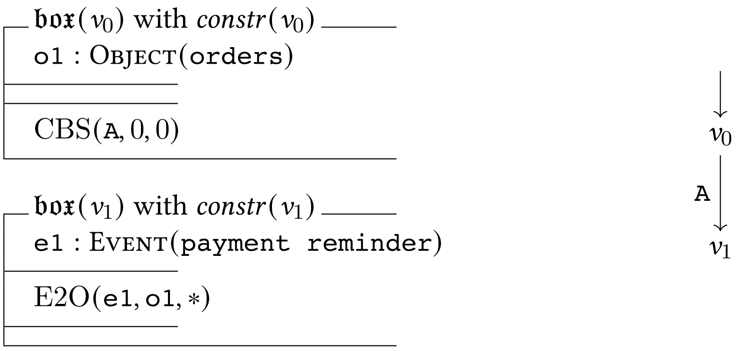

$\mathfrak{box}(v_0)$ with $constr(v_0)$
o1 : Object(orders)
OR ALL(A, C)
$\Delta\mathfrak{box}(v_1)$ with $constr(v_1)$
e1 : Event(confirm order)
E2O(e1, o1, order)
CBS(B, 1, 1)
$\Delta\mathfrak{box}(v_2)$ with $constr(v_2)$
e2 : Event(pay order)
E2O(e2, o2, order)
TBE(e1, e2, 0, 4w)
$\Delta\mathfrak{box}(v_3)$ with $constr(v_3)$
CBS(D, 1, ∞)
$\Delta\mathfrak{box}(v_4)$ with $constr(v_4)$
e1 : Event(payment reminder)
E2O(e1, o1, order)
v_0
A
C
v_1
v_3
B
D
v_2
v_4

Finally, we also want to quantify the filter and violation percentages of nodes in a query tree constraint.

Definition 4.16 (Quantifying Violation Fractions): Let L be an OCED. For a binding box $\mathfrak{b}_L$ under L, the number of output bindings is $n = |out_L(\mathfrak{b}_L)|$.
Considering an additional set of constraint predicates $X \subseteq \mathbb{P}_L$ for $\mathfrak{b}_L$, we can also calculate the violation count and percentage. The *violation count* is $c_v = |\{b \in out_L(\mathfrak{b}_L) \mid b \not\models X\}|$. Similarly, the *violation fraction* (often represented as a percentage) is $q_v = \frac{c_v}{n}$.

For instance, consider the simple query tree constraint presented in Example 4.19. In this constraint, the root node is violated in $c_v = 443$ out of the $n = 2000$ total possible bindings of `orders` objects in the OCED from [36]. In particular, for 443 orders there is at least one `payment reminder` event. Thus, we can calculate the violation fraction of the root node as $\frac{443}{2000} \approx 22.15\%$.

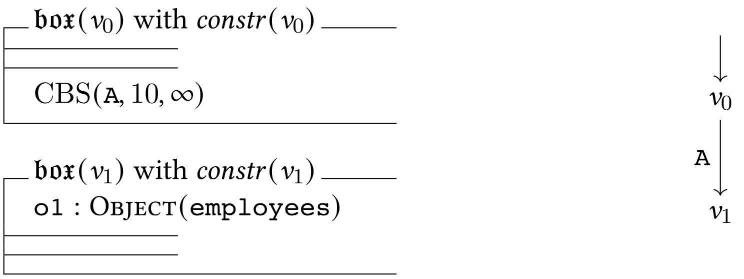
Example 4.19 (Simple Constraint for Calculating Violation Percentage): Consider the query tree constraint shown below. The root node is satisfied for bindings of `orders` objects if there is no event of type `payment reminder` associated with it.


box(v_0) with $constr(v_0)$
o1 : OBJECT(orders)
CBS(A, 0, 0)
box(v_1) with $constr(v_1)$
e1 : EVENT(payment reminder)
E2O(e1, o1, ∗)
v_0
A
v_1

When we discuss an overall query tree constraint as being satisfied or violated, we commonly refer to the violation percentage of its root node. In contrast, the violation percentage of non-root nodes should not be considered directly, as it can be propagated up to the root node in different ways. For instance, when using the logic gate NOT ALL, a high violation percentages of a child node might yield a low violation percentage at the parent node.

Our approach can also be adapted to *global scenarios* where constraints should be formulated not for individual queried bindings but the overall OCED. For that, the root node can simply not introduce any variables, resulting in only the empty binding as an output, which can then represent the violation status of the complete OCED as a whole. For instance, consider Example 4.20 which can be considered globally satisfied if there are at least 10 employees in the whole considered OCED.

Example 4.20 (Global OCED Constraint): Consider the query tree constraint presented below. As the root node does not query any object or event variables and also does not have any predicates, it only contains the empty binding {} as a single output binding. This is convenient to represent global constraints for the considered OCED as a whole. In particular, this query tree node is satisfied (i.e., the empty binding is satisfied for the root node) when there are at least 10 `employees` objects in the considered OCED.



The addition of constraints finalizes our main query and constraint approach. Next, we will present further details regarding the efficient evaluation of binding queries, automatic discovery of constraints and extensions to our approach. In those, we sometimes use an alternative query tree constraint visualization, to further ease readability and represent them more compactly. This alternative representation is also used in the frontend implementation of the tool in Chapter 5. Figure 4.1 shows how this alternative visualization looks like. For instance, predicate types like E2O and O2O are represented using a link icon between the variables names. Note, that as the variables already have a clear type (i.e., are either object or event variable), there is no need to explicitly distinguish between both types of relationship predicate, as they are already indicated by the type of the involved variables. Also, the names of the logic constraint predicates are slightly shortened. In particular, ALL SAT is simply called SAT while ANY SAT is called ANY.

Moreover, ALL NOT, OR ALL, and AND ALL are shown as NOT, OR and AND, respectively.

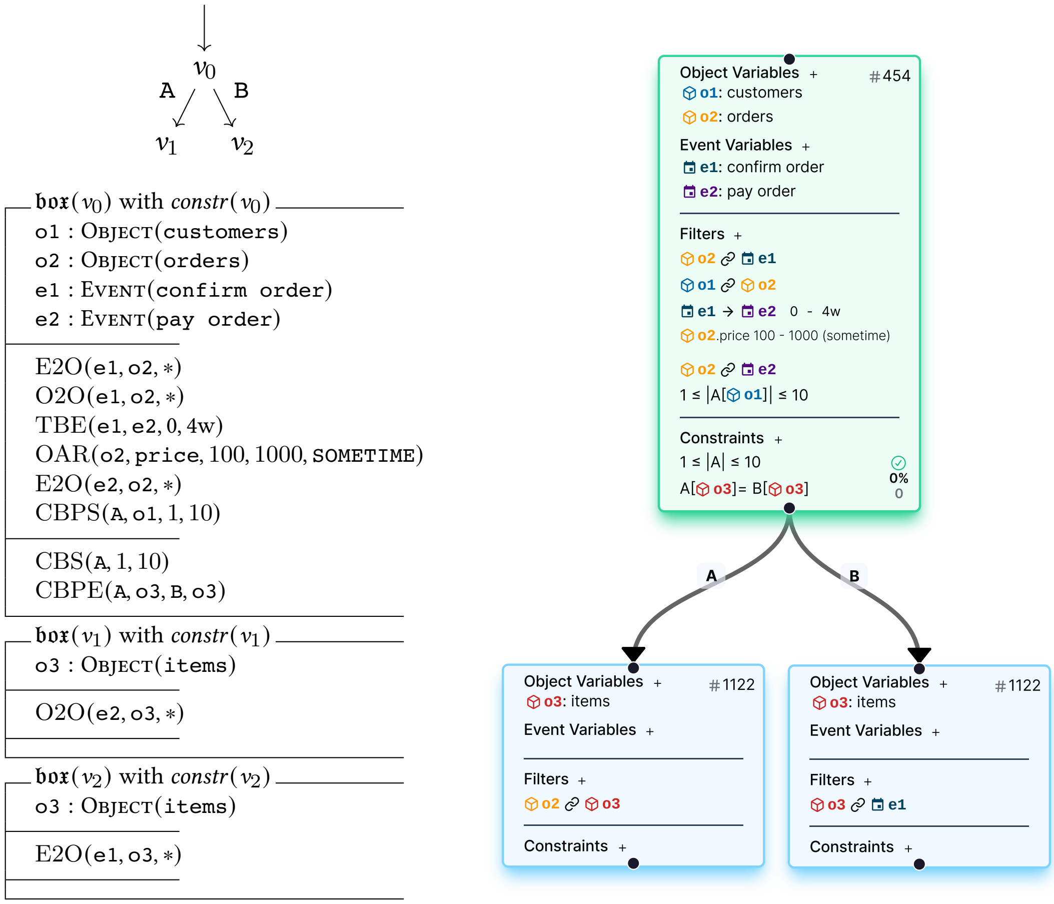


Figure 4.1: The formal representation of a query tree constraint, on the left, with the equivalent fully graphical representation on the right. The presented *kitchen sink* example constraint was constructed to contain many predicate types, to show their representation, but has no intended semantic meaning. When evaluated on the OCED from [36], the root node yields 454 output bindings and is never violated, which is shown on the right top and bottom of the root node. As the two child nodes do not contain any constraint predicates, no violation status is shown for them.

In the next sections, we will discuss more details related to our method. First, we start by describing how the declarative definitions given here can be adapted algorithmically.

4.4 Efficiently Evaluating Binding Queries for OCED

In this section, we describe how to efficiently evaluate the query and constraint concepts introduced previously. Here, we do not describe implementation details (see Chapter 5 for that), but rather the general technique. As such, this section answers **RQ4** and encompasses the research goal **RG5** by providing algorithms allowing for efficient evaluation of declarative queries and constraints, corresponding to our contribution **CT3**.

First, we describe how the evaluation of the declarative approach presented in the previous sections can be translated to a recursive algorithm. The recursive algorithm takes a query tree constraint and an OCED as input and computes all output bindings for each node of the tree and labels each of them according to the defined constraint predicates. Of course, the same approach can also be used to evaluate query trees without constraints, by simply assuming an empty constraint set for each node. Secondly, we describe a substep of this described algorithm, in which an input binding (i.e., a binding of the parent node) is expanded to the output set of a child node in more detail.

4.4.1 Recursive Binding Query Algorithm

In the following, we present a recursive algorithm which, given a query tree (optionally with additional children filter predicates and constraint predicates), calculates the set of output bindings for each node, optionally with constraint information. The algorithm is sketched in Code 3 as Python-like pseudocode. The function `evaluate` is implemented for tree nodes of a query tree constraint and takes a parent binding as an input parameter. Initially, the `evaluate` function of the root node is called with the (implicit) empty parent binding (i.e., {}). The `evaluate` function then consists of three main steps:

1. Expand the input parent binding (line 2, considering $\mathbf{BASIC}_\mathrm{L} \cup \mathbf{ATTRS}_L$), and for each resulting binding...

2. ... evaluate all children and check child set constraints (lines 6 - 11, based on $\mathbf{CHILD\ SET}_u^T$).

3. ... check if the constraints are satisfied (line 12, based on $constr(u)$).

```
1  def evaluate(self: TreeNode, parent_binding: Binding):
2     expanded = self.expand(parent_binding) # Considers $\mathbf{BASIC}_\mathrm{L} \cup \mathbf{ATTRS}_L$
3     all_res = []
4     for b in expanded:
5         res_per_child = {}
6         for c in self.children:
7             c_res = c.evaluate(b)
8             res_per_child[c.name] = c_res
9             all_res.extend(c_res)
10        # Next, check predicates in $\mathbf{CHILD\ SET}^T_{self}$
11        if self.passes_child_filters(res_per_child):
12            viol = self.get_violated_constraints(b,res_per_child)
13            all_res.push([self.name,b,viol])
14    return all_res
```

Code 3: Python-like pseudo code sketching a recursive algorithm to evaluate query trees.

We first describe the algorithm and the inner subroutines and then argue that the algorithm indeed correctly computes the labeled output bindings of each binding box. The algorithm corresponds to the `evaluate` function implemented for the nodes in a query tree constraint. Initially, the `evaluate` function of the root node of the tree is called with the implicit empty parent binding $\{\}$. In the function, the parent binding is first expanded to superset of output bindings of the current node, based on only predicates in $\mathbf{BASIC_L}$ and $\mathbf{ATTRS}_L$. This output set is indeed a superset of the actual output bindings of the node u, as the predicates from $\mathbf{CHILD\ SET}_u^T$ are not yet considered. Next, each binding b of this superset of output binding is considered on its own: Each child node of u is evaluated recursively and the results are saved. After that, based on the obtained child results, it can be checked if the binding b satisfies the predicates in $\mathbf{CHILD\ SET}_u^T$ which were not yet considered. If this is the case, the violation predicates are checked, based on b and the child results, and the obtained information is added to the overall output. If not, i.e., b is not part of the actual output set of u, this is not done. Either way, the recursive results of the children are also added to the overall result, which is returned at the end.

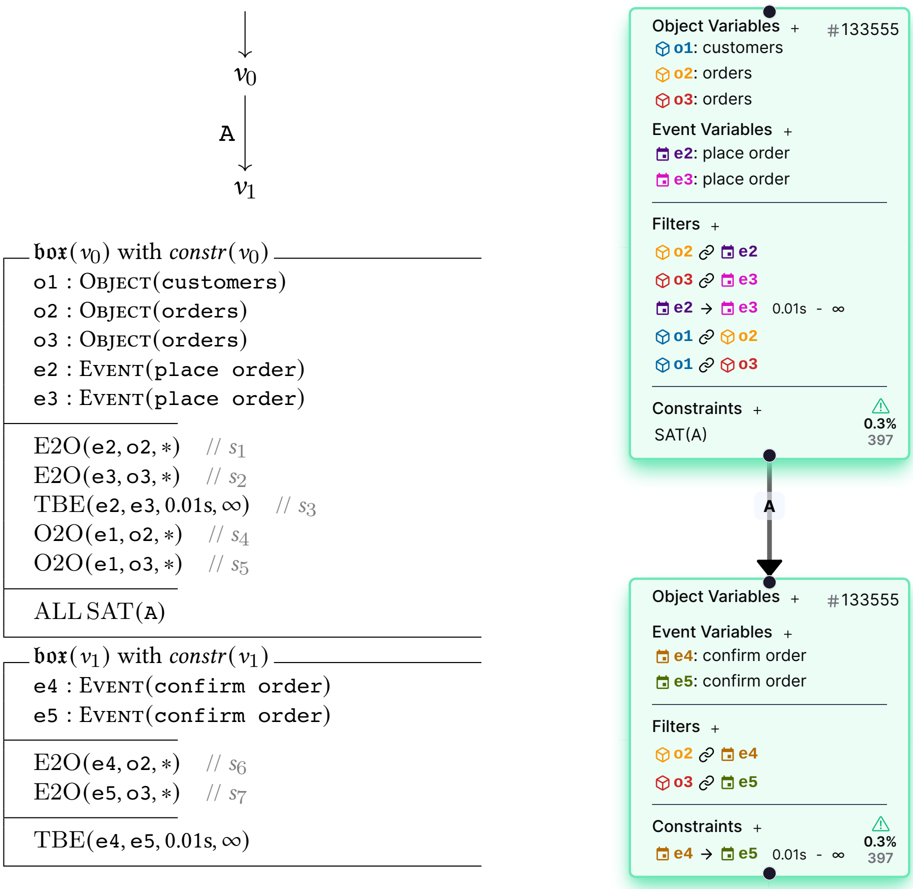
Lemma 4.3 (Recursive Algorithm Computes Correct Output Bindings): We want to show that the recursive algorithm from Code 3 indeed computes the output sets, as formally described in Section 4.2. Let L be an OCED and let $C = ((V, F, r, l, \mathfrak{box}), constr)$ be a query tree constraint. From Lemma 4.2, we know that for two binding boxes $\mathfrak{a}, \mathfrak{b} \in \mathfrak{BOX}_L$ with $\mathfrak{a} \preceq_L \mathfrak{b}$ and any binding $b \in \mathbb{B}_L$ with $b \models \mathfrak{b}$, there exists a unique parent binding $a \in \mathbb{B}_L$ with $a \sqsubseteq_L b$ and $a \models \mathfrak{a}$. Note, that for two nodes $u, v \in V$ with $(u, v) \in F$, it only holds that $\mathfrak{box}(u)|_{\mathbf{BASIC_L} \cup \mathbf{ATTRS}_L} \preceq_L \mathfrak{box}(v)|_{\mathbf{BASIC_L} \cup \mathbf{ATTRS}_L}$, and the binding boxes might additionally contain predicates from $\mathbf{CHILD\ SET}_u^C$. However, in the recursive algorithm from Code 3 child bindings are constructed and evaluated, even if the parent binding does not pass the child filter predicates (cf. line 9). For every output binding $b \in \mathbb{B}_L$ of a node $v \in V$, we can construct a sequence $\pi = \langle (b_1, r), \ldots, (b, v) \rangle$ of bindings and corresponding nodes, such that the nodes (i.e., $\langle r, \ldots, v \rangle$) correspond to a path in the tree (V, F) and all bindings (i.e., $\langle b_1, \ldots, b \rangle$, where $b_1 \models \mathfrak{box}(r)$) are in a $\sqsubseteq_L$ relation. This sequence then corresponds to the recursive calls of the `evaluate` function in Code 3. Thus, inductively clearly all output bindings of all nodes are constructed and returned. Furthermore, as bindings are recursively expanded and filtered based on the provided predicates, there are also no output bindings constructed and returned for nodes, which are not part of their output set. Thus, overall the recursive algorithm sketched in Code 3 computes exactly the output of the nested querying of a query tree, formalized before.

The subroutines `passes_child_filters` and `get_violated_constraints` are rather trivial and do not require extra attention. The binding expansion step (corresponding to the `expand` function call in line 2 from Code 3), however, is more involved and of particular interest. Next, we zoom in on this binding expansion step. As we will see, a naive implementation quickly runs into huge scalability problems. Thus, we also present more intelligent solutions that enables efficient expansion of bindings in practice.

4.4.2 Expanding Bindings

The binding expansion step constructs the (super)set of the child bindings of a node u based on a parent binding. In particular, given parent binding $b \in \mathbb{B}_L$, the set $\{b' \in \mathbb{B}_L \mid b \sqsubseteq_L b' \land b' \models \mathfrak{box}(u)|_{\mathbf{BASIC_L} \cup \mathbf{ATTRS}_L}\}$ is constructed. Intuitively, this involves first constructing a child binding for all possible combinations of the newly bound variables and afterwards filtering this set based on the basic and attribute-filtering predicates of $\mathfrak{box}(u)$.

We use the example order management OCED from [36] to quantify how well different approaches filter out bindings early or prevent the construction of unnecessary bindings. Moreover, we consider the constraint from Figure 4.2 as a running example for a more complex query.



$\mathfrak{box}(v_0)$ with $constr(v_0)$
o1 : Object(customers)
o2 : Object(orders)
o3 : Object(orders)
e2 : Event(place order)
e3 : Event(place order)

E2O(e2, o2, $*$) // s_1
E2O(e3, o3, $*$) // s_2
TBE(e2, e3, 0.01s, ∞) // s_3
O2O(e1, o2, $*$) // s_4
O2O(e1, o3, $*$) // s_5

ALL SAT(A)

$\mathfrak{box}(v_1)$ with $constr(v_1)$
e4 : Event(confirm order)
e5 : Event(confirm order)

E2O(e4, o2, $*$) // s_6
E2O(e5, o3, $*$) // s_7

TBE(e4, e5, 0.01s, ∞)

Object Variables + #133555
o1: customers
o2: orders
o3: orders
Event Variables +
e2: place order
e3: place order
Filters +
o2 🔗 e2
o3 🔗 e3
e2 → e3 0.01s - ∞
o1 🔗 o2
o1 🔗 o3
Constraints + ⚠ 0.3% 397
SAT(A)

A

Object Variables + #133555
Event Variables +
e4: confirm order
e5: confirm order
Filters +
o2 🔗 e4
o3 🔗 e5
Constraints + ⚠ 0.3% 397
e4 → e5 0.01s - ∞

Figure 4.2: A query tree constraint, specifying that two orders placed by the same customer should be confirmed in the order they were placed in. A formal representation is shown on the left and the graphical representation inside the tool UI on the right. In the formal representation, the basic predicates in $\mathfrak{box}(v_0)$ and $\mathfrak{box}(v_1)$ are labeled s_1 through s_7.

In the following, we first demonstrate the problem of evaluating queries by describing the most naive approach to computing output bindings for a binding box. Next, we show how this naive approach can be improved using the simple idea of *filtering early*. However, as we will see, this improvement is still not enough to make evaluating more complex queries feasible. After that, we describe a more sophisticated approach which exploits the types of filter predicates available in **BASIC**$_\mathrm{L}$ and can also handle complex queries.

Naive Approach

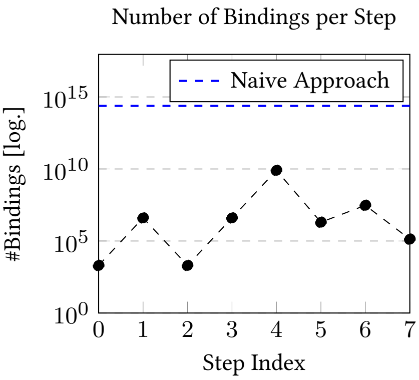
A very naive way to bind all the variables of a binding box would be to simply expand the input binding to one output binding for each possible combination of values. In the example OCED from [36], there are 2000 order objects, 2000 place order events and 15 customer objects. Thus, for $\mathfrak{box}(v_0)$ from Figure 4.2, this would mean constructing $2000 \cdot 2000 \cdot 2000 \cdot 2000 \cdot 15 =$

$240,000,000,000,000 = 2.4 \cdot 10^{14}$ (in words: 240 trillion) bindings for the implicit empty parent binding. Clearly, first constructing all these bindings and then filtering them down, based on the binding predicates, is infeasible.

Early Filtering

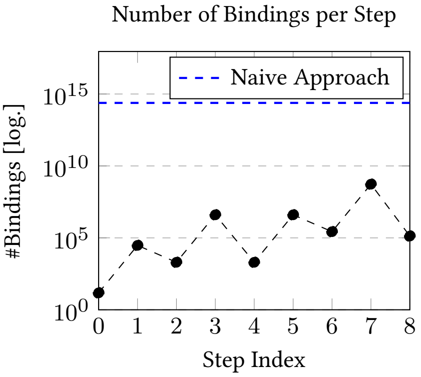
Instead, a better approach would be to filter out bindings as early as possible. For simplicity, first assume some given sequential order of the variables. After binding a new variable, all filter predicates only involving variables already bound to values can be evaluated to determine if the bindings should be kept or discarded. For example, assuming an ordering o2, e2, o3, e3, o1 of the involved variables, after binding o2 and e2, we can already filter out all bindings, in which they are not associated with each other. Figure 4.3 shows the encountered number of bindings in each step for the example query at Figure 4.2 when using this *early filtering* technique, using two different variable orderings. Both scenarios already reduce the maximum number of bindings to consider significantly. The order of variables influences the maximum number of bindings to consider in a step. Later, we will also describe an approach to identify well-performing binding orders in more detail.

Index	Step	#Bindings
0	Bind o2	2000
1	Bind e2	4000000
2	Filter (s_1)	2000
3	Bind o3	4000000
4	Bind e3	8000000000
5	Filter (s_2, s_3)	1999000
6	Bind o1	29985000
7	Filter (s_4, s_5)	133555



(a) Number of bindings per step for ordering o2, e2, o3, e3, o1

Index	Step	#Bindings
0	Bind o1	15
1	Bind o2	30000
2	Filter (s_4)	2000
3	Bind e2	4000000
4	Filter (s_1)	2000
5	Bind o3	4000000
6	Filter (s_5)	269110
7	Bind e3	538220000
8	Filter (s_2, s_3)	133555



(b) Number of bindings per step for ordering o1, o2, e2, o3, e3

Figure 4.3: Number of bindings after each step, assuming the given order for binding the variables. Predicate filters are evaluated as soon as all involved variables are bound. The plots show the number of bindings for each step with a logarithmic y-axis. For both orderings, the maximum number of encountered bindings is significantly lower than in the naive approach. Moreover, the ordering on the bottom involves constructing fewer bindings than the one at the top.

Intelligent Binding Expansion During Construction

While early filtering reduces the maximum number of bindings one has to consider significantly, it still constructs a huge number of unneeded bindings. Next, we describe a more sophisticated approach, which combines early filtering with only expanding necessary variable combinations during construction. To achieve this, we leverage the nature of most of the binding predicates in $\mathbf{BASIC_L}$: They correspond to object-to-object or event-to-object relationships in the OCED. Thus, if one of the variables involved in the predicate is already bound, we can restrict binding the other value to only those values that are in the specified relationship with the already bound variable value. For example, when expanding bindings by binding order objects `o2` based on an already bound customer `o1`, it suffices to consider all orders which are contained in an object-to-object relationship with the value of `o1`. These related values can easily be precomputed as an index when loading an OCED.

To approach this formally, we describe the *Relationship Graph* of an OCED. It containts all elements in the OCED (i.e., all objects and events) as nodes with edges between two elements, if there is a relation between them in the OCED.

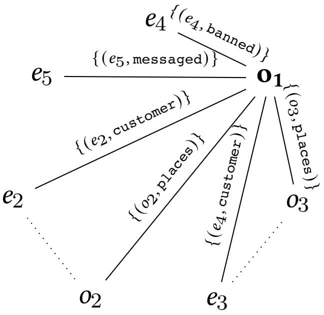
Definition 4.17 (OCED Relationship Graph): Let $L = (E, O, eaval, oaval)$ be an OCED. For a given qualifier $q \in \mathcal{U}_{qual}$, we define a relation $\rightarrow^q_L$ between elements of $O_L \cup E_L$. Given $a, b \in O_L \cup E_L$, we define $a \rightarrow^q_L b$ to hold if and only if:

- $a, b \in O_L$ and $(q, b) \in oaval_a(\texttt{objects})$
- $a \in E_L$, $b \in O_L$ and $(q, b) \in eaval_a(\texttt{objects})$

We can construct a *OCED Relationship Graph* consists of the nodes $V_L = O_L \cup E_L$ and undirected edges $F_L = \Big\{\{x, y\} \mid x, y \in V \land \exists_{q \in \mathcal{U}_{qual}} \; x \rightarrow^q_L y\Big\}$.
For each edge, we can also construct the set of qualifiers and source node supporting this edge. For that, we introduce a function $qual : F_L \rightarrow \mathcal{P}(V_L \times \mathcal{U}_{qual})$ defined as $qual_L(e) = \Big\{(x, q) \in \mathcal{U}_{qual} \mid x \in e \land y \in e \land (x \neq y \lor |e| = 1) \land x \rightarrow^q_L y\Big\}$.

Typically, it is interesting to consider only a subgraph of the OCED Relationship Graph, centered around a specified node and only containing edges related to this node. Next, we present and visualize an example OCED Relationship subgraph.

Example 4.21 (OCED Relationship Subgraph): Below, we show an OCED Relationship Graph subgraph for a customer o_1. The set of qualifiers given by *qual* is annotated on the edges. Dotted arrows indicate some edges that are not shown in this subgraph. For instance, the subgraph shows object-to-object relations between the orders o_2 and o_3 and the customer o_1, as well as multiple event-to-object relationships.


e_4
$\{(e_4, \texttt{banned})\}$
e_5
$\{(e_5, \texttt{messaged})\}$
$\mathbf{o_1}$
$\{(e_2, \texttt{customer})\}$
$\{(o_3, \texttt{places})\}$
e_2
$\{(o_2, \texttt{places})\}$
$\{(e_4, \texttt{customer})\}$
o_3
o_2
e_3

Intuitively, if a variable is already bound to a value a before and a new variable should be bound, for which there is a filter predicate specifying a relationship with a, we only need to consider the neighbors of a in the OCED Relationship Graph. In particular, a similiar graph, based on relations, can also be constructed based on the variables involved in the binding box. Assume that we expand a binding box $\mathfrak{a}_L = (\text{Var}, \text{Pred})$ over L, based on a parent binding b. In the expanded output bindings, the variables $N = \text{dom}(\text{Var})$ should be bound to values. Of which, there might be some variables $\hat{N} = \text{dom}(b)$ already bound by the parent binding. Additionally, consider the set of basic binding predicates $P = \text{Pred} \cap \mathbf{BASIC_L}$ in $\mathfrak{a}_L$. We can construct a graph consisting of the variable in N with edges between them, indicating that the source can be *bound from other target variable's value* based on the predicates in P.

Definition 4.18 (Variable Dependency Graph): Let L be an OCED and $V \subseteq \mathcal{U}_{\text{evVar}} \cup \mathcal{U}_{obVar}$. Additionally, consider a set of basic predicates $P \subseteq \mathbf{BASIC_L}$. The graph (V, F) is the *variable dependency graph* of V and P, with:

$$F = \{(v_1, v_2) \in V \times V \mid \exists_{p \in P} \exists_{q \in \mathcal{U}_{qual} \cup \{*\}} \; p = \text{E2O}(v_1, v_2, q) \lor p = \text{O2O}(v_1, v_2, q) \\ \lor p = \text{E2O}(v_2, v_1, q) \lor p = \text{O2O}(v_2, v_1, q)\}$$

An edge $(v_1, v_2) \in F$ corresponds to the possibility to bind values to v_2 based on a value of the variable v_1. Notice, that here F is essentially symmetric, as we assume that the previously mentioned relationship graphs are constructed in a way which allows binding variables in both relation directions. However, in general, for example when considering expansions for other types of predicate, the set of edges might also be non-symmetric.

In Figure 4.4, we present such *variable dependency graphs* for the two nodes of the considered example shown in Figure 4.2. When expanding the implicit empty parent binding for the root node, first binding `o1` seems to make intuitively the most sense, as two other variables can be bound based on this values (i.e., `o2` and `o3`).

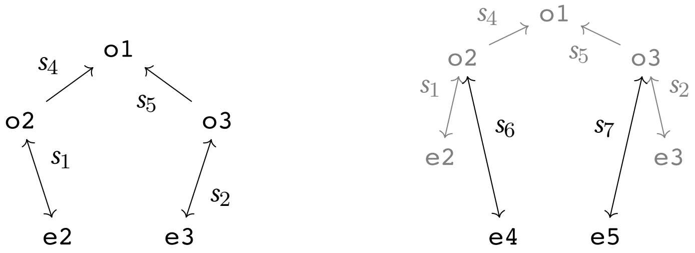


Figure 4.4: The variable dependency graph for the constraint shown in Figure 4.2. On the left, for the root node with an implicit empty parent binding. On the right, for the leaf node with a parent binding which already binds `o1`, `o2`, `o3`, `e2` and `e3`, which are thus shown in gray. The new variables `e4` and `e5` can easily be bound based on the already bound variables `o2` and `o3`, respectively, using an appropriate pre-computed index for the OCED relations.

Variables that can be bound based on other already bound variables should be selected first. Otherwise, for a more intelligent binding expansion approach, we propose iteratively picking the next variable to bind, based on the number of other, unbound variables that can be bound based on it, as indicated in the variable dependency graph. Of course, frequency information from the OCED, e.g., determining the number of possible values for a variable or the average number of relations of an object or event type, could be used to further enhance this heuristic.

Later, in Chapter 5, we provide more details about the actual implementation of those ideas in the OCPQ tool. Afterwards, in Chapter 6, we also evaluate the runtime performance of the OCPQ tool for executing queries or constraints across different OCEDs.

4.5 Discovering Constraints from OCED

In this section, we present how certain types of query tree constraints can be automatically discovered based on input OCED. As such, we address **RQ5**, investigating the automatic discovery of constraints, and fulfill **RG6**, as we present techniques for this automatic constraint discovery. More details on the actual implemented algorithms will be presented later in Chapter 5.

Automatic discovery is a well-studied challenging problem in process mining [38]. Typically, the goal is to discover process models in the form of Petri nets or BPMN models based on input event logs of the process, such that the discovered model is a good description of the underlying process [38]. Different criteria, like *fitness* and *precision*, need to be considered when evaluating if the discovered model is of good quality. For discovering constraints, similar considerations are required. Additionally, our developed constraint model focuses explicitly on high expressiveness and object-centricity, both of which render general discovery of any type of constraint that can be modeled using our approach infeasible. Instead, discovery needs to focus on specific types of constraints. In particular, focus should lay on types of constraints which are commonly relevant for processes. In the following, we will first present approaches to automatically discover the following two simple types of constraints, both with high practical relevance:

1. **Count Constraints**, which constrain the number of objects or events of one type (t') related to instances of a second type (t). For example, the count constraint "Exactly one `pay order` event per object of type `orders`" involves the event type t' = `pay order` and the object type t = `orders`.

2. **Eventually-Follows Constraints**, which constrain for each object of a specified object type (ot), after an event of one type (t) associated with the object, there should be an event of the second type (t') associated with the object within a specified duration. For example, for ot = `orders`, t = `confirm order` and t' = `pay order` a constraint could be: "For all `orders` and `confirm order` events there is at least one `pay order` event related to the order at most ≈ 5 weeks after the confirmation".

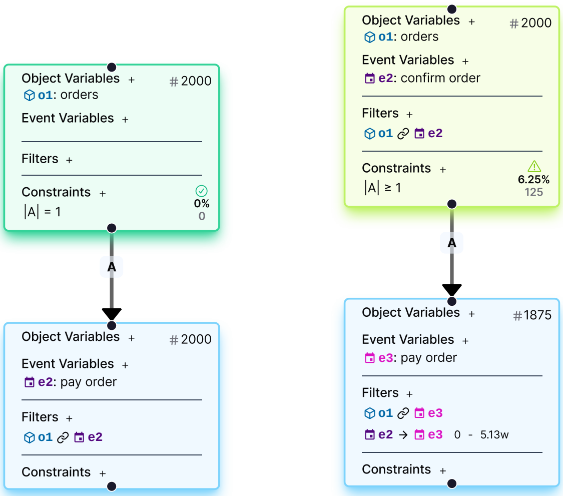


Figure 4.5: An example count constraint (on the left) and eventually-follows constraint (on the right) corresponding to the textual constraint descriptions given above. Both of these constraints were automatically discovered based on the example order management OCED from [36].

Afterwards, we also present a more sophisticated and general approach for combining (simple) discovered constraint constructs to more complex ones. For instance, this allows discovering OR-constructs like "An order is either paid fast (i.e., within ≈ 1.3 weeks) after confirmation or at least one payment reminder is sent out". Figure 4.8 shows such an automatically discovered OR constraint, incorporating both count and eventually-follows elements.

4.5.1 Discovering Count Constraints

Let $L = (E, O, eaval, oaval)$ be an OCED. We define the following function, which map an event or object type to the instances of that type in L:

- $\text{evinst}_L\colon \mathcal{U}_{etype} \to \mathcal{P}(E)$ with $\text{evinst}_L(et) = \{e \in E \mid eaval_e(\texttt{activity}) = et\}$
- $\text{obinst}_L\colon \mathcal{U}_{otype} \to \mathcal{P}(O)$ with $\text{obinst}_L(ot) = \{o \in O \mid oaval_o(\texttt{type}) = ot\}$
- $\text{inst}_L\colon\ (\mathcal{U}_{otype} \cup \mathcal{U}_{etype}) \to (\mathcal{P}(O) \cup \mathcal{P}(E))$ with $\text{inst}_L(t) = \begin{cases} \text{evinst}_L(t), & \text{if } t \in \mathcal{U}_{etype} \\ \text{obinst}_L(t), & \text{if } t \in \mathcal{U}_{otype} \end{cases}$

In particular, this also allows identifying the sets of events and object types which are occurring in L as $ET_L = \{et \in \mathcal{U}_{etype} \mid \text{evinst}_L(et) \neq \emptyset\}$ and $OT_L = \{ot \in \mathcal{U}_{otype} \mid \text{obinst}_L(ot) \neq \emptyset\}$.

Consider a type $t \in OT_L \cup ET_L$ in L. For all instances of type t in L, the number of instances of that type associated with instances of another type $t' \in OT_L \cup ET_L$ can be counted, resulting in a multiset of counts. Instances are considered associated with each other, if they are linked by the `object` attribute and thus through the obj^*_L function. The direction of this connection is also considered. For example, when considering $t = \texttt{pay order} \in ET_L$, an actual pay order event $e \in E$ is expected to be associated with exactly one object instance $o \in O$ of type `orders`, where then $o \in obj^*_L(e)$. For each combination of types t, t', we consider two multisets, counting the number of instances of t' associated with each instance of t. The functions $\#^n_L$ and $\#^r_L$ assign a combination of types to these multisets, where $\#^n_L$ considers the normal relation direction and $\#^r_L$ the reversed one.

$$\#^n_L, \#^r_L\colon (OT_L \cup ET_L) \times (OT_L \cup ET_L) \to \mathcal{B}(\mathbb{N}_0)$$
$$\#^n(t,t') = \left[\left|\left\{i' \in \text{inst}_L(t') \mid i' \in obj^*_L(i)\right\}\right| \;\middle|\; i \in \text{inst}_L(t) \right]$$
$$\#^r(t,t') = \left[\left|\left\{i' \in \text{inst}_L(t') \mid i \in obj^*_L(i')\right\}\right| \;\middle|\; i \in \text{inst}_L(t) \right]$$

> **Example 4.22 (Count Multisets):** Consider an example order management OCED L with $\text{inst}_L(\texttt{orders}) = \{o_1, o_2, o_3\}$ where o_1 and o_2 are associated with 5 objects of type `items` each and o_3 is associated with 3 objects of type `items`. For $t = \texttt{orders}$ and $t' = \texttt{items}$ then $\#^n(t,t') = [3,5,5]$ would be the count multiset for the normal relation direction.

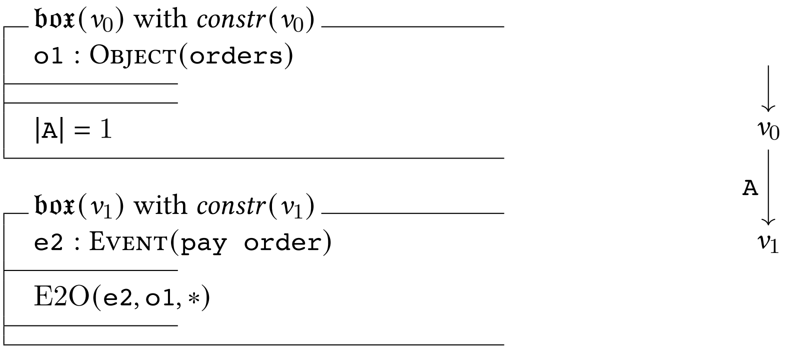
Note, that these definitions are very general for simplicity, even though there are some uninteresting type combinations. For instance, for two event types $t, t' \in ET_L$, clearly $\#^n_L(t,t')$ only contains zeros, as there are no direct event-to-event relations in our definition of OCED.

The assigned multisets can be used to efficiently mine count constraints. In particular, for each combination $t, t' \in ET_L \cup OT_L$, a direction $d \in \{n, r\}$ and a given threshold $0 \leq p \leq 1$ multiple different count ranges, each defined by their two bounds $c_{min}, c_{max} \in \mathbb{N}^{\infty}_0$ with $c_{min} \leq c_{max}$ can be identified such that the fraction of counts inside this range ($cfit_L$ defined below) is at least as large as p (i.e., $cfit_L(t, t', d, c_{min}, c_{max}) \geq p$).

$$cfit_L(t, t', d, c_{min}, c_{max}) = \frac{\left|[c \in \#_L^d(t, t') \mid c_{min} \leq c \leq c_{max}]\right|}{\left|\#_L^d(t, t')\right|}$$

Constructing Constraints We will later present different concepts to select interesting bounds $c_{min}, c_{max} \in \mathbb{N}_0^\infty$. First, we show how a query tree constraint can be constructed based on the identified values $t, t', d, c_{min}, c_{max}$. We omit a full formal definition here for simplicity, and instead only provide a brief description: For each identified combination of the values $t, t', d, c_{min}, c_{max}$ a query tree constraint can be constructed using two nodes: The first one binds an event or object variable to values of type t and contains a constraint for the number of child bindings in the second node, constraining them to be between c_{min} and c_{max}. The second node binds instances of type t' using a filter predicate of type E2O or O2O with t and t' (or flipped if $d = r$).

Example 4.23 (Constructing Count Constraint Tree): Let L be an OCED from an order management process. Consider the count configuration t = `orders`, t' = `pay order`, $c_{min} = c_{max} = 1$ and $d = r$. Below, we visualize the query tree constraint constructed for these values, $C = ((V, F, r, l, \mathfrak{box}), constr)$ with $V = \{v_0, v_1\}$, $F = \{(v_0, v_1)\}$, and $r = v_0$.


$\mathfrak{box}(v_0)$ with $constr(v_0)$
o1 : OBJECT(orders)
$|A| = 1$
$\mathfrak{box}(v_1)$ with $constr(v_1)$
e2 : EVENT(pay order)
E2O(e2, o1, ∗)
v_0
A
v_1

Selecting Interesting Bounds Next, we sketch three approaches to select the count interval bounds $c_{min}, c_{max} \in \mathbb{N}_0^\infty$ for given two types $t, t \in OT_L \cup ET_L$, a direction-mode $d \in \{n, r\}$, and fitness threshold $0 \leq p \leq 1$. All are based on selecting an initial small interval (i.e., $c_{min} = c_{max}$) and then increasing it to achieve the specified threshold p.

- *Centered around mean*: First, identify the mean $\bar{x} = \frac{1}{n} \cdot \sum_{i=1}^{n} x_i$ of the values $\#_L^d(t, t') = [x_1, \ldots, x_n]$. Next, select the minimal $r \in \mathbb{R}^{\geq 0}$ such that for the bounds $c_{min} = \max\{\lceil \bar{x} - r \rceil, 0\}$ and $c_{max} = \lfloor \bar{x} + r \rfloor$ it holds that $cfit_L(t, t', d, c_{min}, c_{max}) \geq p$.

- *Increasing from low*: Select the minimal $k \in \mathbb{N}_0$ such that for the bounds $c_{min} = 0$ and $c_{max} = k$ it holds that $cfit_L(t, t', d, c_{min}, c_{max}) \geq p$.

- *Decreasing from high*: Select the maximal $k \in \mathbb{N}_0$ such that for the bounds $c_{min} = k$ and $c_{max} = \infty$ it holds that $cfit_L(t, t', d, c_{min}, c_{max}) \geq p$.

Figure 4.6 shows a plot of an example count multiset, with intervals identified for all three selection approaches. In practice, not all identified intervals correspond to interesting count constraints. Filtering can be used to select only desired bounds. For example, filtering out interval bounds with perfect fitness for an event log with known undesired behavior.

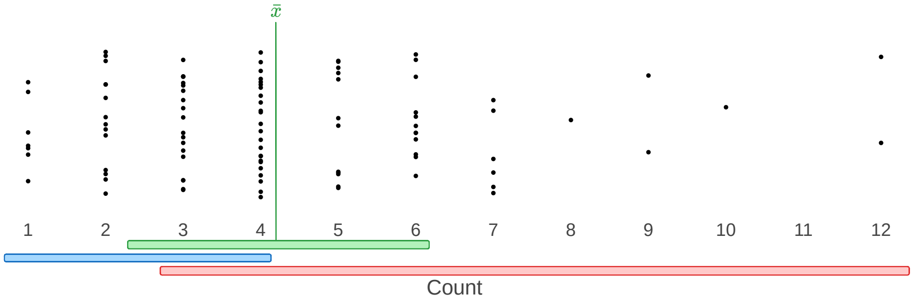


Figure 4.6: An example count multiset, sketching the identification of count bounds in multiple colors on the bottom. The multiset is shown as a scatter plot with randomized y values for visualization. The following count intervals can be derived based on the three identification approaches: *centered around mean* (in green): $c_{min} = 3$, $c_{max} = 6$; *increasing from low* (in blue): $c_{min} = 0$, $c_{max} = 4$; and *decreasing from high* (in red): $c_{min} = 4$, $c_{max} = \infty$.

4.5.2 Discovering Eventually-Follows Constraints

Next, we describe how to discover *eventually-follows constraints* based on an input OCED. It works similarly to the identification of count constraints: First gathering a multiset of the smallest durations between two event types both associated with a common object of a specified object type, and afterwards identifying a duration interval based on this multiset.

First, fix an object type $ot \in OT_L$ for which eventually-follows constraints should be mined. Then, for all combinations $et, et' \in ET_L$ of event types, we again construct a multiset of the observed values for each $o \in \text{inst}(ot)$. In particular, for each $e \in \text{inst}(et)$ with $o \in objs_L(e)$, the earliest event $e' \in \text{inst}(et)$ (w.r.t. $time_L(e')$) with $o \in obj_L(e')$ and $time_L(e) \geq time_L(e')$ as well as $e \neq e'$, if any, is identified and the duration difference $time_L(e') - time_L(e)$ (if there is such an e') or ∞ (if not) is added to the multiset for ot, et, et'. Formally, we first define the time difference $mdiff_L(o, e, et')$ to the earliest next event of a specified type $et' \in ET_L$ for a reference event $e \in E$ and an object $o \in O$. Based on this, we then introduce the function efm_L, which assigns a combination of types ot, et, et' to the corresponding eventually-follows time duration multiset.

$$\begin{aligned}
mdiff_L \colon & O \times E \times ET_L \to \mathbb{T}^{\infty} \\
mdiff_L(o, e, et') = & \min \big\{ time_L(e') - time_L(e) \quad \mid e' \in \text{inst}(et') \land o \in obj_L(e') \\
& \qquad \land time_L(e) \leq time_L(e') \big\} \\
& \text{where we define } \min \emptyset = \infty \text{ for convenience}
\end{aligned}$$

$$\begin{aligned}
efm_L \colon & OT_L \times ET_L \times ET_L \to \mathcal{B}(\mathbb{T}^{\infty}) \\
efm_L(ot, et, et') = & \left[mdiff_L(o, e, et') \;\middle|\; i \in \text{inst}(ot) \land e \in \text{inst}(et) \land o \in obj_L(e) \right]
\end{aligned}$$

Example 4.24 (EF Durations Multiset): Let L be an order management OCED. Consider ot = `orders` and the event types et = `confirm order` and et' = `payment reminder`. Again, assume that $\text{inst}_L(ot) = \{o_1, o_2, o_3, o_4\}$, each associated with exactly one event of type et, i.e., e_1, e_2, e_3, e_4, each. Additionally, assume that there are no events of type et' associated with o_1 and o_2, one event of type et', e_5, associated with o_1 and two events e_6

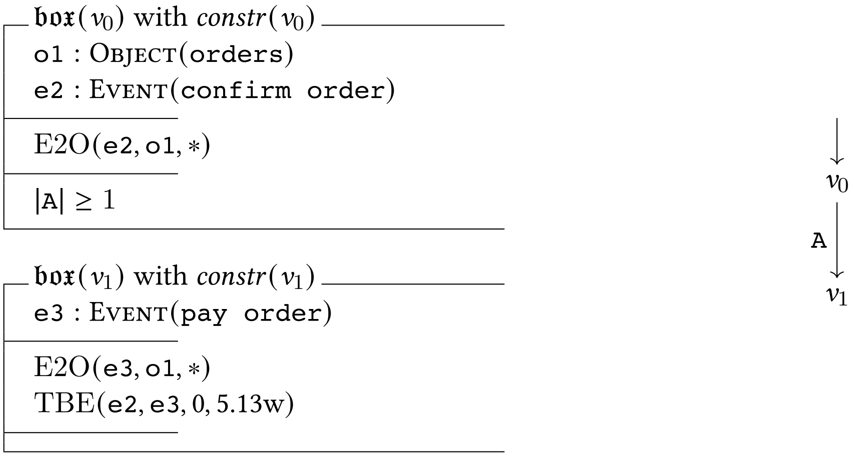
> and e_7 of type et' associated with o_1. Assume, that the time between events of interest are the following: $time_L(e_5) - time_L(e_3) = 2$ days, $time_L(e_6) - time_L(e_4) = 3$ days, and $time_L(e_7) - time_L(e_4) = 4$ days. Then $efm_L(ot, et, et') = [\infty, \infty, 2 \text{ days}, 3 \text{ days}]$. Notice, that for this example there is no event of type et' related to o_1 or o_2 after the e_1 and e_2 events of type et, respectively, leading to the ∞ values in the multiset.

Note that we again formally consider all combination of types $(ot, et, et') \in OT_L \times ET_L \times ET_L$. In practice, for at least some of these combinations the multiset is expected to be uninteresting (e.g., empty or only consisting of ∞). Based on the gathered multiset of delays, interval bounds are identified, similar to the count constraints presented before. However, for eventually-follows constraints we only consider the *Increasing from low* interval bound selection method, as commonly constraints specifying a maximum delay between events are of particular interest. Generally, the other selection methods could, of course, also be adapted for the eventually-follows use case.

The function $effit_L$, defined below, maps a type configuration $(ot, et, et') \in OT_L \times ET_L \times ET_L$ and a maximum duration bound $d_{max} \in \mathbb{T}$ to the fraction of fitting durations (i.e., that are $\leq d_{max}$). Given a fitness threshold $0 \leq p \leq 1$, an eventually-follows constraint candidate can then be constructed for $(ot, et, et') \in OT_L \times ET_L \times ET_L$ by identifying the smallest $d_{max} \in \mathbb{T}$ (if it exists) where $effit_L(ot, et, et', d_{max}) \geq p$.

$$effit_L(ot, et, et', d_{max}) = \frac{\left|[d \in efm_L(ot, et, et') \mid d \leq d_{max}]\right|}{|\, efm_L(ot, et, et')\, |}$$

Constructing a query tree constraint based on the identified parameters ot, et, et' and d_{max} is straightforward. In Example 4.25, we present an example construction.

> **Example 4.25 (Automatically Discover EF Constraint):** Let L again be an order management OCED. For the eventually-follows configuration ot = `orders`, et = `confirm order`, et' = `pay order` and $d_{max} \approx 5$ weeks. Below, we visualize the query tree constraints constructed for these values, $T = ((V, F, r, l, \mathfrak{box}), constr)$ with $V = \{v_0, v_1\}$ $F = \{(v_0, v_1)\}$ and $r = v_0$.
>
> 
>

4.5.3 Discovering Complex Constraints

So far, we presented approaches to mine simple count and eventually-follows constraints. Next, we present a way to combine such simpler constraints to more complex, interesting constraints. For instance, this allows discovering the disjunctions of subconstraints.

In essence, any combination of constraints can be combined, for instance through their disjunction. As an example, consider the OR-constraint “There are at least 1500 orders or at least 3000 items in total”, which is the disjunction of the constraints “There are at least 1500 orders in total” and “There are at least 3000 items in total”. Oftentimes, however, simple combination of full constraints are not of particular interest, as the combination is at the global (i.e., log) level. Instead, more interesting combined constraints are often specified for a shared input binding. For instance, consider the constraint “For every customer, there are at most three `payment reminder` events or at least one `ban customer` event”. Here, both subconstraints (“at most three `payment reminder` events” and “at least one `ban customer` event”) are based on the same input binding context (i.e., the “customer” object).

Such composited constraints with a common parent binding context, can be constructed based on two or more constraints with common object or event variables of the same type. In the following, we focus on composited constraints with exactly one shared object or event variable (with the same type), as they are simple to understand and construct.

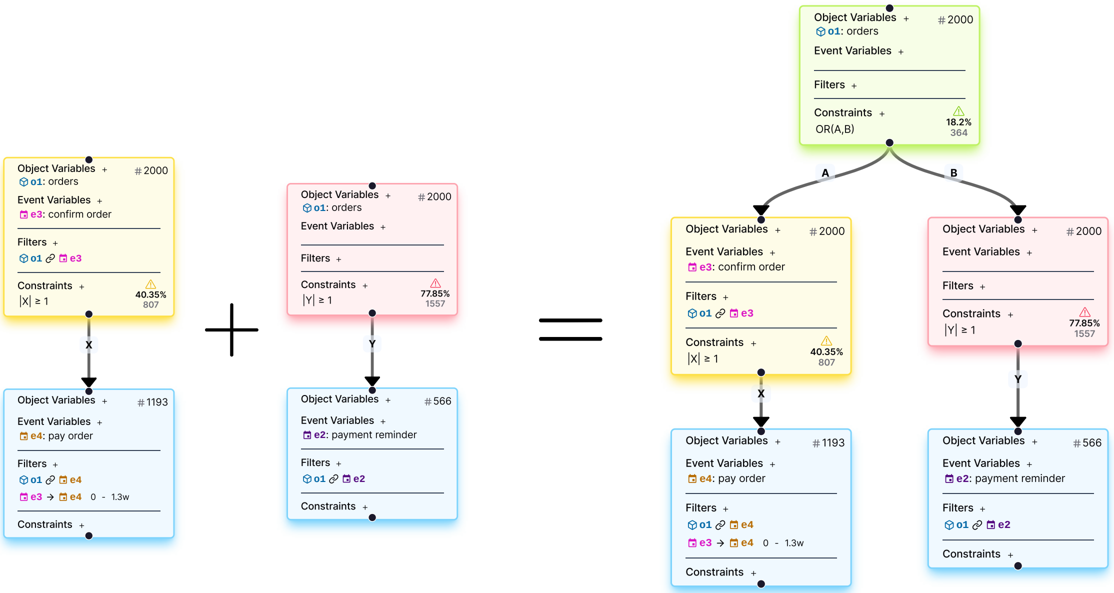


Figure 4.7: More complex constraints, like OR constructs, can be discovered by merging appropriate subconstraints (shown on the left) to a composed constraint (shown on the right). For that, the subconstraints should have a common variable assigned to the same object or event type, in this example the `o1` variable of type `orders`. Additionally, the composed constraint is only interesting and desirable to be discovered for certain subconstraints with some specific features, which are discussed later.

Discovering Composed OR Constraints Consider an event or object type $t \in OT_L \cup ET_L$ and assume a set of discovered (count and eventually-follows) constraints C rooted at this type. Let $c, c' \in C$ be two such query tree constraints with $c = ((V_c, F_c, r_c, l_c, \mathfrak{box}_c), constr_c)$ and $c' = ((V_{c'}, F_{c'}, r_{c'}, l_{c'}, \mathfrak{box}_{c'}), constr_{c'})$. In particular, we assume that there is a single variable $v \in \mathcal{U}_{obVar} \cup \mathcal{U}_{\mathrm{evVar}}$ assigned to the same types $ts \subseteq \mathcal{U}_{etype} \cup \mathcal{U}_{otype}$ by both c and c', i.e., $v \in \mathrm{dom}(\mathrm{Var}(\mathfrak{box}_c(r_c))) \cap \mathrm{dom}(\mathrm{Var}(\mathfrak{box}_{c'}(r_{c'})))$ with $ts = \mathrm{Var}(\mathfrak{box}_c(r_c))(v) = \mathrm{Var}(\mathfrak{box}_{c'}(r_{c'}))(v)$.

Before describing how to construct the combined OR-constraint in the form of a query tree constraints, we first discuss what combinations of constraints c and c' are likely to form a desirable OR-constraint. The following are factors that can be considered (e.g., using a threshold) for discovering interesting OR constraints:

- Good fitness: The combined OR-constraint should have good fitness, i.e., be satisfied for at least the specified fraction p of output bindings. For instance, commonly, only constraints with a fitness of at least 0.8 are considered.
- Better than independent: Let p_1 and p_2 be the satisfied fraction of c and c', respectively. Assuming that the constraints are completely independent, the disjunction is expected to be satisfied in $\hat{p} = 1-((1-p_1)\cdot(1-p_2)) = p_1+p_2-p_1\cdot p_2$ percent of bindings. Considering the fraction $p/\hat{p}$, the combined OR-constraint is interesting if it is satisfied more often than would be expected if both parts were independent, i.e., with larger fractions $p/\hat{p}$ (e.g., ≥ 1.1)

Similar factors or heuristics can also be established for other types of constraint combinations, not only OR-constructs. In Example 4.26, an OR constraint consisting of one count and one eventually-follows constraint is considered, which according to the presented factors is desireable to be discovered.

Example 4.26 (Interesting OR Constraint): Consider an order management OCED L and two constraints c and c', specifying that for an object of type `orders`, there should be:

- c: at least one event of type `pay order` within 4 weeks after every `confirm order` event associated with the order
- c': ≥ 1 events of type `payment reminder` associated with the order

Semantically, it can be assumed that these two constraints are not independent and would indeed be a good OR candidate. Assuming that c has a satisfaction ratio of $p_1 = 0.63$ and c' of $p_2 = 0.22$, the expected satisfaction of the OR, assuming they are independent, would be $0.63 + 0.22 - 0.63 \cdot 0.22 \approx 0.71$. Instead, the OR construct has a satisfaction ratio of 0.86, and thus with $0.86/0.71 \approx 1.21 \geq 1.1$, seems to indeed be an interesting OR candidate.

Of course, there are also other factors to consider. For instance, it is undesirable to discover obvious tautologies as OR constraints, as described in Example 4.27.

Example 4.27 (Uninteresting OR Constraint): Again, consider an order management OCED L and two count constraints c and c', specifying that for an object of type `orders`, there should be:

- c: ≤ 3 objects of type `items` associated with the order
- c': ≥ 4 objects of type `items` associated with the order

An OR construct of c and c' is clearly not very interesting, as it is always fulfilled.
Such candidates should be filtered out. For instance, by not considering two count constraints involving the same related object type. Additionally, if at least some noise in the log is assumed to exist, OR construct candidates with perfect fitness (i.e., $p = 1$) where the constraint part are totally disjoint ($p_1 = 1 - p_2$) could also be filtered out.

An example OR constraint specifying that "An order is either paid fast (i.e., within ≈ 1.3 weeks) after confirmation or at least one payment reminder is sent out" is shown in Figure 4.8, consisting of a combination of both count and eventually-follows subconstraints. This constraint was automatically discovered based on the order management OCED from [36].

Finally, we next describe how the combined OR query tree constraints can be constructed formally based on the constraint parts c and c'. The combined OR constraint from c and c' is then the query tree constraint $T = ((V, F, r, l, \mathfrak{box}), constr)_L$ with:

- $V = \{v_r\} \cup V_c \cup V_{c'}$
- $F = \{f_1, f_2\} \cup F_c \cup F_{c'}$ with $f_1 = (r, r_c)$ and $f_2 = (r, r_{c'})$
- $r = v_r$
- $l\colon F \to \mathcal{U}_{\text{setName}}$ with

$$l(f) = \begin{cases} l_c(f), & \text{if } f \in F_c \\ l_{c'}(f), & \text{if } f \in F_{c'} \\ \texttt{A}, & \text{if } f = f_1 \\ \texttt{B}, & \text{if } f = f_2 \end{cases}$$

- $\mathfrak{box}\colon V \to \mathfrak{BOX}_L$ where

$$\mathfrak{box}(v) = \begin{cases} \mathfrak{box}_c(v), & \text{if } v \in V_c \\ \mathfrak{box}_{c'}(v), & \text{if } v \in V_{c'} \\ \mathfrak{v}_r, & \text{otherwise, i.e., } v = v_r \end{cases}$$

with $\mathfrak{v}_r = (\{v \mapsto ts\}, \emptyset)$.

- $\forall_{u \in V}$ $constr\colon V \to \mathcal{P}(\mathbb{P}_L)$ with

$$constr(u) = \begin{cases} constr_c(u), & \text{if } u \in V_c \\ constr_{c'}(u), & \text{if } u \in V_{c'} \\ \{\text{OR ALL}(\{l(f_1), l(f_2)\})\}, & \text{if } u = v_r \end{cases}$$

$\mathfrak{box}(v_0)$ with $constr(v_0)$

$\texttt{o1} : \text{Object}(\texttt{orders})$

$\text{OR ALL}(\texttt{A}, \texttt{B})$

$\mathfrak{box}(v_1)$ with $constr(v_1)$

$\texttt{e3} : \text{Event}(\texttt{confirm order})$

$\text{E2O}(\texttt{e3}, \texttt{o1}, *)$

$|\texttt{X}| \geq 1$

$\mathfrak{box}(v_2)$ with $constr(v_2)$

$\texttt{e4} : \text{Event}(\texttt{pay order})$

$\text{E2O}(\texttt{e4}, \texttt{o1}, *)$
$\text{TBE}(\texttt{e3}, \texttt{e4}, 0, 1.3\text{w})$

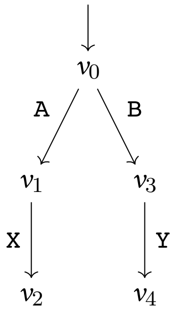
$\mathfrak{box}(v_3)$ with $constr(v_3)$

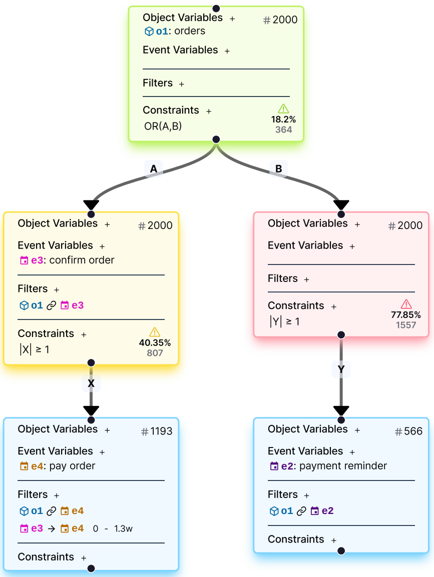
$|\texttt{Y}| \geq 1$

$\mathfrak{box}(v_4)$ with $constr(v_4)$

$\texttt{e2} : \text{Event}(\texttt{payment reminder})$

$\text{E2O}(\texttt{e2}, \texttt{o1}, *)$




Object Variables + #2000
o1: orders
Event Variables +
Filters +
Constraints +
OR(A,B)
18.2%
364

A
B

Object Variables + #2000
Event Variables +
e3: confirm order
Filters +
o1 e3
Constraints +
|X| ≥ 1
40.35%
807

Object Variables + #2000
Event Variables +
Filters +
Constraints +
|Y| ≥ 1
77.85%
1557

X
Y

Object Variables + #1193
Event Variables +
e4: pay order
Filters +
o1 e4
e3 → e4 0 - 1.3w
Constraints +

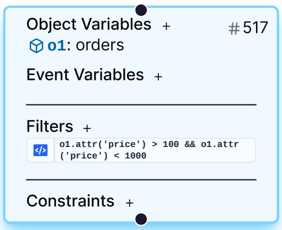
Object Variables + #566
Event Variables +
e2: payment reminder
Filters +
o1 e2
Constraints +

Figure 4.8: Automatically discovered OR-constraint based on an example order management OCED. Put into words, it specifies that an order should be either paid quickly after confirmation (i.e., within 9 days ≈ 1.3 weeks) or at least one payment reminder should be sent out for it.

4.6 Extensions

As the final part of this chapter, we present some extensions of the previously presented main approach in this section. For that, we consider an expression programming language, the so-called *Common Expression Language* (CEL)[1]. We first describe how, through the addition of binding predicates based on CEL, users can implement more advanced predicates themselves. Additionally, expanding on this idea also allows computing *Key Performance Indicators* (KPIs) or other annotations based on the basic querying approach we developed. In a more general sense, annotations of output bindings beyond the Boolean constraint violation status introduced in Section 4.3 are possible. Below we first shortly introduce both extension ideas before providing more details on them in the following pages.

1. **More Expressive Predicates**: First, we focus on a simple use case for a general expression language: Defining binding predicates. New predicate types are sketched, which allow including a CEL program with Boolean output. The predicate is then defined to be satisfied for an input binding exactly if the program outputs `true` for the given input binding as context. Through this simple idea, many more complex predicates can be easily defined by the user without advanced implementation efforts. For instance, in the example query shown below on the left, all bindings of `orders` objects which have a price between 100€ and 1000€ (excluding the bounds) are queried. This query cannot be expressed using the previous predicate types, but is easily modeled using a simple CEL program (shown separately on the right).



Object Variables + #517
o1: orders
Event Variables +

Filters +

```
o1.attr('price') > 100 && o1.attr
('price') < 1000
```

Constraints +

```
o1.attr('price') > 100
    && o1.attr('price') < 1000
```

2. **General Binding Annotations (e.g., KPIs)**: Next, we expand on this idea and additionally allow computing other types of values (e.g., numerical KPIs) using CEL programs. This is achieved by generalizing the concept of annotated output bindings, which were previously only considered in a Boolean format, indicating if the constraints were satisfied or violated. In this more general extensions, multiple annotation columns can be added iteratively, which allows computing KPIs and other values. For example, in the table shown below, each output binding (where `o1` is bound to customer objects) is annotated with two additional column values: `orderVolume`, which is the total order amount of the customer in the binding, and `customerClass`, which is derived from the order volume and indicates the customer rewards class.

`o1`	**`orderVolume`**	**`customerClass`**
o_1	400	`silver`
o_2	800	`gold`
o_3	600	`silver`
o_4	200	`bronze`

[1] See `https://github.com/google/cel-spec` and `https://cel.dev/`.

4.6.1 Further Increasing Predicate Expressiveness

The query and constraint approach presented so far is very expressive, as it considers bindings and can evaluate predicates on concrete constructed bindings. The collections of binding predicates presented in this thesis can also be extended, rendering the whole approach extensible and adaptable to different specific use cases. In theory, any computable function can be expressed using a binding predicate.

However, in practice, this extensibility is still a high barrier for most potential users. Moreover, oftentimes the predicates that users might want to express are not highly specific but very similar to the ones already expressible. Consider, for instance, a predicate filtering bindings such that the value assigned to an object variable should have a specified attribute (e.g., `price`) which value should be within a specified range with *open* bounds (e.g., > 100€ and < 1000€). In the defined attribute data predicates, representing open interval values is not easily possible. However, such predicates can be conveniently represented and modeled in CEL. For instance, assuming that the attribute value is available as the variable `price`, this predicate could be expressed as `price > 100 && price < 1000`. CEL is not Turing-complete, which on the one hand means that, at least in theory, it cannot express all computable functions. On the other hand, this limitation allows CEL to focus on performance and safety without any large expressiveness drawbacks in practice. CEL can also easily be extended and used from within a large variety of architectures and programming languages. We will not present CEL and the functionality exposed to the CEL programs in our approach formally, as this would be outside the scope of this thesis. Instead, we will semi-formally define two new types of predicates based on CEL programs and show examples highlighting how they can be used. The CEL programs inside the predicates will be executed, assuming to return a Boolean value of either `true` or `false`, signaling if the checked binding is satisfied for this predicate. As *input context*, the CEL programs have access to certain predefined variables, for instance corresponding to the object and event variables available in the binding that should be checked. In the following, we first present a basic CEL predicate to be used as a basic or attribute filter.

> **Definition 4.19 (Basic CEL Predicate):** A basic CEL predicate consists of a CEL program C which, given the variables and value of a specific binding as input context, returns either `true` or `false`. We then define $\mathrm{BasicCEL}(C) \in \mathbb{P}_L$ and for any binding $b \in \mathbb{B}_L$ it holds:
>
> $b \models \mathrm{BasicCEL}(C) \Leftrightarrow C$ returns `true` given the variable assignments of b as context

The variable assignment of a binding $b \in \mathbb{B}_L$ in a CEL program is represented as input context using same variable names. Different properties or attributes of the variable value can be accessed using special functions. For instance, the `attr` function is defined, which retrieves the object attribute value of events or objects for a specified attribute name. As there might be multiple different values at different timestamps for objects, this function simply returns the first value. The function `attrAt` can be used instead of objects, which additionally takes a timestamp at which the object attribute value should be retrieved. With these functions, many predicates not covered by the simple predicates presented in the main approach can already easily be expressed. For instance, the order price filter predicate mentioned before can be implemented as a basic CEL filter, with the CEL program shown in Code 4. Inside a query tree constraint node, basic CEL predicates are shown using a blue codeblock icon, as visible in Figure 4.9 on the left.

Some of the other types of functions defined for event and object variable values, are:

- `time`, which retrieves the timestamp of an event.
- `type`, which retrieves the object or event type of the variable value.

```
        o1.attr('price') > 100 && o1.attr('price') < 1000
```

Code 4: A simple CEL program as part of a basic CEL binding predicate. The attribute value `price` of the value from the object variable `o1` has to be within the range $(100, 1000)$ for this predicate to be satisfied for a binding.

Furthermore, there are general functions available which allow accessing all objects or events in the whole OCED or their count (i.e., `numEvents` and `numObjects` as well as `events` and `objects`). Common programming functions, like `count`, `map`, `avg`, and `filter` are also available in the CEL programs. Thus, also global predicates and constraints, independent of a specific variable binding can be modeled, for instance using the CEL programs shown in Code 5.

```
1       numObjects() >= 3000
2       size(objects().filter(x, x.type() == 'employees')) >
         ↪  size(objects().filter(x, x.type() == 'customers'))
```

Code 5: More CEL programs (one per line) that can be part of basic CEL binding predicates. In line 1: Evaluates to true if there are at least 3000 objects in the input OCED (i.e., independent of the considered variable binding). In line 2: Evaluates to true if there are more objects of type `employee` than objects of type `customers` in the input OCED (again, independent of the considered variable binding.)

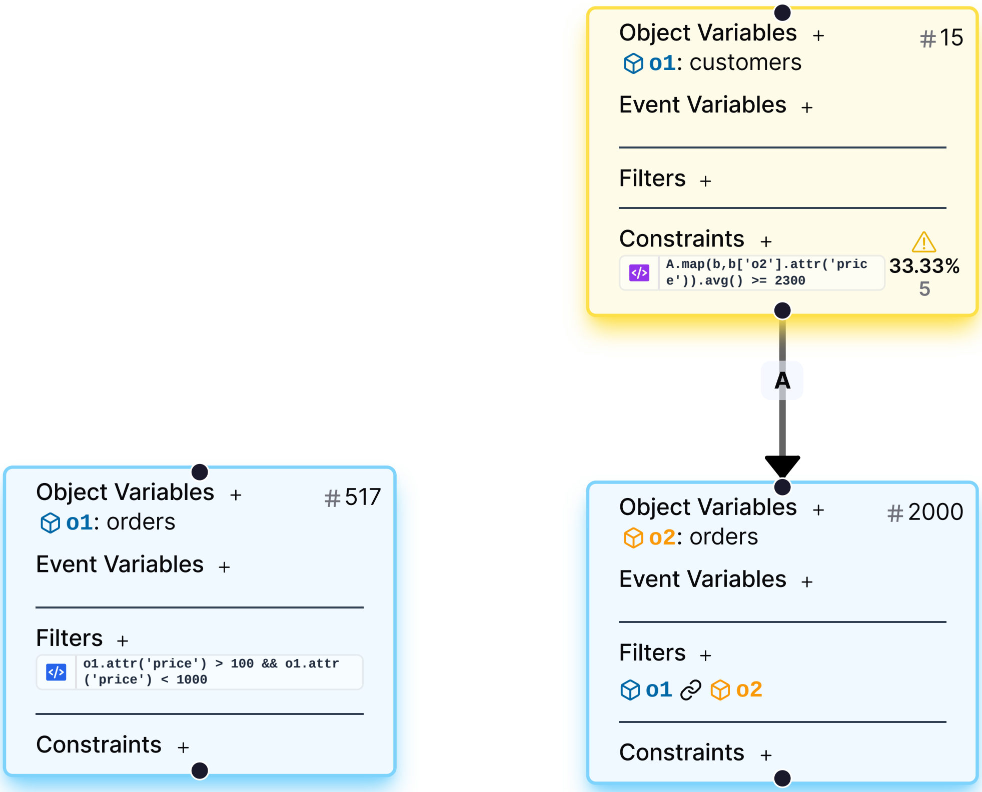
Next, we introduce another type of CEL binding predicates, which can access the set of labeled child bindings of all child nodes in addition to the binding variable values. Again, we underspecify the formalization of these advanced CEL predicates for simplicity.

Definition 4.20 (Advanced CEL Predicate): Let L be an OCED and let $T = ((V, F, r, l, \mathfrak{box}), constr)$ be a query tree constraint. For a node tree $u \in V$, advanced CEL predicates can be added based on a CEL program C, which, given the variables and their value in a specific binding, together with the labeled set of child bindings for each child of u as input context, returns either `true` or `false`. The set of labeled child bindings for all child nodes of u are represented by the set $S_{X,b} = \{b' \cup \{\texttt{satisfied} \mapsto b' \models constr(v)\} \mid b' \in out(\mathfrak{box}(v) \land b \sqsubseteq_L b')\}$ for all $(u, v) \in F$ with $l((u, v)) = X$, where $b' \models constr(v)$ is assumed to represent either the value `true` (satisfied) or `false` (violated). In the program C, the edge name can then be used to access this set of labeled child bindings of each child node of u. We define $\mathrm{AdvCEL}(C) \in \mathbb{P}_L$ and for any binding $b \in \mathbb{B}_L$ it holds:

$$\begin{aligned} b \models \mathrm{AdvCEL}(C) \Leftrightarrow {} & C \text{ returns } \texttt{true} \text{ given the variable assignments of } b \\ & \text{and the labeled child binding sets as input context} \end{aligned}$$

In the input context for advanced CEL predicates, all child bindings available as context are expanded with a variable `satisfied`, which is not an event or object variable, but instead has a value of either `true` or `false`, specifying if the original binding was satisfied or violated for the corresponding child node.

Such advanced CEL predicates can express very complex filters or constraints, involving a set of child bindings. For instance, using an advanced CEL predicate, one could specify that the average order price of a customer should be above 2500€. A graphical representation of a complete constraint containing such an advanced CEL predicate is shown on the right of Figure 4.9, where a purple code icon indicates that this CEL expression is part of an advanced CEL predicate.



```
A.map(b,b['o2'].attr('price')).avg() >= 2300
```

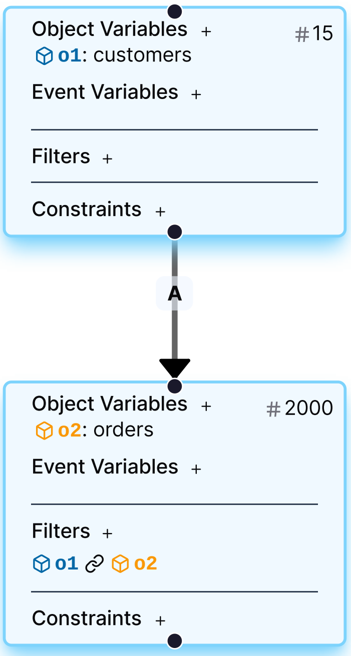
```
o1.attr('price') > 100 && o1.attr('price') < 1000
```

Figure 4.9: Two query trees containing simple or advanced CEL predicates. On the left, a simple query tree consisting of just one node is shown, which contains a basic CEL predicate containing the CEL expression from Code 4. This predicate is satisfied for bindings, where the order `o1` has a price between 100€ and 1000€ (excluding the bounds; at any point in time). On the right, an advanced CEL predicate is part of the root node. The advanced CEL predicate consists of the following CEL program *C*: `A.map(b,b['o2'].attr('price')).avg() >= 2300`. This constraint is satisfied for bindings of customers, where the average order price is at least 2300€.

4.6.2 General Binding Annotations

Apart from enabling the creation of customizable binding predicates, CEL can also be used to calculate arbitrary annotation values for bindings, for instance *Key Performance Indicators* (KPIs). In particular, for each node we could additionally allow specifying a CEL program which calculates a value (e.g., a numerical score) based on an output binding of the node, as well as the corresponding child bindings. For parent nodes, values could also be aggregated (e.g., by taking the average) across values calculated for the child output bindings of the nodes' children.

Formally defining general binding annotations for query trees is outside the scope of this thesis. However, we want to shortly sketch the basic idea. For that, consider the example shown in Figure 4.10. Annotated values are computed using CEL programs based on an input binding and are then added as additional columns to the output binding table. Similar to the CEL-based predicates, complex CEL programs can also use sets of child bindings, for instance to compute the sum of all orders by a customer.



Annotated Output Table of Root Node

o1	**orderVolume**	**customerClass**
o_1	400	silver
o_2	800	gold
o_3	600	silver
o_4	200	bronze

```
A.map(b,b['o2'].attr('price')).sum()
```

CEL program for `orderVolume`

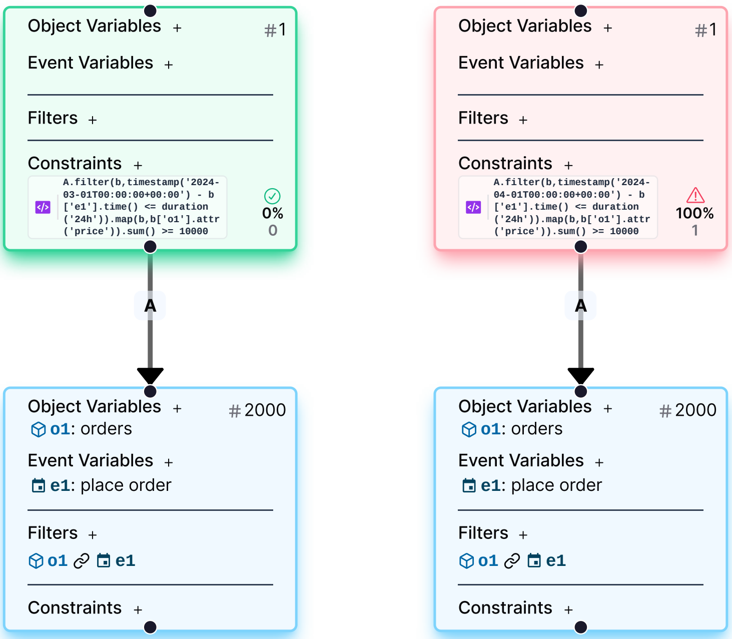
```
orderVolume < 300 ? 'bronze' :
   (orderVolume < 800 ? 'silver' : 'gold')
```

CEL program for `customerClass`.

Figure 4.10: Example of general binding annotations. On the left, a simple query tree is shown, consisting of two nodes which first query customer objects as `o1` and then order objects placed by the customer as `o2`. On the top right, the annotated output table of the root node is shown, where for each output binding assigning `o1` to a customer, two additional values are computed and annotated: `orderVolume` and `customerClass` (shown in bold). Below the table, the CEL programs which compute the corresponding annotated value based on an output binding are shown. The CEL program for `orderVolume` is akin to the average price CEL predicate presented previously. In particular, all child bindings (referred to as `A` as per edge label) of orders by the given customer are mapped to the price of the order which is then summed up. For `customerClass` two ternary operators (of the form `condition ? 'value when true' : 'value when false'`) are used to classify order volumes in different customer classes.

Generally, binding annotations can be defined iteratively and thus build on top of each other. For instance, in the example shown in Figure 4.10, `customerClass` is calculated based on `orderVolume`. In this context, the violation status of a node can also be represented as simply a Boolean annotation of bindings.

Finally, we also demonstrate how constraints based on KPIs can also be added using only the CEL predicates introduced in the previously presented extension. In particular, these constraints can impose that a certain KPI should be within a specified range. Figure 4.11 shows such an example constraint, which defines a target value for the total order volume within the last 24 hours, and is violated if this target KPI value is not met.



```
A.filter(b,timestamp('2024-
03-01T00:00:00+00:00') - b
['e1'].time() <= duration
('24h')).map(b,b['o1'].attr
('price')).sum() >= 10000
```

```
A.filter(b,timestamp('2024-
04-01T00:00:00+00:00') - b
['e1'].time() <= duration
('24h')).map(b,b['o1'].attr
('price')).sum() >= 10000
```

```
A.filter(b,timestamp('...') - b['e1'].time()  <= duration('24h'))
↪  .map(b,b['o1'].attr('price')).sum() >= 10000
```

Figure 4.11: Constraints based on a KPI of total order volume, considering all orders placed in the last 24 hours of a given timestamp, which is assumed to correspond to "now". The base CEL script is shown below the constraints. It filters all bindings `b` of the child node, in this case corresponding to all placed orders, based on whether the placement date is within 24 hours of the provided timestamp. Afterwards, the price of all these orders is summed up. The constraints are satisfied, if the total price sum of all orders placed within the last 24 hours is at least 10000€. The constraint on the left, assuming the first of March as a timestamp, is satisfied, while the one on the right, based on the first of April is violated.

As observable in Figure 4.11, such constraints can quickly become unreadable. However, this could be improved by iteratively introducing annotations of bindings which can build on top of each other. In the shown example, it would, for instance, make sense to first introduce the total order volume as an annotation and then define predicates based on the computed annotated value.

Chapter 5

Implementation

In this chapter, we describe our implementation of the proposed approach, addressing the research goals **RG5** (i.e., implementation of a graphical tool for designing and executing queries or constraints) and **RG6** (i.e., implementing the previously described discovery approach). We implemented a full-stack software tool *OCPQ*, consisting of a backend for efficient query execution and a frontend for designing and managing queries and constraints. The resulting contributions presented in this chapter are **CT4** (i.e., the main graphical tool) and **CT6** (i.e., the Rust-based OCEL 2.0 JSON and XML importers). The *OCPQ* tool is available and can be downloaded at `https://github.com/aarkue/OCPQ`. In the following, we first provide an overview of the architecture of our developed tool. Next, we detail the implementation of the *Execution Engine* for evaluating queries and constraints. In particular, we cover its functionality and techniques applied for achieving high performance. Finally, we present the graphical user interface of our tool and its features, aiming to make modeling and evaluating queries and constraints accessible also to non-technical users.

5.1 Overview and Architecture

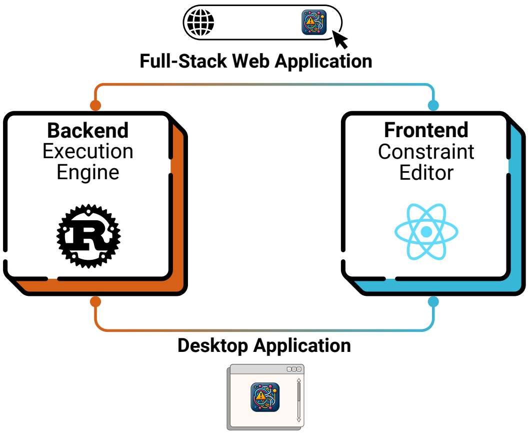


Figure 5.1: Overview of our implementation approach: The two main components are an efficient execution engine implemented in Rust and a user-friendly graphical constraint editor implemented in TypeScript React. This flexible architecture allows for deploying and using the tool as a full-stack web application or an installed desktop application.

An overview of our implementation components is shown in Figure 5.1. The two main parts, the performance-focused execution engine backend and the graphical editor frontend, are largely implemented independently. Common exchange formats, like a JSON-representation of the *Query Tree Constraints* introduced in Chapter 4, allow backend and frontend to communicate with each other. For instance, when a user finishes designing a constraint in the editor, its JSON-representation can be sent to the backend to evaluate the constraint for the currently loaded OCED.

Defining clear interfaces between the two components enables very flexible deployment methods: The tool can be used as a full-stack web application or an installable desktop application. In the full-stack web application scenario, the execution engine backend is served by a server while (multiple) users can access the frontend as a website on their computers. Of course, both backend and frontend can also be hosted on the same machine. Alternatively, end users can also install the tool as a desktop application locally, where both the execution engine and the editor frontend are integrated.

For technical reasons, there might be some special functionality implemented specially for one of the deployment modes. For instance, the full-stack web application should allow users to upload an OCEL 2.0 to the backend, while for the desktop application showing a native file selector to choose an OCEL 2.0 file should be possible.

Next, we first present further details about the implementation of the execution engine, focusing on the implemented features and performance characteristics.

5.2 Execution Engine

The execution engine is implemented in the programming language Rust and focuses on efficiently computing query or constraint results. Its functionality can be mainly differentiated into the following parts:

- OCED Import
- Preprocessing of OCED
- Query Tree Constraints & Query Evaluation
- Discovery of Constraints

OCED Import As part of this thesis, we implemented import algorithms for OCED based on the OCEL 2.0 standard (see [2]) in the XML and JSON versions. They were contributed upstream to the Rust4PM project[1], a Rust library for process mining, which we introduced in [39].

Preprocessing of OCED To enable fast evaluation of multiple queries based on the same input OCED, the backend allows loading an active OCED which is preprocessed directly after loading. In OCEL 2.0 objects and events are linked based on their identifiers. However, when expanding a huge numbers of bindings, looking up events or objects by identifier impacts performance negatively. Recall, that the expansion of bindings is largely based on the object-to-object and event-to-object relationships, as described in Section 4.4. Thus, to enable faster expansion of input bindings, the OCED is processed to reference objects and events by an index instead. The

[1] `https://github.com/aarkue/rust4pm`

relations in the OCED are also extracted in a way that, given an object or event index, the related objects and events can easily be identified. An additional advantage of the index-based reference to events and objects is reduced memory usage: Storing a single variable assignment requires at least the size of the variable name and event or object identifier, plus some additional bytes used for memory management. Storing both the variable identifier and object or event identifiers, as indices instead requires, on 64-bit architectures, only 8 bytes each and thus 16 bytes total. Depending on the length of used string identifiers, this significantly reduces the overall memory usage and also enables constructing all output bindings for queries which would otherwise not fit into system memory. We also represent object and event variables as numbers. For example, the object variable `o1` would be represented as $\mathrm{O}(0)$. In that, one bit (represented as either O or E, for object and event variables, respectively) specifies that it is an object variable and the integer 0 is the zero-based variable index. A variable binding is then represented as $\{\mathrm{O}(0) \mapsto 1, \mathrm{O}(1) \mapsto 3, \mathrm{E}(0) \mapsto 12\}$ through this index-based approach, where 1 and 3 are the object-indices of the variable values of the object variables `o1` and `o2`, respectively, and 12 is the event-index of the variable value of the event variable `e1`.

Query Tree Constraint & Query Evaluation Our query and constraint method presented in Chapter 4 (specifically Section 4.2 and Section 4.3), is implemented using different data structures corresponding to the introduced concepts of *bindings*, different types of *predicates*, *binding boxes* and *query tree constraints.* For nodes of a query tree constraint, an evaluation function is implemented according to the recursive algorithm approach sketched in Section 4.4. The expansion of bindings as well as the recursive evaluation calls and the evaluation of predicates is parallelized, which further increases speeds. For the expansion of bindings, our implementation follows the intelligent binding construction approach presented in Section 4.4. In particular, a sequence of binding and filter steps are constructed. Initially, for all new variables a simple step binding the variable to all possible values is added, and all filter predicate are added as steps at the end. Next, based on how variables can be bound on other variables, the binding order is updated and steps are replaced to bind variable values based on already bound variables. Filter predicates which are automatically fulfilled based on their binding step are removed. Additionally, the remaining filter predicates are moved up to be evaluated as soon as all involved variables are bound.

Discovery of Constraints The backend also allows automatically discovering query tree constraints based on input OCED, as described in Section 4.5. However, the approach is optimized for runtime and uses the pre-computed symmetric relationship graph of the OCED to quickly discover count and eventually-follows constraints. Additionally, OR constraints are not constructed based on just combining all discovered constraints. Instead, a sample of input bindings (i.e., binding the object type of interest) is constructed and a count or eventually-follows constraints are discovered for this sample, such that it is not satisfied in (nearly) all the bindings. Next, the bindings for which this constraint is *not* satisfied are considered and used to discover a second count or eventually-follows constraint which is satisfied for these bindings. As a result, combining these constraints often yields interesting OR constructs, which are also primarily exclusively satisfied (i.e., would also be a good exclusive choice XOR construct).

Next, we will present the implemented user interface, detailing how to use it and what features are supported.

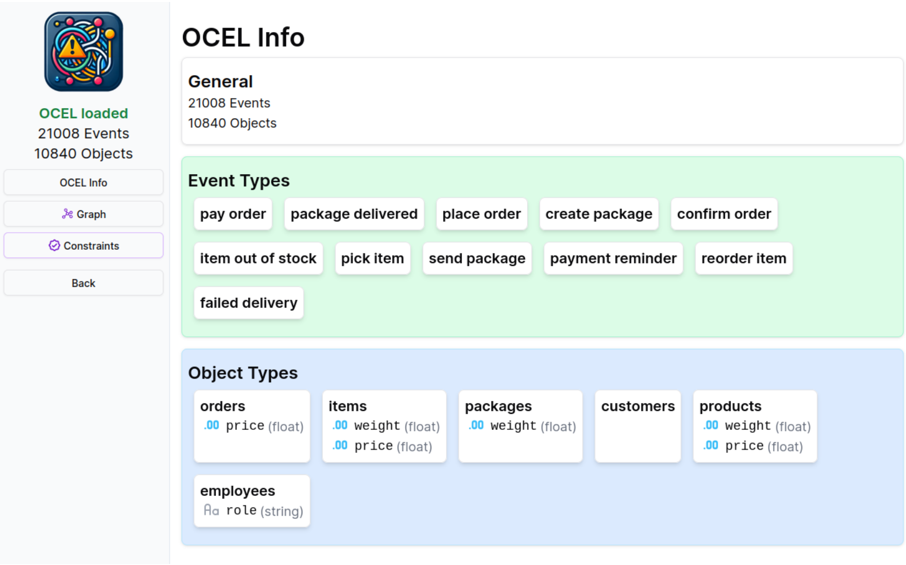
5.3 User Interface

The graphical user frontend is the main way users can design and evaluate constraints and queries. As such, the frontend should support the following functionalities:

- Load OCEL 2.0 files and display log information
- Manage and organize multiple constraints
- Design constraints and queries in interactive editor
- Execute queries and constraints and view the resulting output
- Configurable auto-discovery

Next, we present each of these features individually together with corresponding screenshots of the tool interface.

Loading OCEL 2.0 Initially, when no OCED is loaded, the frontend prompts to select an OCEL 2.0 file to load. Depending on the setup, a file can either be selected from pre-defined options or a custom OCEL 2.0 XML or JSON file can be selected or uploaded from the user's machine. After the OCED is loaded and pre-processed, some basic information, like the number of events and objects or the contained event and object types, are shown (see Figure 5.2).



OCEL Info

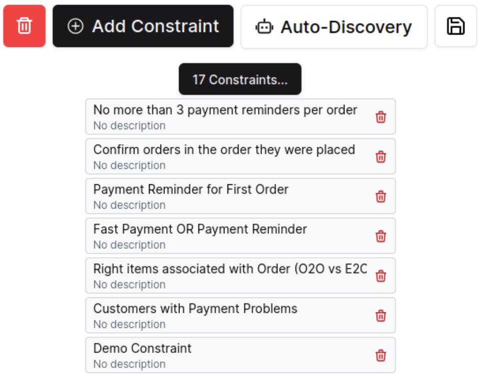
General
21008 Events
10840 Objects

Event Types
pay order
package delivered
place order
create package
confirm order
item out of stock
pick item
send package
payment reminder
reorder item
failed delivery

Object Types
orders
.00 price (float)
items
.00 weight (float)
.00 price (float)
packages
.00 weight (float)
customers
products
.00 weight (float)
.00 price (float)
employees
Aa role (string)

Figure 5.2: Screenshot of the displayed information about the currently loaded OCEL 2.0 file in the tool frontend.

Constraint Management After an OCED is loaded, the constraint and query overview can be accessed using the corresponding button in the menu on the left. As shown in Figure 5.3, the constraint overview shows a list of all saved constraints together with their name and description. Additionally, a button at the top allows adding new constraints. Individual constraints can also be selected by clicking on them. Furthermore, constraints can also be deleted from this overview using the corresponding buttons.



No more than 3 payment reminders per order
No description

Confirm orders in the order they were placed
No description

Payment Reminder for First Order
No description

Fast Payment OR Payment Reminder
No description

Right items associated with Order (O2O vs E2C
No description

Customers with Payment Problems
No description

Demo Constraint
No description

Figure 5.3: A screenshot of the tool frontend showing an overview over the locally saved constraints, each of which is displayed with its corresponding title and description.

Constraint & Query Editor Once a new constraint was added or an existing one selected, details of the constraint are shown. A screenshot of this view is shown in Figure 5.4. At the top, the title, and description of the constraint can be updated. Additionally, a quick selector allows selecting one of the other saved constraints. On the bottom, the interactive editor allows designing the corresponding query tree constraint.

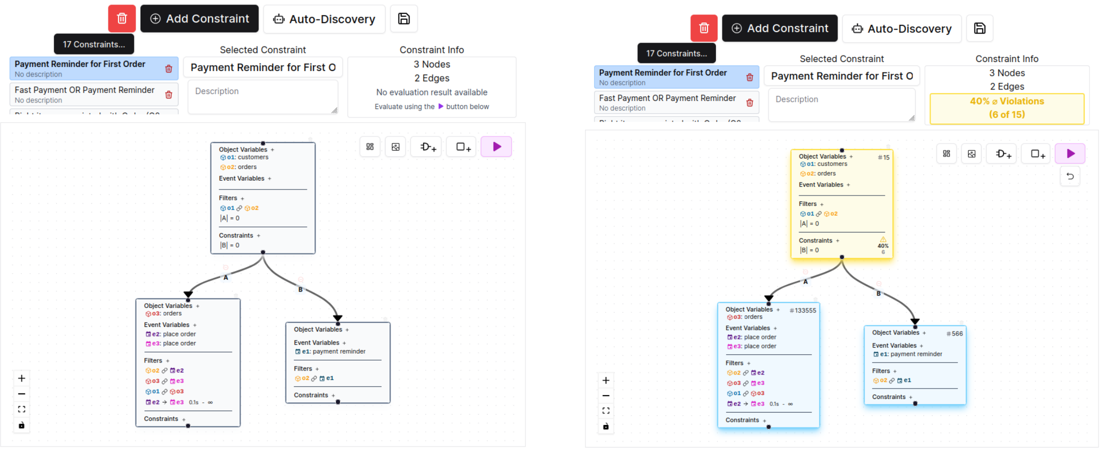


Figure 5.4: Two screenshots of the same constraint shown inside the editor: On the left, without evaluation results (i.e., before the constraint was evaluated) and on the right with evaluation results. On the right, the individual tree nodes show the number of bindings and violations, and are colored corresponding to their violation percentage.

In the query tree constraint editor, new nodes can be added using the plus buttons on the top right. Existing nodes can also be modified or removed. To connect a node to an existing query tree, a new edge can be created by dragging the source connector (which are on the top and bottom of nodes) to the target connector. Just like in the previously shown formal notation, variables and predicates are only shown in nodes where they are introduced and are omitted from the children. In particular, this means that by connecting a parent node with a new child, all variables and predicates (in $\mathbf{BASIC}_{\mathrm{L}}$ and $\mathbf{ATTRS}_{L}$) are automatically assumed to now also be present in the child, without the need to manually add them. Thus, query tree constraints involving multiple nodes can easily be constructed without the overhead of manually fulfilling the formal requirements. In Figure 5.5 some of the dialogs that open to allow modifying the predicates or variables of a binding box node are presented. These dialogs can be accessed by clicking on the plus icons to add new variables or predicates, or clicking on already existing variables and predicates to modify existing ones. The editor allows modeling multiple query tree constraints for the same constraint, which are simply considered and evaluated separately. The save button on the top right of the whole frontend can be used to save all modeled constraints, together with their name and description, locally on the users machine. They are then automatically loaded when accessing the frontend.

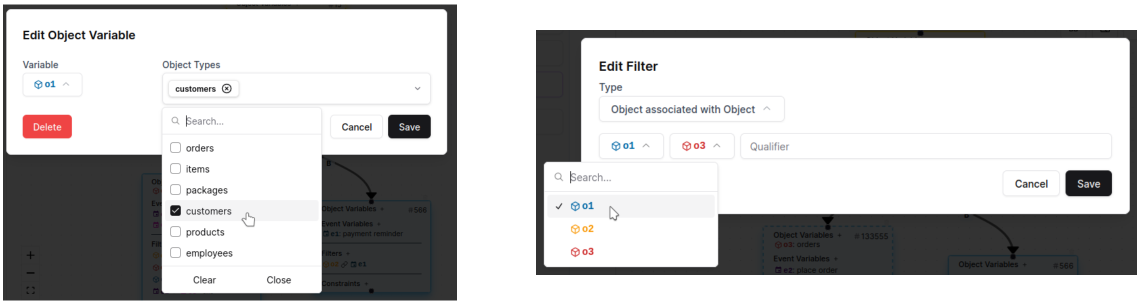
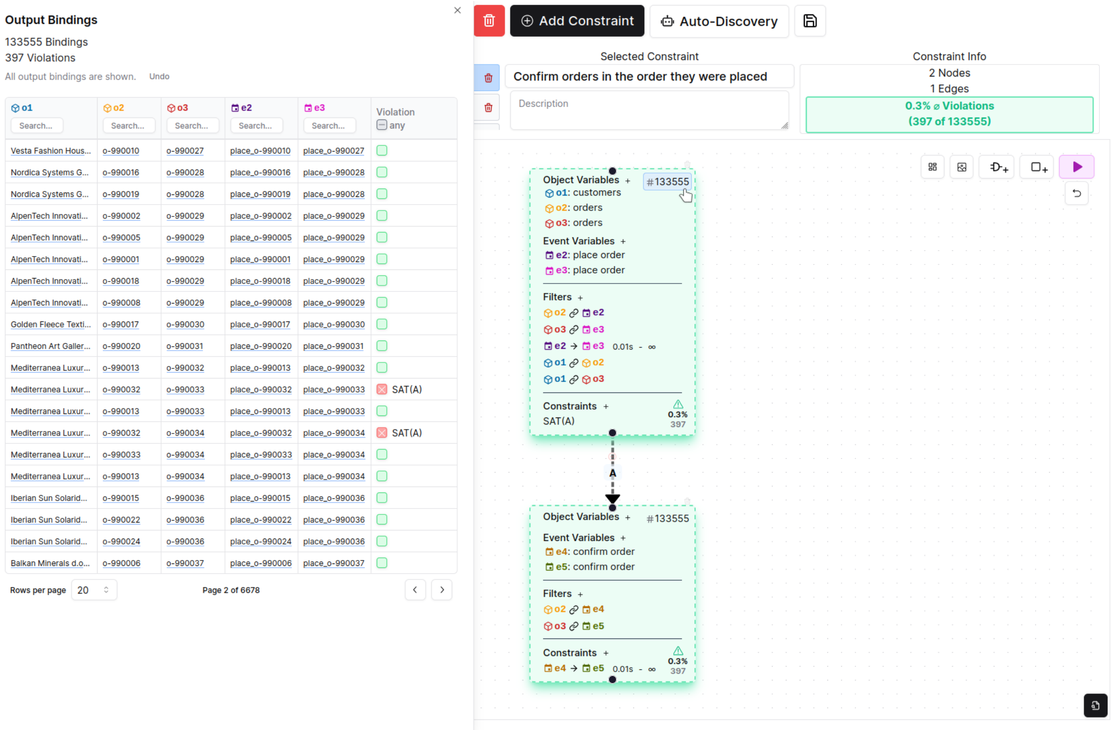


Figure 5.5: Screenshots of different edit dialogs for the object variables and filter predicate of a node in the frontend.

Execution & Viewing Results After the query tree constraint is complete, it can be evaluated by clicking on the play button on the top right. A JSON representation of the modeled query tree constraint is then transmitted to the execution engine, where it is evaluated. When the backend finishes the evaluation, the results, i.e., all labeled output bindings, are sent back to the frontend and displayed there, as shown in Figure 5.4. The violation percentage of output bindings for each node is used to color it, ranging from green (no violations) to red (high percentage of violations). Nodes which only query and do not contain constraints (i.e., $v \in V$ where $constr(v) = \emptyset$) are simply shown in blue. For each node, the total number of output bindings is shown on the top right and the number of violated output binding as well as the violation percentage is shown on the bottom right. The individual labeled output bindings of a node can also be viewed in more detail by clicking on the violation percentage indicator, which opens a panel with further details on the side.

Advanced Editor Features The query tree constraint editor has some advanced features which supports users in modeling complex constraints. Tree nodes and edges can be selected by either clicking on them (while holding the control key when selecting multiple elements) or by holding shift and dragging a rectangular selection with the mouse. Selected elements can be copied and pasted using the conventional keyboard shortcuts (i.e., Ctrl+c and Ctr+v). In Figure 5.7a the query tree constraint on the left was selected, copied and pasted to the right, resulting in the duplication of the tree on the right, which is currently selected in the screenshot. Additionally, automatic layouting can be applied to either all nodes or only selected nodes using the *Auto layout* button on the top right of the editor. Figure 5.7b shows the node editor content before and after applying automatic layouting.

CEL Script Editor We also implemented the predicate extension based on Common Expression Language (CEL) scripts we introduced in Subsection 4.6.1. Apart from the implementation



Output Bindings

133555 Bindings
397 Violations

All output bindings are shown. Undo

o1	o2	o3	e2	e3	Violation any
Vesta Fashion Hous...	o-990010	o-990027	place_o-990010	place_o-990027	□
Nordica Systems G...	o-990016	o-990028	place_o-990016	place_o-990028	□
Nordica Systems G...	o-990019	o-990028	place_o-990019	place_o-990028	□
AlpenTech Innovati...	o-990002	o-990029	place_o-990002	place_o-990029	□
AlpenTech Innovati...	o-990005	o-990029	place_o-990005	place_o-990029	□
AlpenTech Innovati...	o-990001	o-990029	place_o-990001	place_o-990029	□
AlpenTech Innovati...	o-990018	o-990029	place_o-990018	place_o-990029	□
AlpenTech Innovati...	o-990008	o-990029	place_o-990008	place_o-990029	□
Golden Fleece Texti...	o-990017	o-990030	place_o-990017	place_o-990030	□
Pantheon Art Galler...	o-990020	o-990031	place_o-990020	place_o-990031	□
Mediterranea Luxur...	o-990013	o-990032	place_o-990013	place_o-990032	□
Mediterranea Luxur...	o-990032	o-990033	place_o-990032	place_o-990033	☒ SAT(A)
Mediterranea Luxur...	o-990013	o-990033	place_o-990013	place_o-990033	□
Mediterranea Luxur...	o-990032	o-990034	place_o-990032	place_o-990034	☒ SAT(A)
Mediterranea Luxur...	o-990033	o-990034	place_o-990033	place_o-990034	□
Mediterranea Luxur...	o-990013	o-990034	place_o-990013	place_o-990034	□
Iberian Sun Solarid...	o-990015	o-990036	place_o-990015	place_o-990036	□
Iberian Sun Solarid...	o-990022	o-990036	place_o-990022	place_o-990036	□
Iberian Sun Solarid...	o-990024	o-990036	place_o-990024	place_o-990036	□
Balkan Minerals d.o...	o-990006	o-990037	place_o-990006	place_o-990037	□

Rows per page 20 Page 2 of 6678

Selected Constraint: Confirm orders in the order they were placed

Description

Constraint Info
2 Nodes
1 Edges
0.3% ⌀ Violations
(397 of 133555)

Object Variables + #133555
o1: customers
o2: orders
o3: orders
Event Variables +
e2: place order
e3: place order
Filters +
o2 🔗 e2
o3 🔗 e3
e2 → e3 0.01s - ∞
o1 🔗 o2
o1 🔗 o3
Constraints +
SAT(A) 0.3% 397

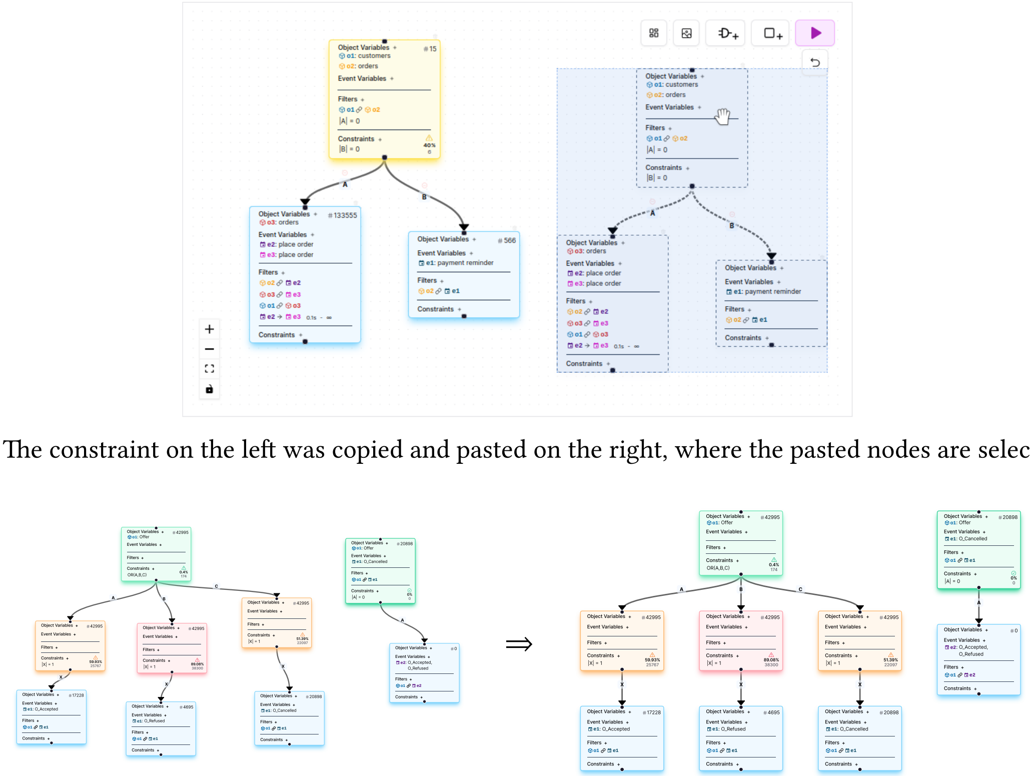
Object Variables + #133555
Event Variables +
e4: confirm order
e5: confirm order
Filters +
o2 🔗 e4
o3 🔗 e5
Constraints +
e4 → e5 0.01s - ∞ 0.3% 397

Figure 5.6: A screenshot showing the evaluation results for the selected constraint node inside the tool UI. On the results panel on the left, the output binding table corresponding to the node is shown. In the last column, the violation status is shown, together with the first encountered violated constraint predicate (if any). The table can be searched and filtered, for example, to only show violated or satisfied output bindings.

in the execution engine, the frontend features a interactive code editor with auto-complete, syntax highlighting, and function signature hints. We implemented these editor features as custom extensions of the Monaco editor[2], which is a very popular browser-based code editor. These advanced CEL-related features allow users to easily start writing CEL scripts. Moreover, they also allow non-programmers to model simple to more complex predicates in CEL. In Figure 5.8, a screenshot of the editor is shown, where a function description is currently displayed.

Automatic Discovery Using a button at the top, the automatic discovery approach presented before can be executed. Multiple discovery parameters, like the event or object types to discover constraints for, or the minimum fitness fraction of the discovered constraints, can be configured. The discovered constraints are afterwards added to the constraint list in the frontend and can then be deleted and modified, just like manually constructed constraints.

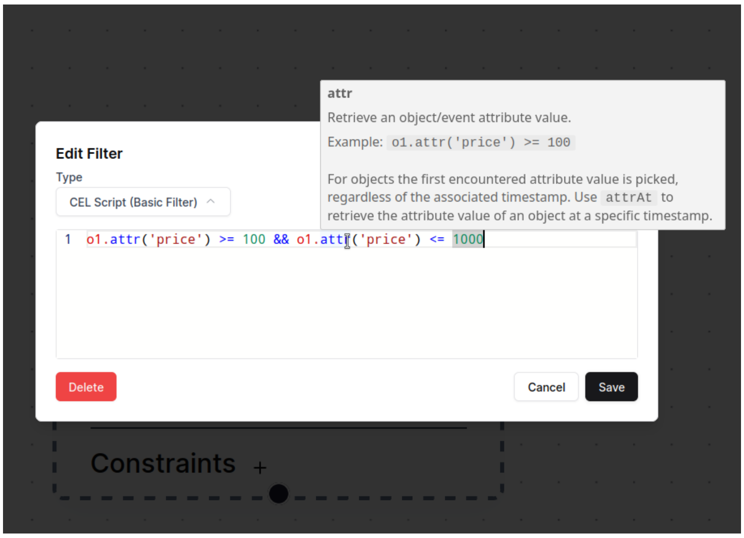
[2] See `https://github.com/microsoft/monaco-editor`.



(a) The constraint on the left was copied and pasted on the right, where the pasted nodes are selected.

(b) Two constraints inside the editor before (on the left) and after (on the right) automatic layouting.

Figure 5.7: Screenshots showing some of the advanced editor features: Copy-and-paste support and automatic layouting.



attr

Retrieve an object/event attribute value.

Example: `o1.attr('price') >= 100`

For objects the first encountered attribute value is picked, regardless of the associated timestamp. Use `attrAt` to retrieve the attribute value of an object at a specific timestamp.

Edit Filter

Type

CEL Script (Basic Filter)

```
o1.attr('price') >= 100 && o1.attr('price') <= 1000
```

Figure 5.8: A screenshot showing the CEL script editor for implementing advanced predicates in the frontend UI.

Chapter 6

Evaluation

In this chapter, we evaluate the proposed querying and constraint approach. As such, it addresses the research question **RQ3** and fulfills research goals **RG7** and corresponds to our contribution **CT5**. We perform two types of evaluations, focusing on the expressiveness of the approach and the runtime:

Qualitative Analysis We present some example query and constraint formulations in natural language and then demonstrate how they can be expressed using our approach. Some of the example constraints presented in the *Qualitative Analysis* were also automatically discovered, serving as examples of how high-quality constraints can be discovered automatically based on OCED.

Performance Analysis We investigate the runtime of the implemented execution engine when evaluating queries and constraints on input OCED. In the *Performance Analysis*, we investigate the execution time when evaluating some of the presented example constraints and queries. Additionally, we also investigate the expected runtime more systematically to identify the scalability of our approach and implementation.

Finally, we also discuss threats to the validity of the evaluation performed in this chapter. Before presenting the evaluation results, we first describe our experimental setup and introduce the used OCED as well as our hardware setup.

6.1 Experimental Setup

6.1.1 Datasets

We evaluated our approach on a variety of the publicly available OCEL 2.0 files, an overview over which is shown in Table 6.1. Most of these are simulated example logs sourced from `https://www.ocel-standard.org`. As a notable exception, we also include an OCEL 2.0 version of the BPI Challenge 2017 event log, originally distributed in the XES format as a flat, traditional event log in [30]. The BPI Challenge 2017 is significantly larger than the other considered OCEDs and is derived from a real-life loan application process of a Dutch financial institute [30]. As the log involves *applications*, *offers*, and *workflows*, it was also in previous analysis often considered only in parts. An object-centric version of the log is available from [40], which we converted to the OCEL 2.0 standard file format, additionally adding object-to-object relationships.

Table 6.1: An overview of the datasets used for the qualitative and quantitative evaluation.

Name	#Events	#Objects	#Event Types	#Object Types	Reference
Procure-to-Pay	14671	9543	10	7	[41]
Order Management	21008	10840	11	6	[36]
Container Logistics	35413	13910	14	7	[42]
BPI Challenge 2017 OCED	1202267	106162	26	4	[30, 40]

Apart from the expressiveness of the approach, we also investigate the runtime of queries and constraints in the quantitative analysis. As such, the size of the considered OCED is of particular interest. In Figure 6.1, the number of events and objects inside each considered log are presented in a logarithmic scatter plot. The number of events and objects inside the simulated OCEDs mostly lay in the 10,000 – 50,000 range, while the BPI Challenge 2017 OCED contains over one million (1,000,000) events and one hundred thousand (100,000) objects.

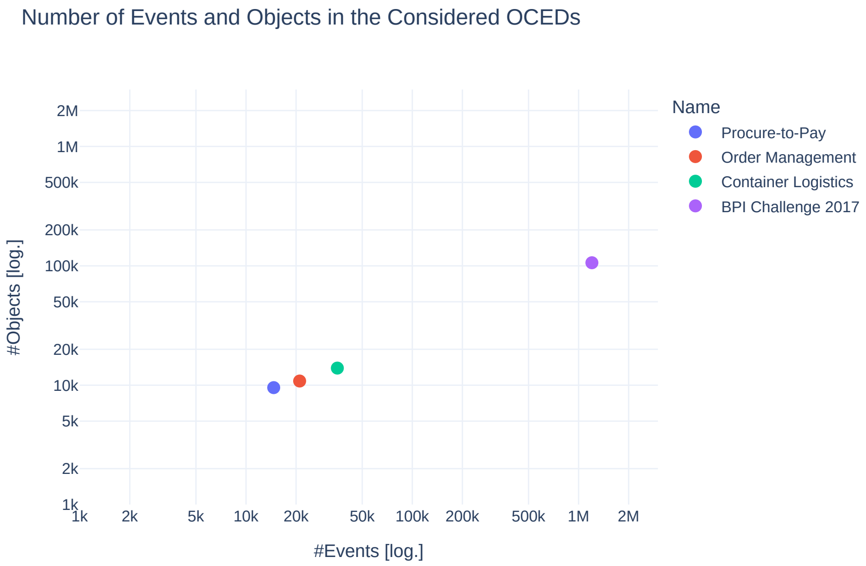


Figure 6.1: A logarithmic scatter plot showing the numbers of events and objects inside the considered dataset. Notably, the BPI Challenge 2017 is by far the largest OCED with more than one million events.

6.1.2 Hardware

All of our execution time performance measurements were done on a laptop equipped with an AMD Ryzen 9 5900HX CPU (8 cores/16 threads) and 32 GB of memory. To account for expected variations in execution speed, we repeated all measurements ten times and plotted all results together with their mean.

6.2 Qualitative Analysis

In this section, we present example constraints and queries implemented using our approach. They are organized according to the OCED used for the presented query or constraint. For each of them, we also give a textual formulation of the constraints, mention the evaluation results and contextualize the type of considered constraint or query regarding the related work presented in Chapter 2. We also segment the textual formulation to indicate which segments correspond to a subquery part which we name explicitly, for instance, as $Q1$ and $Q2$. These query parts correspond to a node in the query tree constraint. Moreover, we mention what variable is introduced in which part of the textual constraint description. However, we omit the formal definitions of the constraints and present the tool UI constraint visualization instead.

Additionally, we also measured the evaluation time for each presented example 10 times. This evaluation time includes the complete construction of all output bindings with violation information, but does not include the time it takes to transfer this result back to the frontend of the tool. To better represent the measured durations, we categorize the mean execution time in differently colored duration ranges. All 10 measured values are plotted inside a range plot, where the different duration categories are colored accordingly. Two such example plots are shown inside Figure 6.2.

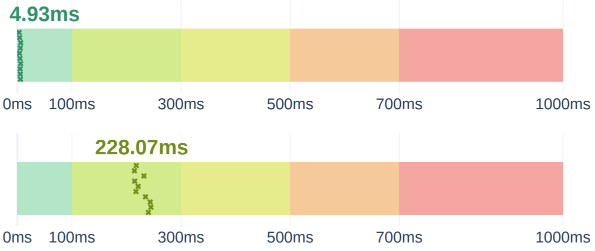


Figure 6.2: Example execution times of different constraints. The execution times on the top are very fast and can be categorized as nearly instant. On the bottom, the execution time is still very fast but takes longer than 0.2 seconds.

The axis covers a range from 0ms to 1000ms (i.e., 1s). All example constraints presented in this section fit well within this range. Of course, it is also possible to model constraints which take longer than 1s to complete on our machine and the considered OCEDs. For most of the constraints or queries we consider, the execution times are located towards the lower end of this scale. This demonstrates the good performance of our approach, on different OCEDs, including one large real-life dataset (i.e., the BPI Challenge 2017 OCED). However, we still selected 1s as the maximum range value, as this execution time is still rather fast, especially in comparison to other approaches and related work. Additionally, this range selection is well suited to compare the execution times for different constraints and datasets.

We also include some example constraints that were automatically discovered based on the OCED as input data. They demonstrate that the automatic discovery approach we presented in Section 4.5 yields useful and understandable constraints, even for real-life datasets.

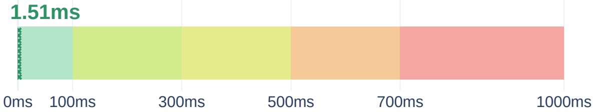
6.2.1 Order Management

Order Paid Exactly Once (Automatically Discovered)

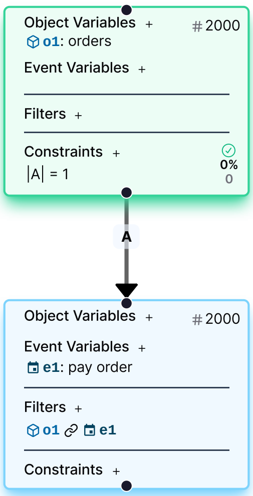
$$\text{“}\underbrace{\text{Every }\overbrace{\text{order}}^{\texttt{o1}}}_{Q1}\ \textit{should}\ \underbrace{\overbrace{\text{be paid}}^{\texttt{e1}}}_{Q2}\ \text{exactly once.”}$$

The representation of this constraint in our tool UI is shown on the right. It consists of two binding boxes: The top one corresponds to $Q1$, while the bottom one corresponds to $Q2$. In essence, this constraint restricts the number of associated events with a specified activity (i.e., `pay order`) for object instances of a given object type (i.e., `orders`).

Evaluation Result In the input OCED, this constraint is never violated. 2000 bindings of order objects are considered. Evaluating this constraint on the OCED only takes around 1.5ms.



Related Work This basic type of constraint, i.e., specifying the allowed number of events of one type related to object instances of another type, is expressible by most constraint approaches. In particular, such constraints are implementable in [6, 31, 34] and even in [10, 20] if the OCED is considered flattened on the `orders` object type.

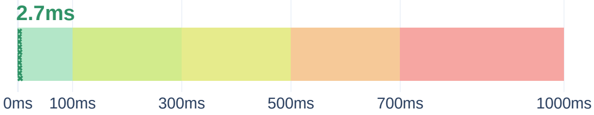

Object Variables + #2000
o1: orders
Event Variables +
Filters +
Constraints +
|A| = 1
0%
0
A
Object Variables + #2000
Event Variables +
e1: pay order
Filters +
o1 🔗 e1
Constraints +

At Most 5 Items Per Order (Automatically Discovered)

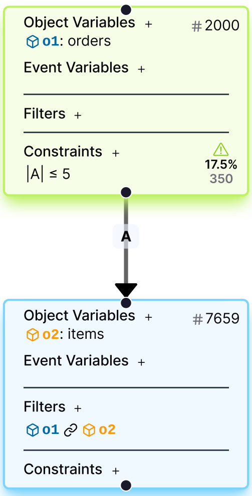
$$\text{“}\underbrace{\text{For every }\overbrace{\text{order}}^{\texttt{o1}}}_{Q1}\ \text{the number of}\ \underbrace{\overbrace{\text{contained items}}^{\texttt{o2}}}_{Q2}\ \textit{should}\ \text{be at most 5.”}$$

The constraint structure is very similar to the previous constraint, now querying associated objects instead of events.

Evaluation Results In the input OCED, this constraint is violated for 350 of the 2000 considered order object bindings, accounting for 17.5%. It can be evaluated in around 2.7ms.



Related Work This basic object count constraint is expressible only in some object-centric constraint approaches (i.e., [31]) but not in [6, 34], as these approach does not consider object-to-object relations. Similarly, it also cannot be expressed in traditional approaches, as there is no clear concept of relations between objects or cases for flat event logs.

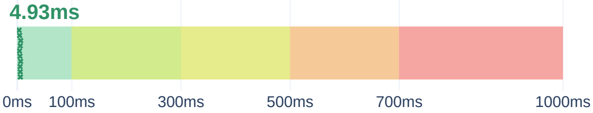

Object Variables + #2000
o1: orders
Event Variables +
Filters +
Constraints +
|A| ≤ 5
⚠
17.5%
350
A
Object Variables + #7659
o2: items
Event Variables +
Filters +
o1 🔗 o2
Constraints +

Items in Order also Associated with Order Placement

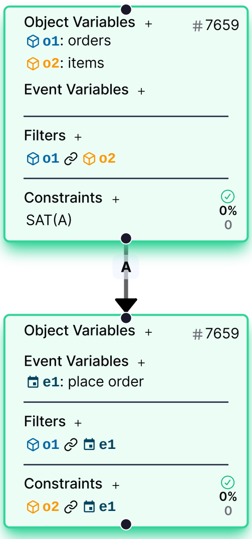
“$\underbrace{\text{For every } \overbrace{\text{order and contained}}^{\texttt{o1}}\ \overbrace{\text{item}}^{\texttt{o2}}}_{Q1}$ the item should also be associated with the $\underbrace{\overbrace{\text{place order}}^{\texttt{e1}}\text{ event of the order}}_{Q2}$.”

First, $Q1$ queries combinations of related order (`o1`) and item (`o2`) objects. Next, for all `place order` events `e1` related to the order `o1`, $Q2$ checks if `e1` is also related to the item `o2`. The constraint of $Q1$ is satisfied for a binding of `o1` and `o2`, if this condition holds for all corresponding `e1` events.

Evaluation Results In the input OCED, this constraint is never violated. 7659 bindings are constructed for $Q1$, as per average there are around 3.82 items per each of the 2000 total orders. This constraint can be evaluated in around 5ms.



Related Work This constraint links object-to-object relationships with “transitive” event-to-object relationships. Such types of constraints are only also expressible in [31]. In [6, 34], connections between objects are not considered, and even if they were, these approaches also do not include concepts allowing explicitly linking multiple relations together.

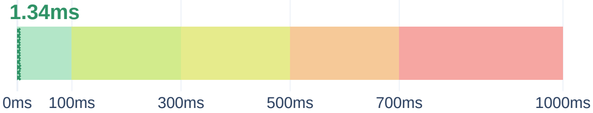


Customer Orders Should Be On Average ≥ 2300€

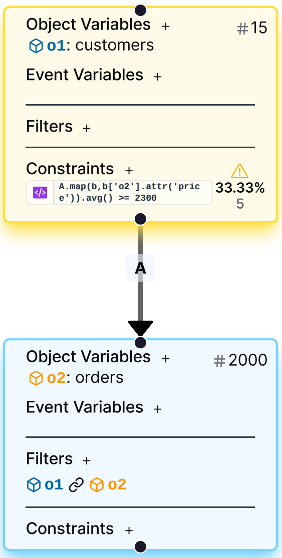
“$\underbrace{\text{For every } \overbrace{\text{customer}}^{\texttt{o1}}}_{Q1}$ the average total price of all $\underbrace{\text{their } \overbrace{\text{orders}}^{\texttt{o2}}}_{Q2}$ *should* be at least 2300€.”

This constraint makes use of an advanced CEL predicate, which computes the average price of orders placed by the customer, based on the set of child bindings in $Q2$, which queries the orders of the customer. The CEL script included in the predicate is: `A.map(b,b['o2'].attr('price')).avg() >= 2300`

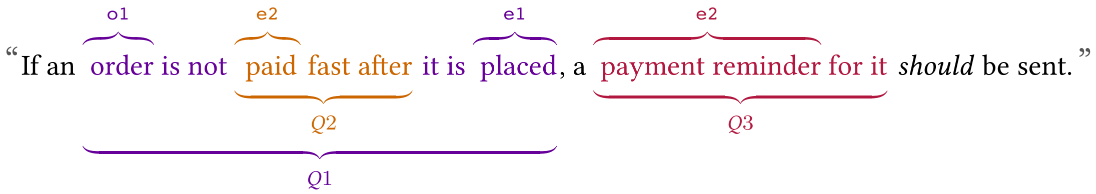
Evaluation Results In the input OCED, this constraint is violated for 5 of the 15 considered customer object bindings, accounting for 33.33%. It can be executed almost instantly, taking only slightly longer than 1ms.

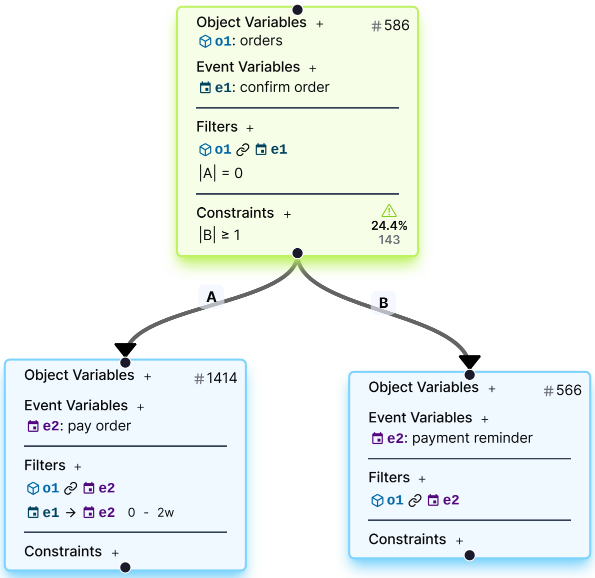


Related Work Such a constraint is not expressible in any considered related work for process constraints. However, SQL-like non-visual querying approaches like [27, 29] might be able to express such aggregation and nested querying.



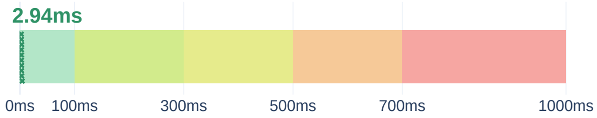
Send Payment Reminder for Orders Not Paid Fast


o1
e2
e1
e2
“If an order is not paid fast after it is placed, a payment reminder for it *should* be sent.”
Q2
Q3
Q1



The top binding box, relating to $Q1$, contains a child-filter predicate based on the number of `pay order` events related to it occurring soon after confirmation (corresponding to $Q2$ on the bottom left). The constraint is then expressed through the child binding set size of the nested querying regarding the number of `payment reminder` events in $Q3$ (shown on the bottom right).

Evaluation Results In the input OCED, this constraint is violated in 143 out of 586 total queried bindings, amounting to around 24.4%. Note that the queried bindings already include the set filter based on `A`, i.e., only bindings with slow payment are included in this number. The number of queried bindings for $Q3$ might seem to suggest that only for 20 of the parent bindings, there is no payment reminder. However, this cannot be observed with confidence, as multiple payment reminders might be sent for a single order. This constraint can be evaluated in around 3ms.

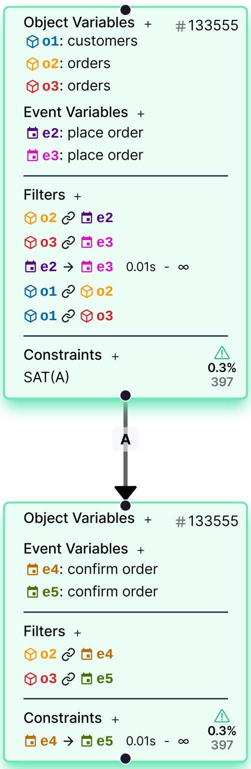


Related Work To the best of our knowledge, there is no graphical approach that enables expressing such constraint types for object-centric logs. However, if the log would be flattened on `orders` as a single case notion, LTL-based constraint approaches like DECLARE [10] with extensions for time durations might be able to express this specific constraint directly in LTL.

Confirm Two Orders by a Customer in the Order They Were Placed

“$\underbrace{\overbrace{\text{Two orders by the same}}^{\texttt{o2, o3}}\ \overbrace{\text{customer}}^{\texttt{o1}}\ \overbrace{\text{placed after each other}}^{\texttt{e2, e3}}}_{Q1}$ *should* also $\underbrace{\text{be}\ \overbrace{\text{confirmed chronologically.}}^{\texttt{e4, e5}}}_{Q2}$”

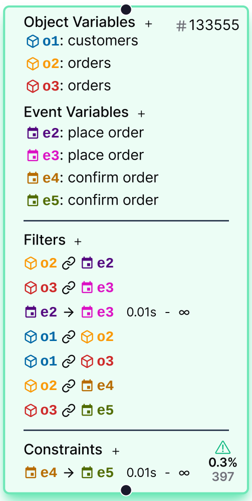
The UI representation of this constraint is shown on the top right, where $Q1$ corresponds to the parent and $Q2$ to the child node. First, $Q1$ queries two `orders` objects associated with the same customer and their corresponding `place order` events. Additionally, a time-between-events predicate filters these events to only those where `e2` occurs at least some time (0.01 seconds) before `e3`. Next, for all such order combination bindings, the `confirm order` events `e4` and `e5`, associated with the orders `o2` and `o3`, respectively, are queried in $Q2$. The constraint regarding the time difference between `e4` and `e5` is implemented in $Q1$ and is then propagated up using the SAT predicate in $Q2$. In particular, this constraint can also be expressed using just a single node (shown on the bottom right), by combining $Q1$ and $Q2$ and simply removing the SAT constraint. However, the constraint formulation using two nodes might be easier to read and understand for unfamiliar users.



Object Variables + #133555
o1: customers
o2: orders
o3: orders
Event Variables +
e2: place order
e3: place order
Filters +
o2 🔗 e2
o3 🔗 e3
e2 → e3 0.01s - ∞
o1 🔗 o2
o1 🔗 o3
Constraints + ⚠ 0.3% 397
SAT(A)

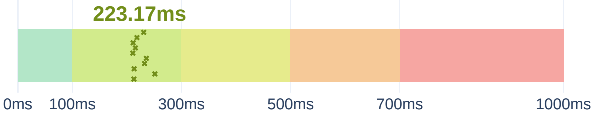
Object Variables + #133555
Event Variables +
e4: confirm order
e5: confirm order
Filters +
o2 🔗 e4
o3 🔗 e5
Constraints + ⚠ 0.3% 397
e4 → e5 0.01s - ∞

or

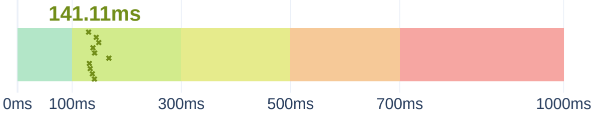


Object Variables + #133555
o1: customers
o2: orders
o3: orders
Event Variables +
e2: place order
e3: place order
e4: confirm order
e5: confirm order
Filters +
o2 🔗 e2
o3 🔗 e3
e2 → e3 0.01s - ∞
o1 🔗 o2
o1 🔗 o3
o2 🔗 e4
o3 🔗 e5
Constraints + ⚠ 0.3% 397
e4 → e5 0.01s - ∞

Evaluation Results Overall, 133555 combinations of placed orders by the same customer are considered, for 0.3% of which the constraint is not satisfied, amounting to 397 violated bindings. The very large number of considered bindings in this complex constraint leads to a higher than previously encountered execution time, with a mean of around 223ms for the formulation using two nodes. For the single node formulation, the execution time is measurably faster with a mean of around 141ms, because of the reduced overhead that occurs in the recursive evaluation, for example when checking if the bindings of the child node are satisfied.



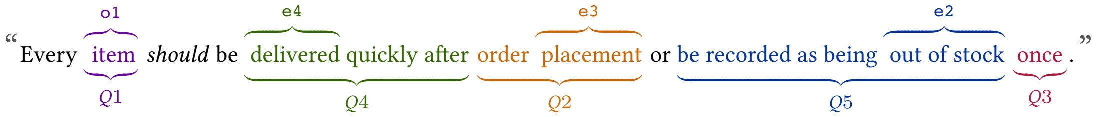
(two nodes)



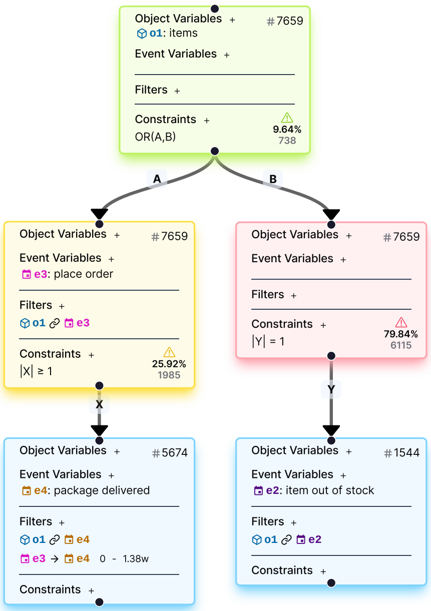
(single node)

Related Work None of the considered related work for visual queries or constraints allow specifying this constraint, as it involves two object instances of the same object type. In particular, the general type of query, in this example involving two orders by the same customer, demonstrates the very high expressiveness of our approach as well as the limitations of previous work.

Items Delivered Quickly After Placement Or Out Of Stock (Automatically Discovered)

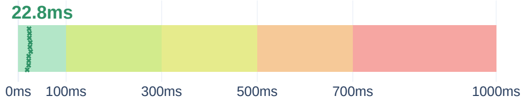


"Every item *should* be delivered quickly after order placement or be recorded as being out of stock once."



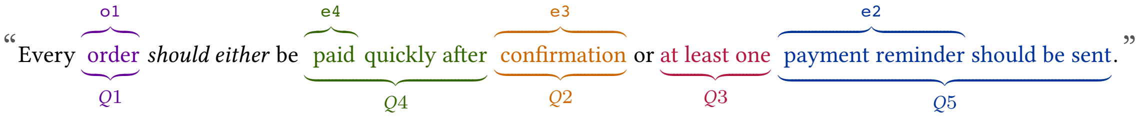
This constraint consists of *Q*1 through *Q*5, numbered from top to bottom, left to right in the query tree constraint. This structure is typical for automatically discovered OR constructs: The common variable (o1) is queried at *Q*1 and the two OR conditions are implemented using two nodes each.

Evaluation Results The root node has 7659 output bindings, of which 738 (9.64%) are violated. Evaluating this constraint takes around 23ms.

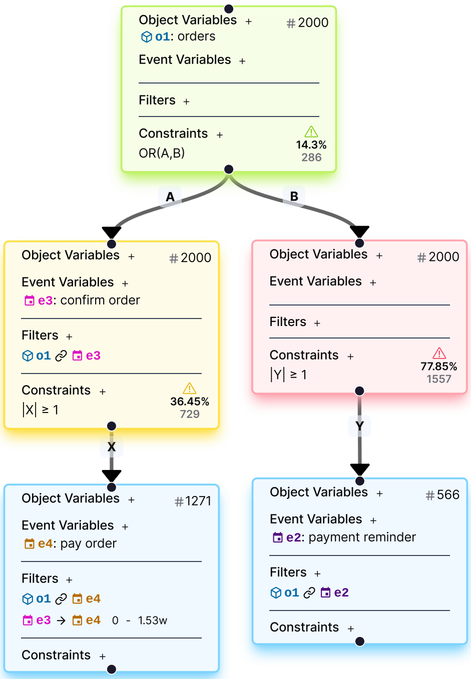


Related Work This constraint cannot be modeled in any of the considered object-centric approaches, as it is an OR construct between a count- and eventually-follows part. Generally, however, flat OR constructs can be implemented in DECLARE [10].

Order Paid Fast After Confirmation Or Payment Reminder (Automatically Discovered)

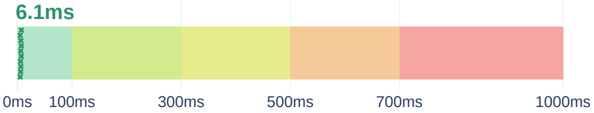


"Every order *should either* be paid quickly after confirmation or at least one payment reminder should be sent."



The constraint structure is very similar to the previously presented constraint. In fact, this automatically discovered constraint is again an OR construct of an eventually-follows and count constraint subconstraint.

Evaluation Results In the input OCED, this root constraint node is violated for 286 of the 2000 considered order object bindings, accounting for 14.3%. It can be evaluated in around 6ms.

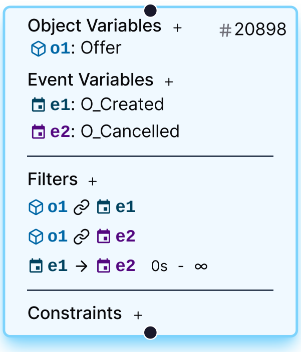


Related Work Like the previous example, this type of constraint is again not expressible in any of the considered object-centric constraint approaches.

6.2.2 BPI Challenge 2017 OCED

Query Offer Creations and Cancellations with Eventually-Follows

$$\text{“}\underbrace{\text{Query all }\overbrace{\text{creation and cancel events}}^{\texttt{e1,e2}}\text{ associated with }\overbrace{\text{same offer}}^{\texttt{o1}}\text{, where the create event occurs first.}}_{Q1}\text{”}$$



Object Variables + #20898
o1: Offer

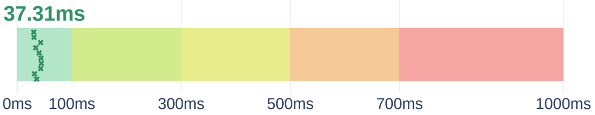
Event Variables +
e1: O_Created
e2: O_Cancelled

Filters +
o1 🔗 e1
o1 🔗 e2
e1 → e2 0s - ∞

Constraints +

This query corresponds to the Cypher query shown in Code 2 in Chapter 2. For simplicity, we assume that no two events occur at the exactly same time, and thus this query effectively filters only eventually-follows relations between `e1` and `e2`.

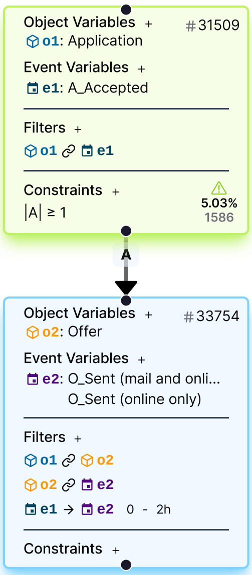
Evaluation Results This query constructs 20898 output bindings in total, just like the Cypher query from Code 2. In our implementation, evaluating this query takes on average around 37ms, which is significantly less than the execution time of the Cypher query, reported by the authors of [29] as 140ms.



Related Work Such queries can be expressed in most existing process instance querying approach we covered, in particular in [27, 29]. However, none of these approaches allow for visual construction of this query and [27] also does not focus on object-centricity, which might limit similar queries which additionally involve another object type.

At Least One Offer Sent Quickly After Application Accepted

$$\text{“For every }\underbrace{\overbrace{\text{accepted}}^{\texttt{e1}}\ \overbrace{\text{application}}^{\texttt{o1}}}_{Q1}\text{ there \textit{should} be at least one }\underbrace{\overbrace{\text{offer}}^{\texttt{o2}}\ \overbrace{\text{sent out}}^{\texttt{e2}}\text{ within 2 hours after acceptance.}}_{Q2}\text{”}$$



Object Variables + #31509
o1: Application

Event Variables +
e1: A_Accepted

Filters +
o1 🔗 e1

Constraints +
|A| ≥ 1
⚠ 5.03%
1586

A

Object Variables + #33754
o2: Offer

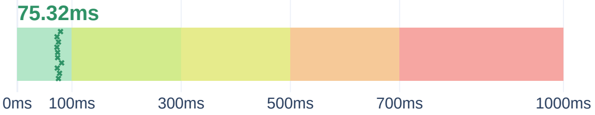
Event Variables +
e2: O_Sent (mail and onli...
O_Sent (online only)

Filters +
o1 🔗 o2
o2 🔗 e2
e1 → e2 0 - 2h

Constraints +

$Q1$ first queries all accepted applications and then constraints that there should be at least one child binding in $Q2$. $Q2$ queries all offers related to this application and offer sent events that occur within 2 hours of the application acceptance event.

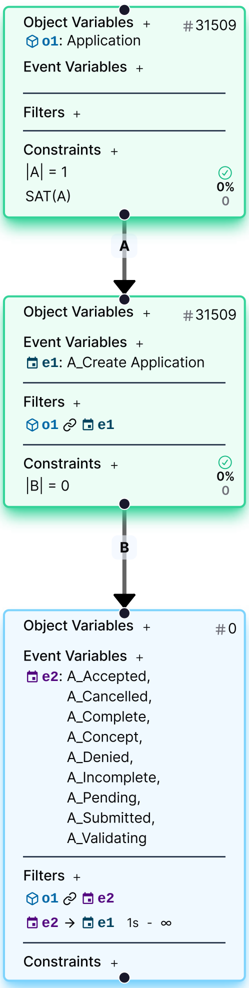
Evaluation Results This constraint constructs 31509 output bindings in total, of which 1586 are violated, corresponding to 5.03%. Evaluating this constraint takes around 75ms.



Related Work This constraint cannot be modeled in any of the considered approaches, as it involves an eventually-follows constraint limiting the maximum duration between events related to different objects of different object types, which are linked through object-to-object relationships.

Applications are First Created Exactly Once

$$\text{“}\underbrace{\text{Every }\overbrace{\text{Application}}^{\texttt{o1}}}_{Q1}\ \textit{should}\ \underbrace{\text{be }\overbrace{\text{created}}^{\texttt{e1}}}_{Q2}\ \text{once}\ \underbrace{\text{initially}}_{Q3}.\text{”}$$



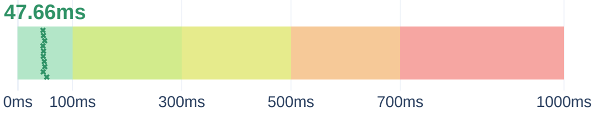
Object Variables + #31509
o1: Application
Event Variables +
Filters +
Constraints +
|A| = 1
SAT(A)
0%
0

Object Variables + #31509
Event Variables +
e1: A_Create Application
Filters +
o1 🔗 e1
Constraints +
|B| = 0
0%
0

Object Variables + #0
Event Variables +
e2: A_Accepted,
A_Cancelled,
A_Complete,
A_Concept,
A_Denied,
A_Incomplete,
A_Pending,
A_Submitted,
A_Validating
Filters +
o1 🔗 e2
e2 → e1 1s - ∞
Constraints +

The UI representation of this constraint is again shown on the right, where $Q1$, $Q2$, and $Q3$ are shown from the top to the bottom. First, $Q1$ queries all `Application` objects. Next, all `A_Create Application` events `e1` related to the application `o1` are queried in $Q2$. $Q3$ queries all events of other event types which occur *before* `e1`. Regarding the constraints, $Q2$ is satisfied if there is no such event (i.e., `e1` is the first event involving `o1`). $Q1$ is then satisfied if there is exactly one child binding in $Q2$ and this child binding is satisfied in $Q2$.

Evaluation Results In the input OCED, this constraint is never violated. 31509 bindings are constructed for $Q1$ and $Q2$ each. For $Q3$ no binding is constructed, which is also clear as otherwise the constraint would be violated for at least one binding. Evaluating this constraint takes around 47.6ms.



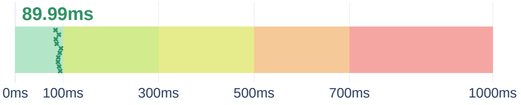
Related Work This constraint corresponds to a traditional DECLARE construct, specifying an initial activity and that this activity should be executed exactly once [10]. However, DECLARE, of course, considers only traditional event data and cannot represent this constraint if not flattened on the `Application` object type.

Offers Are Either Cancelled or Returned (Automatically Discovered)

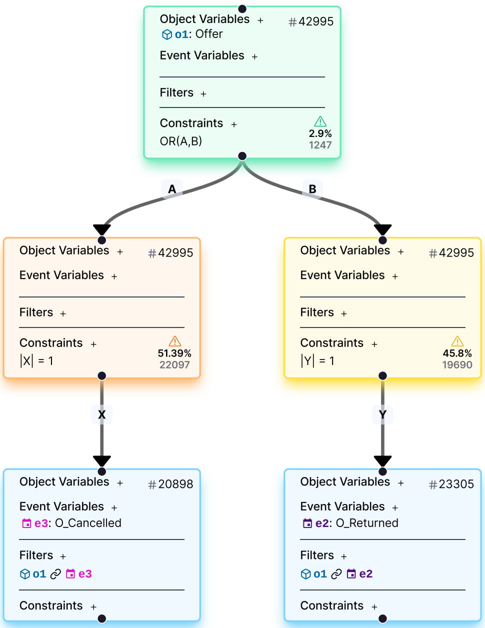
$$\text{“Every } \underbrace{\overbrace{\text{offer}}^{\texttt{o1}}}_{Q1} \textit{ should} \text{ either be } \underbrace{\overbrace{\text{returned}}^{\texttt{e2}}}_{Q4} \underbrace{\text{exactly once}}_{Q2} \text{ or } \underbrace{\overbrace{\text{cancelled}}^{\texttt{e3}}}_{Q5} \underbrace{\text{exactly once}}_{Q3}\text{.”}$$

The root node, corresponding to $Q1$, queries all offer objects and implements an OR constraint of the two children $Q2$ and $Q3$. These children are satisfied if there is exactly one child binding in $Q4$ and $Q5$, respectively.

Evaluation Results The root node constructs 42995 output bindings in total, of which 1247 (2.9%) are violated. Evaluating this constraint takes around 90ms.



Related Work This constraint cannot be modeled in any of the considered approaches, as it contains an OR construct between two parts, each specifying that there should be exactly one event of a specific type per order.

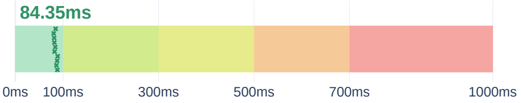


Applications Are Accepted Quickly Or Submitted Online (Automatically Discovered)

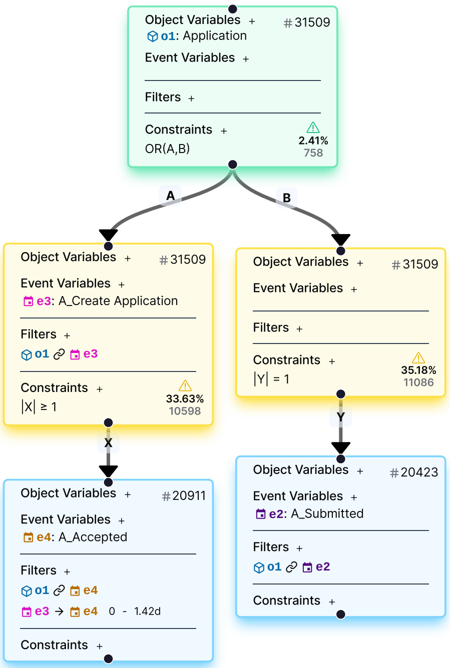
$$\text{“Every } \underbrace{\overbrace{\text{application}}^{\texttt{o1}}}_{Q1} \textit{ should} \text{ either be } \underbrace{\overbrace{\text{accepted}}^{\texttt{e4}}}_{Q4} \underbrace{\text{quickly after being } \overbrace{\text{created}}^{\texttt{e3}}}_{Q2} \text{ or } \underbrace{\overbrace{\text{submitted online}}^{\texttt{e2}}}_{Q5} \underbrace{\text{exactly once}}_{Q3}\text{.”}$$

The A_Submitted activity corresponds to an online submission. Thus, this constraint implies that applications not submitted online are accepted quickly. The five query tree nodes correspond to $Q1$ through $Q5$, ordered from the top to bottom and left to right.

Evaluation Results The root node has 31509 output bindings, of which 758 (2.41%) are violated. Evaluation takes around 84ms.



Related Work This constraint cannot be modeled in any of the considered approaches, as it is an OR construct between a count- and eventually-follows part.



6.3 Performance Analysis

In the previous section, we already presented execution times for the considered examples. In this section, we now focus on evaluating the runtime performance and scalability of our implementation in more detail. A challenging aspect of the high expressiveness of our query tree constraint is that the evaluation time heavily depends on the type of modeled constraint or query. As such, we cannot give a single expected runtime for evaluating arbitrary queries. Instead, we present a few simple query scenarios and argue about the expected runtime trend across different OCEDs, encompassed by a *scaling factor*, afterwards comparing these estimations with measurements. Additionally, we explore the scalability of a single complex constraint in more detail, creating artificial OCEDs of different sizes and measuring the resulting execution time.

6.3.1 Query & Constraint Scenarios

We explore different query or constraint scenarios, which can be formulated in multiple different OCEDs. The following scenarios are considered:

S1 Bind one object of a specified type.
S2 Bind an object and a related event or object.
S3 Bind an object and do a nested query for a set of related objects or events, evaluate a constraint regarding the number of elements in this set.
S4 Bind two objects without specifying any predicates linking them.

To quantify the expected scalability behavior for each considered OCED and scenario, we determine a *Scaling Factor* for each configuration, where a factor of 1 corresponds to 1000 instances or bindings. This factor corresponds to the maximal number of output bindings considered for each query tree node. For example, for **S1**, the expected scaling factor is the number of instances of the specified object type. Below, we describe how we calculated the scaling factors:

Procure-to-Pay

S1 Object Type: `purchase_requisition`. There are 927 such objects, resulting in a scaling factor of 0.927.
S2 Object Type: `purchase_requisition`, related object type: `quotation`. There are 927 purchase requisition objects, each with exactly 1 related `quotation` object, resulting in a scaling factor of 0.927.
S3 See S1/S2, as the maximum number of considered bindings stays the same. Scaling factor: 0.927
S4 927 objects of type `purchase_requisition`, 1598 objects of type `purchase_order`, yielding a scaling factor of 1481.346

Order Management

S1 Object Type: `orders`. There are 2000 order objects, resulting in a scaling factor of 2.
S2 Object Type: `orders`, related event type: `pay order`. There are 2000 order objects each with 1 `pay order` event, resulting in a scaling factor of 2.
S3 See S1/S2, as the maximum number of considered bindings stays the same. Scaling factor: 2
S4 2000 objects of type `orders`, 15 objects of type `customers`, yielding a scaling factor of 30

Container Logistics

S1 Object Type: `Handling Unit`. There are 10579 handling unit objects in the log, resulting in a scaling factor of 10.579.
S2 Object Type: `Handling Unit`, related event type: `Load Truck`. There are 10553 instances of such related object and events, resulting in a scaling factor of 10.553
S3 See S1, as first only the handling units are quried. Scaling factor: 10.579.
S4 10579 objects of type `Handling Unit`, 6 objects of type `Truck`, yielding a scaling factor of 63.474.

BPI Challenge 2017

S1 Object Type: `Application`. There are 31509 such objects in the log, resulting in a scaling factor of 31.509.
S2 Object Type: `Application`, related event type: `A_Complete`. There are 31362 instances of such related object and events, resulting in a scaling factor of 31.362
S3 See S1, as first only an application is queried. Scaling factor: 31.509.
S4 31509 objects of type `Application`, 149 objects of type `Case_R`, yielding a scaling factor of 4694.841.

Figure 6.3 shows the measured execution time for each scenario, indicating a trendline of how the execution time scales together with the OCED scaling factor. In all considered scenarios, the execution time seems to scale linearly with the scaling factor. This supports our assumption that for the most common constraint and query situations, the execution time of evaluating them scales linearly with the number of output bindings. The number of output bindings themselves scale linearly with the relevant aspects of the input OCED (e.g., number of objects of a specified type, or average number of related objects), of course depending on the considered query tree.

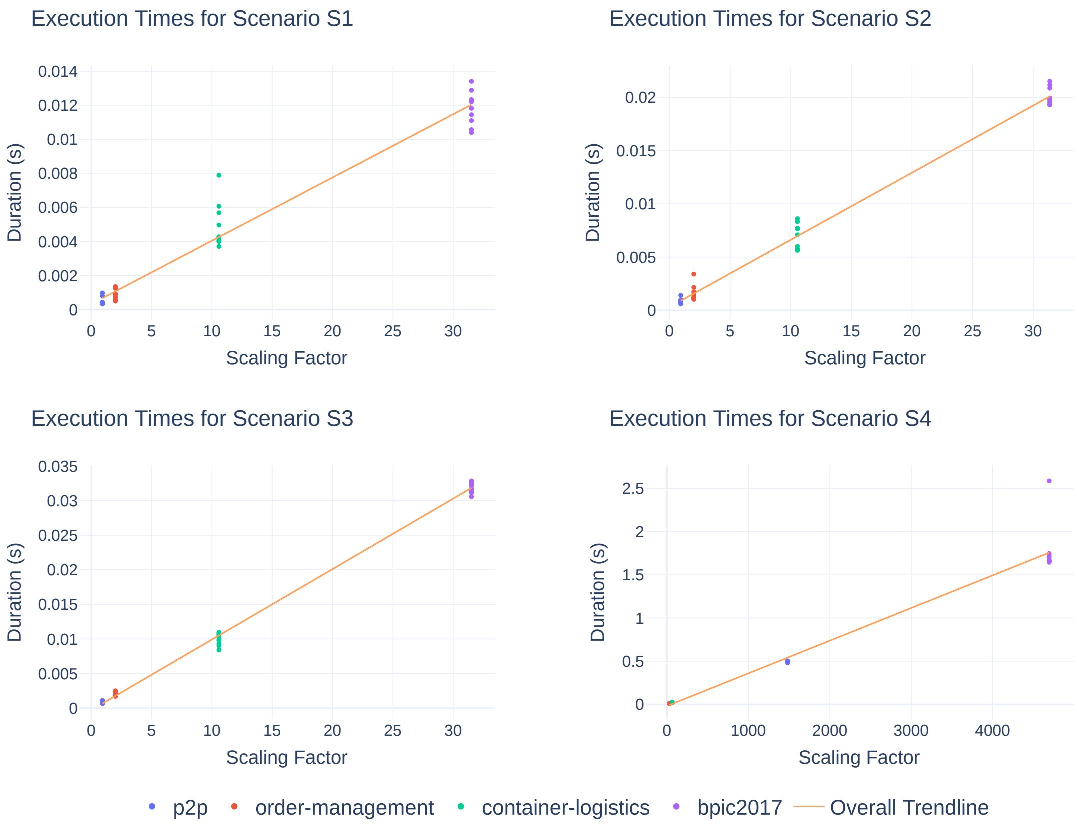


Figure 6.3: Scatter plots showing the execution times for evaluating the scenarios across different OCEDs with scaling factors derived from the OCED based on the scenario. Each scenario was evaluated 10 times for each OCED and a trendline for each scenario indicates the scalability trend.

6.3.2 Scalability for Complex Constraints

To evaluate how the execution time for evaluating complex constraints grows when considering larger event data, we modified the simulation model which was used to create the OCEL 2.0 order management logs from [36]. In particular, we scaled the number of generated objects of type `orders` with a factor $k \in \{1, 2, 3, 4\}$ and additionally either:

a) Also scaled the number of `customer` objects by the same factor (starting with 15 at $k = 1$)

b) Kept the number of `customers` constant at 15 (as in the original log)

The underlying simulation model connects orders to customers randomly. In particular, it can thus reasonably be assumed, that the number of orders per customer is at least roughly similar and there are no underlying patterns between specific customers and the number of associated orders. The number of other object and event instances in the scaled OCED was not manually influenced and thus follows the simulation mode based on the scaled number of orders and customers. For instance, there is still exactly one event of type `place order` and `confirm order` for each order object.

For each factor $k \in \{1, 2, 3, 4\}$, we created two OCEDs, one with constant customers and one with scaled number of customers. We then evaluated the constraint for two orders placed by the same customer we presented in Section 6.2.1 for each constructed OCED and recorded the execution time of evaluating the constraint. We repeated each measurement 10 times.

To derive the expected scalability trends, we estimate the expected number of output bindings in each setting under some simplifications. In the constraint, two orders by the same customers are considered. First, we only consider the related object and event filter predicates and ignore the time-between-events filter predicate, which only reduces the result by a constant factor. Additionally, in the order management OCED, there is exactly one `place order` and `confirm order` per order object, so only considering the bindings of the customer and orders is sufficient to estimate the output binding size. For simplicity, we only consider the average number of orders per customer N_O and the number of customers N_C and the corresponding scaling factor $k \in \{1, 2, 3, 4\}$, we can estimate the number of considered bindings (i.e., combinations of two orders by the same customer) in the different settings as:

a) $k \cdot N_C \cdot (N_O \cdot N_O) = k \cdot N_C \cdot N_O^2$

b) $N_C \cdot (k \cdot N_O \cdot k \cdot N_O) = k^2 \cdot N_C \cdot N_O^2$

In particular, the scaling factor affects the number of bindings linearly in setting a), and quadratically in setting b). This might seem counterintuitive, as the overall number of objects in the OCED in setting b) is smaller than in setting a). However, for this particular constraint, the number of orders per customer is what translates to a quadratic factor for b). With a constant number of customers, the number of orders per customer increases by the factor k (and thus k^2 for two orders by the same customer), while for a linearly scaled number of customers, the number of orders per customer stays the same.

Of course, there are other factors to also consider, like additional specifics of the considered constraint or data, and the parallelization of the execution engine. However, the general trend is, in fact, confirmed by the measured execution times results, which we plotted in Figure 6.4. In particular, for setting a) when linear scaling of both orders and customers, the execution time also grows linearly, while in setting b) when keeping the number of customers constant and only increasing the number of orders, a quadratic trend of the execution times could be observed.

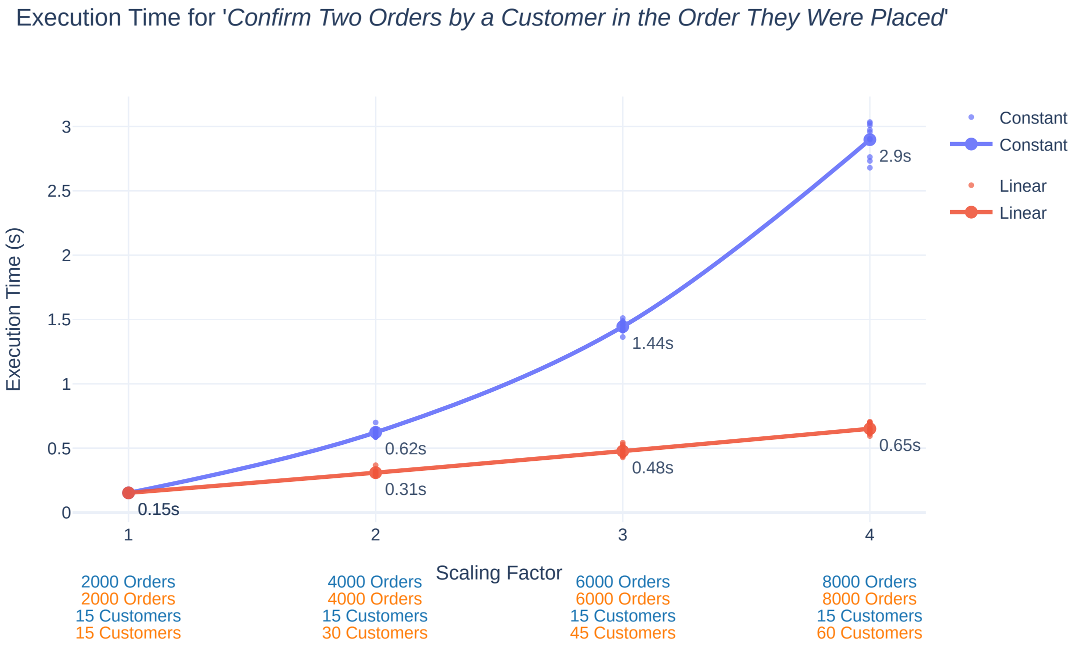


Figure 6.4: Execution times of a constraint involving two orders by the same customer. Constant scaling refers to only increasing the number of orders by a factor and keeping the number of customers the same, while linear refers to also scaling customers with the same factor. We repeated each measurement 10 times and plotted the resulting values as a scatter plot. Additionally, the means are shown as larger points and are connected with a spline curve, indicating the scalability trend. As expected, for the linear scaling, the execution times follow a linear trend. For a constant number of customers, the execution times increase at a non-constant rate and can be well described by a quadratic function, which also corresponds to the expected outcome. In particular, this quadratic trend can be explained by the fact that for a constant number of customers, the average number of orders per customer scales linearly. Thus, when considering two orders by the same customer, two linear factors are multiplied, resulting in quadratic scaling.

6.4 Threats to Validity

Throughout this chapter, we evaluated the constraint and query approach presented in this thesis. In the qualitative evaluation, we presented some example constraints for different OCED together with their execution time. The examples demonstrate the features and expressiveness of our approach, ranging from simple to more complex constraints and queries. Of course, this selection only scratches the surface of the types of constraints that are implementable.

Additionally, most of the OCEDs considered in this chapter were rather small and based on simulated data, as there is a shortage of other publicly available larger OCED based on real-life processes. As a notable exception, we also considered the BPI Challenge 2017 OCED, which is a larger real-life dataset.

While we included some example constraints that were automatically discovered, we did not evaluate the automatic discovery presented in Section 4.5 in more detail, both in terms of quality and execution performance.

Furthermore, we did not study the expressiveness of our approach compared to related work systematically, but only by example constraints. Such a systematic evaluation would be interesting but as many approaches from related work focus on specific different aspects, for example on aggregated queries or constraints rooted in performance measures, it would also be difficult to conduct meaningfully.

In the quantitative evaluation, the second part of this chapter, we focused purely on the evaluation time and its scalability. In the scenario analysis, we only considered rather basic types of queries and constraints, for which we demonstrated that the runtime indeed scales linearly with the expected scalability factor, which mostly corresponds to the number of considered bindings. However, we did not cover more complex queries in these scenarios. Later on, in the scalability experiments for a more complex constraint, only one example constraint was considered and evaluated on different OCEDs, where the number of instances of certain object types was varied. It would be interesting to also investigate the scalability of other constraints on this OCED.

Chapter 7

Discussion

In this chapter, we discuss design choices made throughout this thesis. In particular, we describe limitations regarding the scalability of the approach, especially considering the tradeoff between allowing users to design complex constraints and preventing explosions in the number of output bindings. Additionally, we address and justify our implementation approach consisting of a custom execution engine against other established querying languages, like SQL.

7.1 Scalability Limitations

Throughout our approach, we focused on allowing very high expressiveness for queries and constraints. In particular, binding boxes can bind an arbitrary number of object and event variables and also contain arbitrary filter predicates. However, with this great power also comes great responsibility: It is not difficult to construct a binding box with a gigantic output binding set, even for smaller OCEDs. Simply consider binding many event and object variables to values of all event and object types and adding no filter predicates. Even if we consider a very small OCED with only 10 objects, a binding box with 10 object variables assigned to any type would still yield an output binding set of size 10^{10} (10 billion) and would most likely not be able to fit into system memory on most machines. However, this limitation is mostly theoretical. After all, such an example binding box would have no realistic use case. In particular, while the number of output bindings *can explode in size*, it is the responsibility of the user to design useful constraints that can efficiently be evaluated. One good heuristic to fulfill this responsibility in practice is making sure that all introduced object and event variables are connected at each step, through appropriate event-to-object and object-to-object relationship filter predicates.

It is also important to note that these scalability limitations are inherent to the very general types of constraints expressible in our approach. For instance, *general* constraints regarding two orders placed by the same customer can only be evaluated for concrete instantiations (i.e., bindings) of the orders and customer. Of course, for certain specific constraints there might be some optimizations possible, like the ones we explore in Section 4.4 and also implemented in our tool.

One of the design decisions we made with our approach was not restricting query tree constraints or binding boxes to only allow constructs corresponding to the previously mentioned heuristic (e.g., requiring that all introduced variables are somehow connected through predicates). We believe that allowing the creation of unusual constraints with disconnected parts is the better choice, as it increases the set of implementable constraints and also keeps the approach formalization simpler. Furthermore, there are some less strict mitigation strategies to prevent users from accidentally designing problematic constraints, for instance, by showing a warning for such unconnected variables in the tool UI. Moreover, in future development of the OCPQ tool, also adding safeguards to make the execution engine more robust is an important aspect. For instance, this could be achieved by limiting the number of constructed output bindings to prevent overload caused by very large queries. Another approach could be to implement lazy evaluation with streaming support for the resulting output bindings, which could allow constructing even larger output binding sets and would also allow stopping evaluation early if enough interesting results are received.

7.2 Why not SQL?

In this thesis, we proposed a custom implementation of an execution engine for constructing the queried bindings and checking the included constraints. As the OCEL 2.0 standard for object-centric event logs also includes SQLite as an exchange format [2], it begs the question whether implementing a simple translation between our approach and SQL queries might be a better choice than implementing an own execution engine. In the following, we will shortly discuss why we opted for a dedicated implementation, motivated by both flexibility and performance.

First, we need to mention some advantages that an implementation rooted in SQLite would have. SQL is well-supported across different architectures and has a rich ecosystem around it. Secondly, SQL is very well optimized, generally speaking. At the same time, SQL has several different use cases and is not specialized for the specific types of queries at hand. Moreover, while SQL supports many features and is quite flexible in general, translating our proposed very expressive and extensible predicate statements to SQL might not even be possible. There, the flexibility of a full general programming language triumphs over the large number of statements and queries supported in SQL.

Consider the query tree shown in Figure 7.1 as well as the SQL query from Code 6. Both implement the same querying of two orders, `o1` and `o2`, with corresponding place order events `e1` and `e2`, both from the same customer `o0`, such that `e1` happens before `e2`. In the SQL, the binding predicates from Figure 7.1 are translated to `INNER JOIN` constructs and `WHERE` clauses. Executing the SQL query shown in Code 6 takes around 19000ms. Executing a similar query in our tool takes only roughly 200ms. This implies that, at least for evaluating this example, the SQLite query is around 100 times slower than our tool.

There might be more efficient ways to structure OCEL 2.0 database tables and execute queries in SQL, specifically focused on our use case. However, optimization of the query plan and using heuristics to optimize execution time is generally considered a strong suit of SQL. The SQLite file that was used for this example evaluation had existing indexes for all interesting JOIN-operations. The query execution plan for the example query, which details how the query is handled, also indicates that query optimization does indeed happen (e.g., initially the second event is queried, then later an index is used to find the corresponding order object in `E202`). However, our implementation is specifically designed for such types of queries and also builds abstractions and structures to efficiently construct output binding sets. Additionally, as the construction of one element of the output binding sets does not influence other elements, it is heavily parallelized in our implementation. Also, more complex predicates that are not easily representable in SQL can be added to our approach, as bindings are checked for predicates using a full programming language. For instance, the CEL extension described in Subsection 4.6.1 and the corresponding binding predicates introduced there could not be easily implemented in a SQL-based approach.

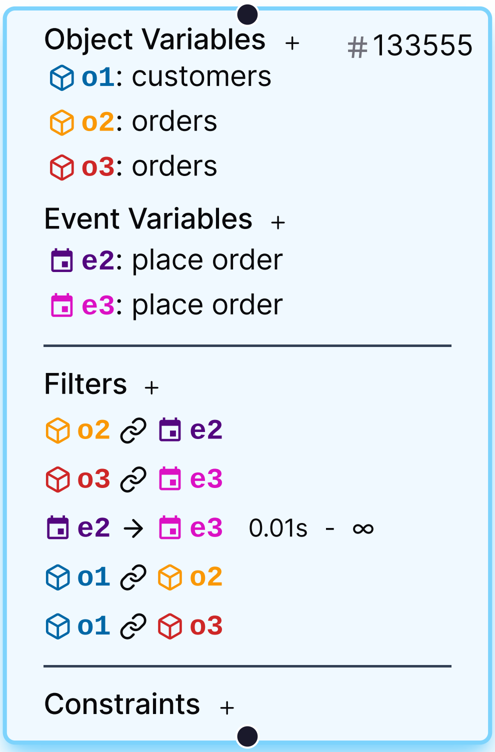


Figure 7.1: An equivalent query to the SQL.

```sql
    SELECT
    A0.ocel_id AS o0,
    A1.ocel_id AS o1,
    A2.ocel_id AS o2,
    E1.ocel_id AS e1,
    E2.ocel_id AS e2
  FROM
    object AS A0,
    object AS A1,
    object AS A2,
    event AS E1,
    event AS E2
    INNER JOIN object_object AS O2O ON O2O.ocel_source_id = A0.ocel_id
    AND O2O.ocel_target_id = A1.ocel_id
    INNER JOIN object_object AS O2O2 ON O2O2.ocel_source_id = A0.ocel_id
    AND O2O2.ocel_target_id = A2.ocel_id
    INNER JOIN event_object AS E2O ON E2O.ocel_event_id = E1.ocel_id
    AND E2O.ocel_object_id = A1.ocel_id
    INNER JOIN event_object AS E2O2 ON E2O2.ocel_event_id = E2.ocel_id
    AND E2O2.ocel_object_id = A2.ocel_id
    INNER JOIN event_PlaceOrder AS E_PO ON E_PO.ocel_id = e1
    INNER JOIN event_PlaceOrder AS E_PO2 ON E_PO2.ocel_id = e2
    AND E_PO.ocel_time < E_PO2.ocel_time
  WHERE
    A0.ocel_type = 'customers'
    AND A1.ocel_type = 'orders'
    AND A2.ocel_type = 'orders'
    AND E1.ocel_type = 'place order'
    AND E2.ocel_type = 'place order'
    AND o1 != o2
    AND E_PO.ocel_time < E_PO2.ocel_time
```

Code 6: SQL query statement for selecting two orders of the same customer, analogous to the query shown in Figure 7.1 in our approach.

```
QUERY PLAN
|--SEARCH E2 USING INDEX type (ocel_type=?)
|--SEARCH E_PO2 USING INDEX idx_event_PlaceOrder_ocel_id (ocel_id=?)
|--SEARCH E2O2 USING COVERING INDEX idx_event_object_composite_pk (ocel_event_id=?)
|--SEARCH A2 USING INDEX ID (ocel_id=?)
|--SEARCH E1 USING INDEX type (ocel_type=?)
|--SEARCH E_PO USING INDEX idx_event_PlaceOrder_ocel_id (ocel_id=?)
|--SEARCH E2O USING COVERING INDEX idx_event_object_composite_pk (ocel_event_id=?)
|--BLOOM FILTER ON A1 (ocel_id=?)
|--SEARCH A1 USING INDEX ID (ocel_id=?)
|--SEARCH O2O2 USING INDEX idx_object_object_ocel_target_id (ocel_target_id=?)
|--BLOOM FILTER ON A0 (ocel_id=?)
|--SEARCH A0 USING INDEX ID (ocel_id=?)
`--SEARCH O2O USING COVERING INDEX o2o (ocel_source_id=? AND ocel_target_id=?)
```

Code 7: The SQL query plan for the SQL statement from Code 6.

Chapter 8

Conclusion

In this thesis, we presented an object-centric query and constraint approach, which allows nested querying of *bindings* (i.e., combinations of process instances – events and objects). Constraints can be added to label the output bindings of a query as either *allowed* or *forbidden*. Our approach is based on the concepts of the aforementioned *variable bindings*, which encompass combinations of named process instances as well as *binding predicates*, which induce a set of variables for which they are satisfied. Through its flexibility, the presented query and constraint approach is very expressive and allows querying situations and formulating constraints that are not possible to express in prior work. We first introduced the proposed declarative query and constraint approach formally. Next, we described how the presented declarative concepts can be mapped to an algorithm that efficiently computes the required outputs of queries and constraints. We highlighted how naive approaches to this problem run into scalability and performance issues already for relatively small data. As a solution, we present more sophisticated algorithmic approaches, consisting of a recursive algorithm for evaluating nested queries and constraints, as well as a binding expansion ordering strategy, which enables efficient construction of query output bindings. These general algorithmic procedures were also implemented in the software tool *OCPQ* we developed, consisting of a performance-centric execution engine and an interactive editor frontend. The frontend constraint editor also allows inexperienced users to design, evaluate, and analyze the results of simple to complex queries and constraints. Additionally, we showcased how different types of constraints can automatically be discovered based on an input OCED. These types of constraints, count constraints and eventually-follows constraints, can rather easily be mined based on input data and are simultaneously very relevant for real-life processes. Moreover, we demonstrated how also more complex types of constraints can be discovered, by presenting a way to combine specific constraint parts into an OR-construct. In our evaluation, we wanted to show two main important aspects of our approach: 1) that it is very expressive and can formulate interesting, simple or complex queries and constraints conveniently, that were otherwise not possible to model in existing related work and 2) that even though we allow for such a high expressiveness, evaluating queries and constraints is still very fast and can also be applied on large, real-life datasets. To address 1) we presented more than ten different constraints and queries, demonstrating the high expressiveness and convenient representation of constraints. Additionally, to also address 2) partly, we recorded the time it takes to evaluate all of these examples and plotted the results. All the measured execution times were way below 1s, even for larger real-life OCED containing more than a million events, indicating very good runtime performance. Moreover, we also specifically analyzed the scalability of our implementation by creating artificial OCED datasets, manipulating the number of objects of certain object types. Through this analysis, we confirmed that for the most common query and constraint constructs, the runtime scales linearly with the size of the input OCED. Finally, we also discussed the design choices we made throughout this thesis.

Future Work The proposed query and constraint approach is inherently extendable by expanding the collections of binding predicates introduced in this thesis. For example, interesting predicate types not considered in this thesis are predicates based on directly-follows relationships between event variables. In particular, in the context of OCED, directly-follow predicates should also consider an additional object variable, corresponding to the object instance for which the two events should directly follow each other. There are heaps of other predicate types to consider. For instance, ranging from performance-metric inspired predicates, like a maximum waiting or processing time of events, to convenient shorthands for predicates already expressible by more complex constructs in the current approach, like specifying that an event should be the first event associated with an object.

Apart from extensions of the possible predicates, also more fundamental additions are intriguing to consider. For example, expanding the concept of variable bindings to also allow other types of variables or values, ranging from attribute values, like numbers, to sets of queried child bindings. As a first step, the extension idea based on arbitrary annotation values of output bindings should be formalized and implemented in *OCPQ*.

We also shortly covered related work on event log filtering approaches, which allow specifying filters as well as analyzing and exporting the resulting filtered view on the data. The querying approach we presented here could also build the foundation of a filtering approach, where future work could address how exactly a complete OCED can be constructed based on queries.

Furthermore, analyzing the expressiveness of the proposed approach and systematically comparing it to related work would be of interest.

Finally, evaluating the presented approach on more real-life datasets would be of high importance, both in terms of evaluating the practical applicability of the approach conceptually and investigating the runtime performance and scalability on even larger data. Depending on the size of the considered dataset, future work could also consider evaluating queries in a distributed manner across multiple machines, expanding on the existing parallelization of the implementation.

Bibliography

[1] Wil M. P. van der Aalst. Object-Centric Process Mining: Dealing with Divergence and Convergence in Event Data. In Peter Csaba Ölveczky and Gwen Salaün, editors, *Software Engineering and Formal Methods - 17th International Conference, SEFM 2019, Oslo, Norway, September 18-20, 2019, Proceedings*, volume 11724 of *Lecture Notes in Computer Science*, pages 3–25. Springer, 2019. doi: 10.1007/978-3-030-30446-1_1. URL `https://doi.org/10.1007/978-3-030-30446-1_1`.

[2] Alessandro Berti, István Koren, Jan Niklas Adams, Gyunam Park, Benedikt Knopp, Nina Graves, Majid Rafiei, Lukas Liß, Leah Tacke genannt Unterberg, Yisong Zhang, Christopher T. Schwanen, Marco Pegoraro, and Wil M. P. van der Aalst. OCEL (Object-Centric Event Log) 2.0 Specification. *CoRR*, abs/2403.01975, 2024. doi: 10.48550/ARXIV.2403.01975. URL `https://doi.org/10.48550/arXiv.2403.01975`.

[3] Alessandro Berti, Marco Montali, and Wil M. P. van der Aalst. Advancements and Challenges in Object-Centric Process Mining: A Systematic Literature Review. *CoRR*, abs/2311.08795, 2023. doi: 10.48550/ARXIV.2311.08795. URL `https://doi.org/10.48550/arXiv.2311.08795`.

[4] Wil M. P. van der Aalst. Object-Centric Process Mining: Unraveling the Fabric of Real Processes. *Mathematics*, 11(12):2691, January 2023. ISSN 2227-7390. doi: 10.3390/math11122691. URL `https://www.mdpi.com/2227-7390/11/12/2691`.

[5] Wil M. P. van der Aalst and Alessandro Berti. Discovering Object-centric Petri Nets. *Fundam. Informaticae*, 175(1-4):1–40, 2020. doi: 10.3233/FI-2020-1946.

[6] Gyunam Park and Wil M. P. van der Aalst. Monitoring Constraints in Business Processes Using Object-Centric Constraint Graphs. In Marco Montali, Arik Senderovich, and Matthias Weidlich, editors, *Process Mining Workshops - ICPM 2022 International Workshops, Bozen-Bolzano, Italy, October 23-28, 2022, Revised Selected Papers*, volume 468 of *Lecture Notes in Business Information Processing*, pages 479–492. Springer, 2022. doi: 10.1007/978-3-031-27815-0_35. URL `https://doi.org/10.1007/978-3-031-27815-0_35`.

[7] Tijs Slaats. Declarative and Hybrid Process Discovery: Recent Advances and Open Challenges. *J. Data Semant.*, 9(1):3–20, 2020. doi: 10.1007/S13740-020-00112-9. URL `https://doi.org/10.1007/s13740-020-00112-9`.

[8] Maja Pesic and Wil M. P. van der Aalst. A Declarative Approach for Flexible Business Processes Management. In Johann Eder and Schahram Dustdar, editors, *Business Process Management Workshops, BPM 2006 International Workshops, BPD, BPI, ENEI, GPWW, DPM, Semantics4ws, Vienna, Austria, September 4-7, 2006, Proceedings*, volume 4103 of *Lecture*

Notes in Computer Science, pages 169–180. Springer, 2006. doi: 10.1007/11837862_18. URL https://doi.org/10.1007/11837862_18.

[9] Wil M. P. van der Aalst and Maja Pesic. DecSerFlow: Towards a Truly Declarative Service Flow Language. In Mario Bravetti, Manuel Núñez, and Gianluigi Zavattaro, editors, *Web Services and Formal Methods, Third International Workshop, WS-FM 2006 Vienna, Austria, September 8-9, 2006, Proceedings*, volume 4184 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 2006. doi: 10.1007/11841197_1. URL https://doi.org/10.1007/11841197_1.

[10] Maja Pesic, Helen Schonenberg, and Wil M. P. van der Aalst. DECLARE: Full Support for Loosely-Structured Processes. In *11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007), 15-19 October 2007, Annapolis, Maryland, USA*, pages 287–300. IEEE Computer Society, 2007. doi: 10.1109/EDOC.2007.14.

[11] Evelina Lamma, Paola Mello, Marco Montali, Fabrizio Riguzzi, and Sergio Storari. Inducing Declarative Logic-Based Models from Labeled Traces. In Gustavo Alonso, Peter Dadam, and Michael Rosemann, editors, *Business Process Management, 5th International Conference, BPM 2007, Brisbane, Australia, September 24-28, 2007, Proceedings*, volume 4714 of *Lecture Notes in Computer Science*, pages 344–359. Springer, 2007. doi: 10.1007/978-3-540-75183-0_25. URL https://doi.org/10.1007/978-3-540-75183-0_25.

[12] Andrew Cropper and Sebastijan Dumancic. Inductive Logic Programming At 30: A New Introduction. *J. Artif. Intell. Res.*, 74:765–850, 2022. doi: 10.1613/JAIR.1.13507. URL https://doi.org/10.1613/jair.1.13507.

[13] Fabrizio Maria Maggi, Arjan J. Mooij, and Wil M. P. van der Aalst. User-guided discovery of declarative process models. 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 192–199. IEEE, 2011. doi: 10.1109/CIDM.2011.5949297.

[14] Rakesh Agrawal and Ramakrishnan Srikant. Fast Algorithms for Mining Association Rules. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 487–499. Morgan Kaufmann, 1994. URL http://www.vldb.org/conf/1994/P487.PDF.

[15] Fabrizio Maria Maggi, R. P. Jagadeesh Chandra Bose, and Wil M. P. van der Aalst. Efficient Discovery of Understandable Declarative Process Models from Event Logs. In Jolita Ralyté, Xavier Franch, Sjaak Brinkkemper, and Stanislaw Wrycza, editors, *Advanced Information Systems Engineering - 24th International Conference, CAiSE 2012, Gdansk, Poland, June 25-29, 2012. Proceedings*, volume 7328 of *Lecture Notes in Computer Science*, pages 270–285. Springer, 2012. doi: 10.1007/978-3-642-31095-9_18. URL https://doi.org/10.1007/978-3-642-31095-9_18.

[16] Fabrizio Maria Maggi, Claudio Di Ciccio, Chiara Di Francescomarino, and Taavi Kala. Parallel algorithms for the automated discovery of declarative process models. *Inf. Syst.*, 74 (Part):136–152, 2018. doi: 10.1016/J.IS.2017.12.002. URL https://doi.org/10.1016/j.is.2017.12.002.

[17] Claudio Di Ciccio and Massimo Mecella. On the Discovery of Declarative Control Flows for Artful Processes. *ACM Trans. Manag. Inf. Syst.*, 5(4):24:1–24:37, 2015. doi: 10.1145/2629447.

[18] Thomas T. Hildebrandt and Raghava Rao Mukkamala. Declarative Event-Based Workflow as Distributed Dynamic Condition Response Graphs. In Kohei Honda and Alan Mycroft, editors, *Proceedings Third Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES 2010, Paphos, Cyprus, 21st March 2010*, volume 69 of *EPTCS*, pages 59–73, 2010. doi: 10.4204/EPTCS.69.5.

[19] Christoffer Olling Back, Tijs Slaats, Thomas Troels Hildebrandt, and Morten Marquard. DisCoveR: Accurate and efficient discovery of declarative process models. *Int. J. Softw. Tools Technol. Transf.*, 24(4):563–587, 2022. doi: 10.1007/S10009-021-00616-0. URL `https://doi.org/10.1007/s10009-021-00616-0`.

[20] Søren Debois, Thomas T. Hildebrandt, Paw Høvsgaard Laursen, and Kenneth Ry Ulrik. Declarative process mining for DCR graphs. In Ahmed Seffah, Birgit Penzenstadler, Carina Alves, and Xin Peng, editors, *Proceedings of the Symposium on Applied Computing, SAC 2017, Marrakech, Morocco, April 3-7, 2017*, pages 759–764. ACM, 2017. doi: 10.1145/3019612.3019622.

[21] Viktorija Nekrasaite, Andrew Tristan Parli, Christoffer Olling Back, and Tijs Slaats. Discovering Responsibilities with Dynamic Condition Response Graphs. In Paolo Giorgini and Barbara Weber, editors, *Advanced Information Systems Engineering - 31st International Conference, CAiSE 2019, Rome, Italy, June 3-7, 2019, Proceedings*, volume 11483 of *Lecture Notes in Computer Science*, pages 595–610. Springer, 2019. doi: 10.1007/978-3-030-21290-2_37. URL `https://doi.org/10.1007/978-3-030-21290-2_37`.

[22] Paul Pichler, Barbara Weber, Stefan Zugal, Jakob Pinggera, Jan Mendling, and Hajo A. Reijers. Imperative versus Declarative Process Modeling Languages: An Empirical Investigation. In Florian Daniel, Kamel Barkaoui, and Schahram Dustdar, editors, *Business Process Management Workshops - BPM 2011 International Workshops, Clermont-Ferrand, France, August 29, 2011, Revised Selected Papers, Part I*, volume 99 of *Lecture Notes in Business Information Processing*, pages 383–394. Springer, 2011. doi: 10.1007/978-3-642-28108-2_37. URL `https://doi.org/10.1007/978-3-642-28108-2_37`.

[23] Cornelia Haisjackl, Irene Barba, Stefan Zugal, Pnina Soffer, Irit Hadar, Manfred Reichert, Jakob Pinggera, and Barbara Weber. Understanding Declare models: Strategies, pitfalls, empirical results. *Softw. Syst. Model.*, 15(2):325–352, 2016. doi: 10.1007/S10270-014-0435-Z. URL `https://doi.org/10.1007/s10270-014-0435-z`.

[24] Artem Polyvyanyy, Chun Ouyang, Alistair Barros, and Wil M. P. van der Aalst. Process querying: Enabling business intelligence through query-based process analytics. *Decis. Support Syst.*, 100:41–56, 2017. doi: 10.1016/J.DSS.2017.04.011. URL `https://doi.org/10.1016/j.dss.2017.04.011`.

[25] Bettina Fazzinga, Sergio Flesca, Filippo Furfaro, Elio Masciari, Luigi Pontieri, and Chiara Pulice. A Framework Supporting the Analysis of Process Logs Stored in Either Relational or NoSQL DBMSs. In Floriana Esposito, Olivier Pivert, Mohand-Saïd Hacid, Zbigniew W. Ras, and Stefano Ferilli, editors, *Foundations of Intelligent Systems - 22nd International Symposium, ISMIS 2015, Lyon, France, October 21-23, 2015, Proceedings*, volume 9384 of *Lecture Notes in Computer Science*, pages 52–58. Springer, 2015. doi: 10.1007/978-3-319-25252-0_6. URL `https://doi.org/10.1007/978-3-319-25252-0_6`.

[26] José Miguel Pérez-Álvarez, Antonio Cancela Díaz, Luisa Parody, Antonia M. Reina Quintero, and María Teresa Gómez-López. Process Instance Query Language and the Process Querying Framework. In Artem Polyvyanyy, editor, *Process Querying Methods*, pages 85–111. Springer, 2022. doi: 10.1007/978-3-030-92875-9_4. URL `https://doi.org/10.1007/978-3-030-92875-9_4`.

[27] Thomas Vogelgesang, Jessica Ambrosy, David Becher, Robert Seilbeck, Jerome Geyer-Klingeberg, and Martin Klenk. Celonis PQL: A Query Language for Process Mining. In Artem Polyvyanyy, editor, *Process Querying Methods*, pages 377–408. Springer International Publishing, Cham, 2022. ISBN 978-3-030-92875-9. doi: 10.1007/978-3-030-92875-9_13. URL `https://doi.org/10.1007/978-3-030-92875-9_13`.

[28] Klaus Kammerer, Jens Kolb, and Manfred Reichert. PQL - A Descriptive Language for Querying, Abstracting and Changing Process Models. In Khaled Gaaloul, Rainer Schmidt, Selmin Nurcan, Sérgio Guerreiro, and Qin Ma, editors, *Enterprise, Business-Process and Information Systems Modeling - 16th International Conference, BPMDS 2015, 20th International Conference, EMMSAD 2015, Held at CAiSE 2015, Stockholm, Sweden, June 8-9, 2015, Proceedings*, volume 214 of *Lecture Notes in Business Information Processing*, pages 135–150. Springer, 2015. doi: 10.1007/978-3-319-19237-6_9. URL `https://doi.org/10.1007/978-3-319-19237-6_9`.

[29] Stefan Esser and Dirk Fahland. Multi-Dimensional Event Data in Graph Databases. *J. Data Semant.*, 10(1-2):109–141, 2021. doi: 10.1007/S13740-021-00122-1. URL `https://doi.org/10.1007/s13740-021-00122-1`.

[30] Boudewijn van Dongen. BPI Challenge 2017, February 2017. URL `https://data.4tu.nl/articles/_/12696884/1`.

[31] Wil M. P. van der Aalst, Alessandro Artale, Marco Montali, and Simone Tritini. Object-Centric Behavioral Constraints: Integrating Data and Declarative Process Modelling. In Alessandro Artale, Birte Glimm, and Roman Kontchakov, editors, *Proceedings of the 30th International Workshop on Description Logics, Montpellier, France, July 18-21, 2017*, volume 1879 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2017. URL `https://ceur-ws.org/Vol-1879/paper51.pdf`.

[32] Baoxin Xiu, Guangming Li, and Yidan Li. Discovery of Object-Centric Behavioral Constraint Models With Noise. *IEEE Access*, 10:88769–88786, 2022. doi: 10.1109/ACCESS.2022.3199345.

[33] Jan Niklas Adams, Daniel Schuster, Seth Schmitz, Günther Schuh, and Wil M. P. van der Aalst. Defining Cases and Variants for Object-Centric Event Data. In Andrea Burattin, Artem Polyvyanyy, and Barbara Weber, editors, *4th International Conference on Process Mining, ICPM 2022, Bolzano, Italy, October 23-28, 2022*, pages 128–135. IEEE, 2022. doi: 10.1109/ICPM57379.2022.9980730.

[34] Tian Li, Gyunam Park, and Wil M. P. van der Aalst. Checking Constraints for Object-Centric Process Executions. In Johannes De Smedt and Pnina Soffer, editors, *Process Mining Workshops - ICPM 2023 International Workshops, Rome, Italy, October 23-27, 2023, Revised Selected Papers*, volume 503 of *Lecture Notes in Business Information Processing*, pages 392–405. Springer, 2023. doi: 10.1007/978-3-031-56107-8_30. URL `https://doi.org/10.1007/978-3-031-56107-8_30`.

[35] Alessandro Berti. Filtering and Sampling Object-Centric Event Logs, May 2022. URL http://arxiv.org/abs/2205.01428.

[36] Benedikt Knopp and Wil M.P. van der Aalst. Order management object-centric event log in OCEL 2.0 standard, October 2023. URL https://doi.org/10.5281/zenodo.8428112.

[37] Jonathan P. Bowen. The Z Notation: Whence the Cause and Whither the Course? In Zhiming Liu and Zili Zhang, editors, *Engineering Trustworthy Software Systems - First International School, SETSS 2014, Chongqing, China, September 8-13, 2014. Tutorial Lectures*, volume 9506 of *Lecture Notes in Computer Science*, pages 103–151. Springer, 2014. doi: 10.1007/978-3-319-29628-9_3. URL https://doi.org/10.1007/978-3-319-29628-9_3.

[38] Wil M. P. van der Aalst. *Process Mining - Data Science in Action, Second Edition.* Springer, 2016. ISBN 978-3-662-49850-7. doi: 10.1007/978-3-662-49851-4.

[39] Aaron Küsters and Wil M. P. van der Aalst. Developing a High-Performance Process Mining Library with Java and Python Bindings in Rust. *CoRR*, abs/2401.14149, 2024. doi: 10.48550/ARXIV.2401.14149. URL https://doi.org/10.48550/arXiv.2401.14149.

[40] Shahrzad Khayatbashi, Olaf Hartig, and Amin Jalali. BPI Challenge 2017 (OCEL), August 2023. URL https://data.4tu.nl/datasets/6889ca3f-97cf-459a-b630-3b0b0d8664b5/1.

[41] Gyunam Park and Leah Tacke genannt Unterberg. Procure-To-Payment (P2P) Object-centric Event Log in OCEL 2.0 Standard, October 2023. URL https://zenodo.org/records/8412920.

[42] Benedikt Knopp and Nina Graves. Container Logistics Object-centric Event Log, August 2023. URL https://zenodo.org/records/8428084.

Acknowledgments

First, I would like to thank Prof. Wil van der Aalst for providing the opportunity to write my thesis at PADS, allowing me to dive deeper into this research field. Moreover, I would additionally like to thank him for also supervising this thesis project. Throughout writing this thesis, he provided me with excellent feedback and ideas, but especially also a lot of freedom to explore different directions.

I also want to thank Prof. Stefan Decker for accepting the position as second examiner for this thesis.

Last but not least, I would like to thank the people around me for their continued support, motivation, and love, especially Alina, who spent hours proofreading this thesis, as well as Leya, Angela, and Walter.